MPC 技术笔记---鞠春宇-20200524

一、MPC 应用背景

在真正的汽车中，致动命令不会立即执行-随着命令在整个系统中传播，会有延迟。实际的延迟可能约为 100 毫秒。这是一个称为“延迟”的问题，对于某些控制器（如 PID 控制器）来说，这是一个艰巨的挑战。但是模型预测控制器可以很好地适应，因为我们可以在系统中对此延迟进行建模。

1、PID 控制器

PID 控制器将计算相对于当前状态的误差，但是将在车辆处于未来（可能不同）的状态时执行致动。有时可能会导致不稳定。PID 控制器可以尝试根据将来的误差来计算控制输入，但是如果没有车辆模型，则不太可能准确无误。

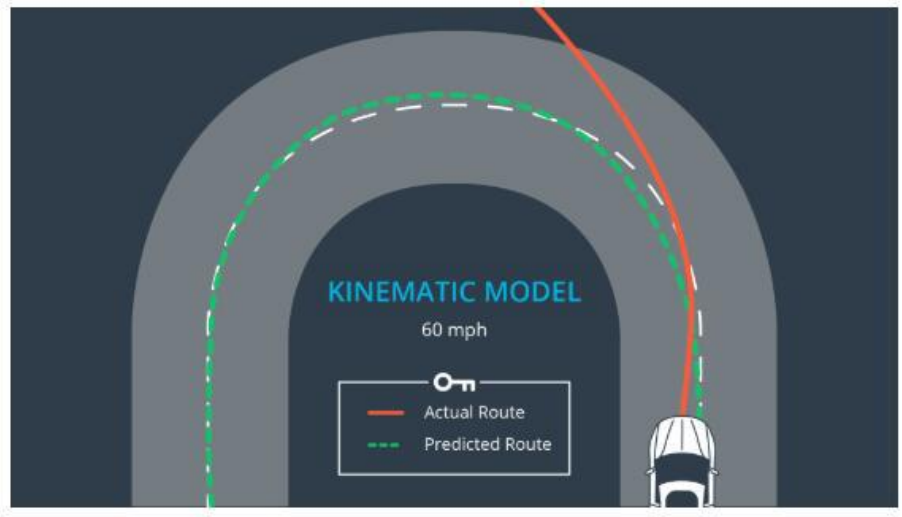
2、模型预测控制

延迟的一个重要因素是执行器动态特性。例如，从您指令转向角度到实际达到该角度之间所经过的时间。这可以通过简单的动态系统轻松建模，并整合到车辆模型中。一种方法是使用车辆模型从等待状态的持续时间的当前状态开始进行仿真。仿真产生的状态是 MPC 的新初始状态。因此，与 PID 控制器相比，MPC 通过明确考虑到延迟可以更加有效地处理延迟。接下来，我们将深入了解实施 MPC！

二、补充：车的运动学模型和动力学模型

1、运动学模型

运动学模型是忽略轮胎力，重力和质量的动力学模型的简化。这种简化降低了模型的准确性，但也使它们更易于处理。在低速和中速时，运动学模型通常近似于实际的车辆动力学。

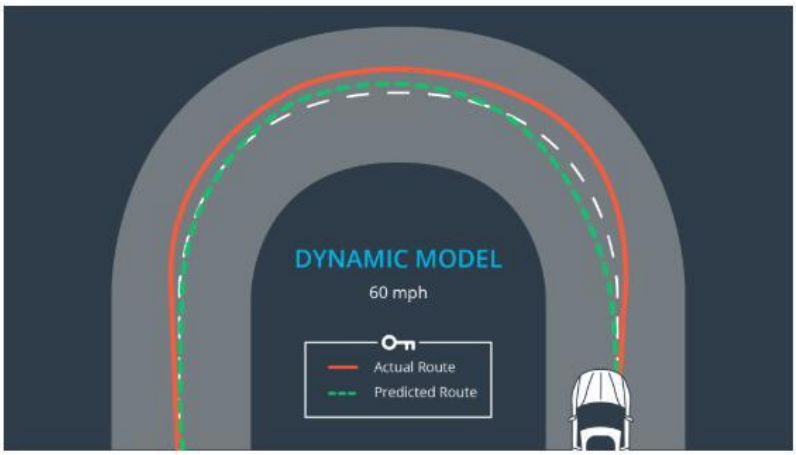


The vehicle is forced off the road due to forces not accounted for.

由于未考虑到的力，车辆被迫离开道路。

2、动态模型

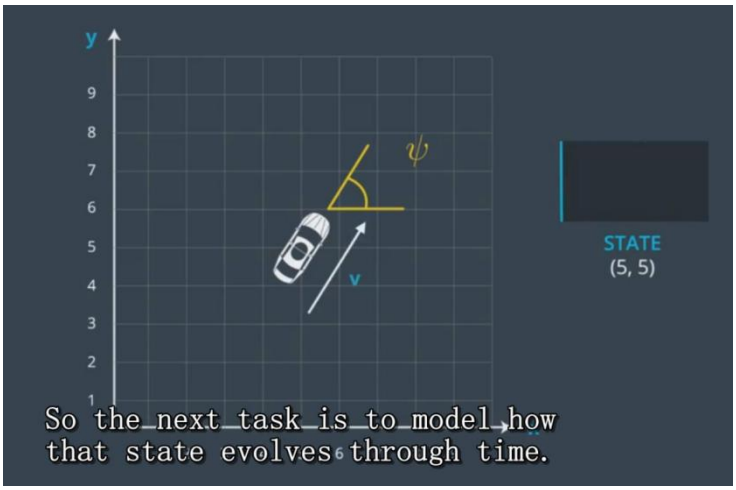
动态模型旨在尽可能地体现实际的车辆动力学。它们可能包括轮胎力，纵向力和横向力，惯性，重力，空气阻力，阻力，质量以及车辆的几何形状。并非所有动态模型都是一样的！有些人可能比其他人考虑更多这些因素。先进的动态模型甚至考虑了内部车辆的力量-例如，底盘悬架的响应速度。



The dynamic model is able to stay on the road, knowledge of forces is embedded in the model.

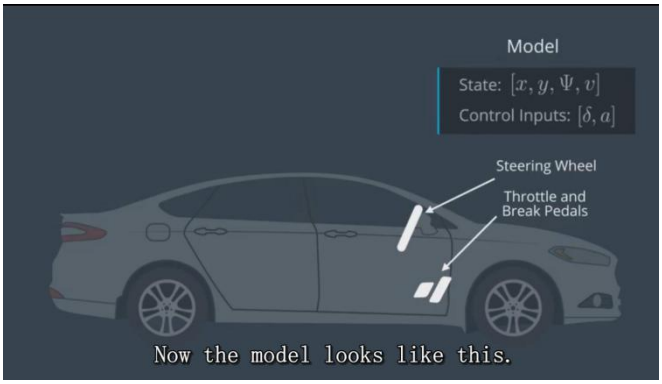
动态模型能够持续行驶，而力的知识则嵌入到模型中。

下一步是研究出模型去建立车的状态与时间是一种什么样的关系。

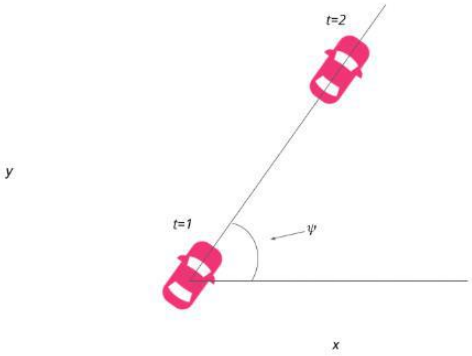


3、建立运动学模型

大多数车有三个执行器：方向盘、油门踏板和刹车踏板。这里我们将油门踏板和刹车踏板作为成一个执行器来处理，负值表示制动，正值表示加速。二者分别记为转向角度 δ 和加速度 a。现在的车输入控制和状态如下图所示。



建立运动学模型



（1）对于 x,y 有　　X=x+v*cos(ψ)

　　Y=y+v*sin(ψ)

（2）接下来，让我们将注意力转向 ψ：

$$\psi = \psi + \frac{v}{L_f} * \delta * dt$$

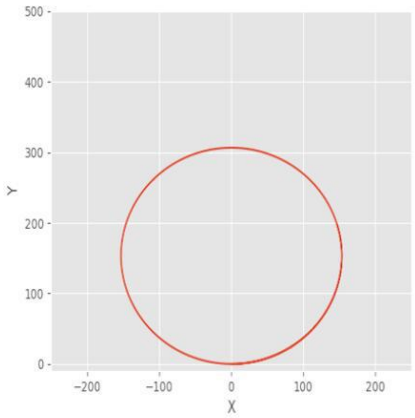
简而言之，我们将转向角的乘积 δ 添加到 ψ 。

L_f 衡量车辆前部与其重心之间的距离。车辆越大，转弯速度越慢。

如果您驾驶车辆，则应该清楚地知道，以较高的速度转弯比以较低的速度转弯更快。这就是为什么 v 是包含在更新中的原因。

关于绕圈行驶的问题，这实际上是测试模型有效性的一种好方法！ 如果以恒定的速度和转向角围绕测试车辆行驶而产生的圆的半径与模拟中模型的圆相似，那么您就在正确的轨道上。

下面是该项目如何确定 L_f 的值！



在上图中，车辆从原点开始，以 0 度定向，然后模拟了 δ 值为 1 度和 L_f 值 2.67。

（3）最后，让我们看一下速度 v 的建模方式：

　　v = v + a * dt

其中 a 可以取值介于-1 和 1 之间（包括-1 和 1）。

太棒了！现在，我们根据先前的状态和当前的执行器输入定义了状态，执行器以及状态如何随时间变化。

这里是：

$$x_{t+1} = x_t + v_t * cos(\psi_t) * dt$$

$$y_{t+1} = y_t + v_t * sin(\psi_t) * dt$$

$$\psi_{t+1} = \psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

$$v_{t+1} = v_t + a_t * dt$$

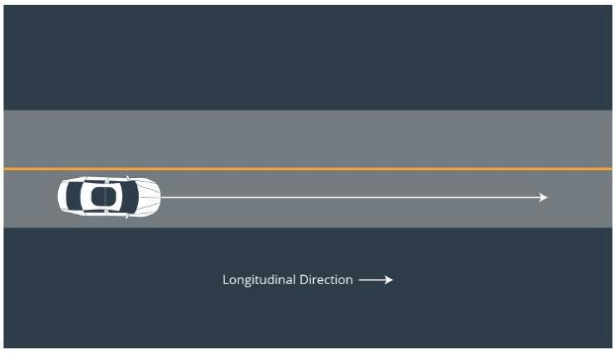
```cpp
// In this quiz you'll implement the global kinematic model.
#include <math.h>
#include <iostream>
#include "Dense"

//
// Helper functions
//
double pi() { return M_PI; }
double deg2rad(double x) { return x * pi() / 180; }
double rad2deg(double x) { return x * 180 / pi(); }

const double Lf = 2;

// TODO: Implement the global kinematic model.
// Return the next state.
//
// NOTE: state is [x, y, psi, v]
// NOTE: actuators is [delta, a]
Eigen::VectorXd globalKinematic(Eigen::VectorXd state,
                                Eigen::VectorXd actuators, double dt) {
  Eigen::VectorXd next_state(state.size());

  //TODO complete the next_state calculation ...

  next_state[0] = state[0] + state[3] * cos(state[2])*dt;
  next_state[1] = state[1] + state[3] * sin(state[2])*dt;
  next_state[2] = state[2] + state[3]/Lf *actuators[0]*dt;
  next_state[3] = state[3] + actuators[1]*dt;

  return next_state;
```

感知模块：感知无人驾驶汽车周围环境。
定位模块：通过将模型与地图进行比较来确定自身位置。
路径规划模块：使用环境信息进行路径规划。
控制：控制回路应用执行器跟踪这一轨迹。通过参考轨迹作为多项式轨迹 $a_0+a_1x+a_2x^2+a_3x^3$，三次多项式是一种常见的多项式，可以适用于大数道路。

4、减少跟踪的轨迹和参考的轨迹误差，我们首先要预测出未来的时刻与跟踪的轨迹偏差，然后再改变控制输入。

通过将围绕这些错误的运动学模型作为新的状态向量，我们可以捕获我们感兴趣的错误随时间变化的方式。新状态为[x，y，ψ，v，cte，eψ]。

假设车辆沿直线行驶，并且纵向与 x 轴相同。



（1）跨轨误差：

然后，我们可以将道路中心与车辆位置之间的误差表示为交叉轨迹误差：

$$cte_{t+1} = cte_t + v_t * sin(e\psi_t) * dt$$

在这种情况下，cte 可以表示为线与当前车辆位置 y 之差。 假设参考线是一阶多项式 f：

$$cte_t = f(x_t) - y_t$$

如果我们将 cte_t 替换回原始方程式，则结果为：

$$cte_{t+1} = f(x_t) - y_t + (v_t * sin(e\psi_t) * dt)$$

这可以分为两部分：

1）$f（x_t）-y_t$ 是当前的跨轨错误。

2）$v_t*sin(e\Psi_t)$是由于车辆运动引起的误差变化。

（2）方向误差

好的，现在让我们看一下方向错误：

$$e\psi_{t+1} = e\psi_t + \frac{v_t}{L_f} * \delta_t * dt$$

更新规则与ψ基本相同。

$e\psi_t$ 从当前方向减去所需的方向：

$$e\psi_t = \psi_t - \psi des_t$$

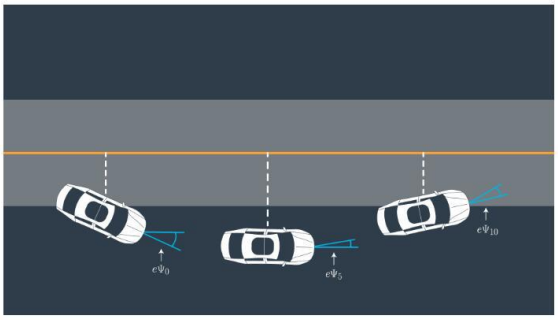
我们已经知道ψt），因为它是我们状态的一部分。 我们尚不知道ψdest（期望的的 psi）-到目前为止，我们所要遵循的只是多项式。 dest 可以计算为在 x 处评估的多项式 f 的切向角 x_t，arctan（f’（x_t））。 f’是多项式的导数。

有：

$$e\psi_{t+1} = \psi_t - \psi des_t + (\frac{v_t}{L_f} * \delta_t * dt)$$

Similarly to the cross track error this can be interpreted as two parts:

1. $\psi_t - \psi des_t$ being current orientation error.
2. $\frac{v_t}{L_f} * \delta_t * dt$ being the change in error caused by the vehicle's movement.

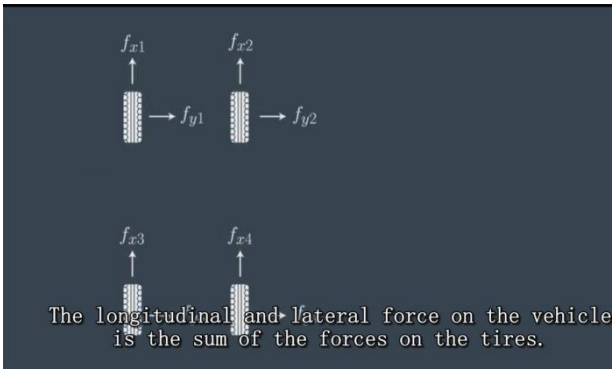


白虚线是交叉轨道误差。

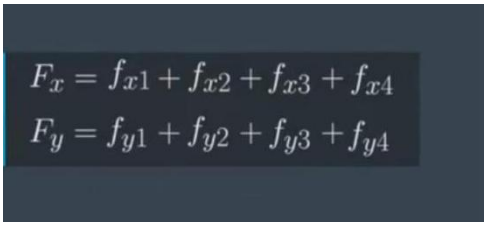
5 动态模型建立

当我们考虑很多动态力，比如车胎与地面的力量时。一般把所有力考虑进去主要有如下两种：横向力和纵向力。横向力使汽车前进后退，纵向力式汽车左右推动。

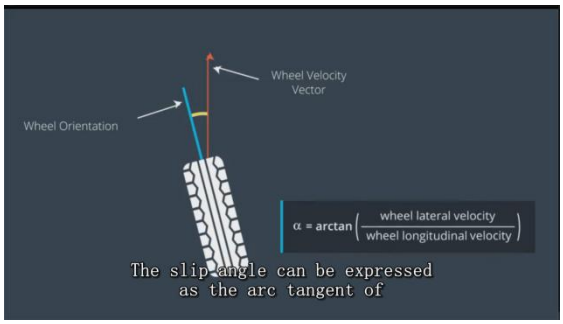
车辆的纵向力和横向力是轮胎受力总和。



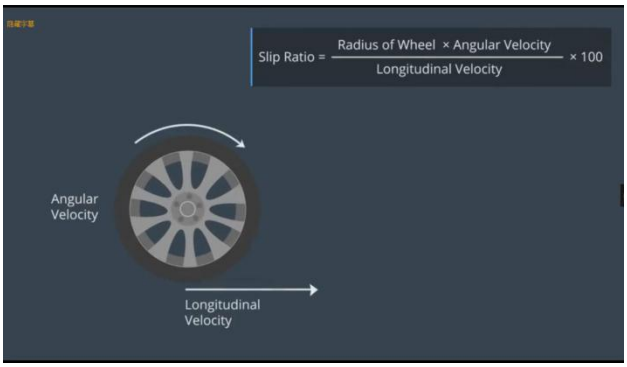
此时的横向力和纵向力为：



滑动角度：滑动角度的产生可以说是反应转弯的方式。



滑移率：车轮的速度和预期的纵向速度不一致的时候



二、参考状态

成本函数的一个好的开始是考虑您想最小化的误差。 例如，测量距车道中心的偏移量，其中车道中心可以称为参考状态或所需状态。我们先前在状态向量中捕获了两个错误：cte 和 eψ。理想情况下，这两个误差均为 0-从实际的车辆位置和航向到所需的位置和航向都没有差异。我们的成本应该是这些误差与 0 相差多少的函数。

这是一个如何在每个时间步增加成本的示例：

```
double cost = 0;for (int t = 0; t < N; t++) {

    cost += pow(cte[t], 2);

    cost += pow(epsi[t], 2);

}
```

三、汽车行驶过程中突然停止了，没有跟踪我们要求的速度 35km/h 如何处理？

1、代价函数限制

1） 采取惩罚函数对其惩罚

$Cost+=(v(t)-35)^2$

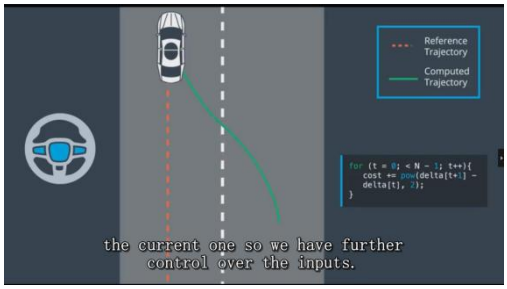
2） 当前车的位置和目的地之间的距离（欧式距离）

成本函数不仅仅包括车辆的状态，还包括控制输入。这里要做的是惩罚输入的大小和方向。

3） 方向盘转弯不要太急剧，有如下限制：

```
for (t = 0; < N - 1; t++){
   cost += pow(delta[t+1] -
   delta[t], 2);
}
```

Delta 代表输入方向盘转角度，整个式子代表上一个输入和当前的方向盘输入要平滑一点，所以用了了代价函数 cost 来限制这点。



2、长度和时长

预测范围是进行未来预测的持续时间。 我们将其称为 T。

T 是其他两个变量 N 和 dt 的乘积。N 是地平线上的时间步数。 dt 是两次致动之间经过的时间。 例如，如果 N 为 20，而 dt 为 0.5，则 T 为 10 秒。

N，dt 和 T 是超参数，您需要针对所构建的每个模型预测控制器进行调整。 但是，有一些通用准则。 **T 应该尽可能大，而 dt 应该尽可能小**。

这些准则会产生折衷。

地平线

在开车的情况下，T 最多应为几秒钟。 超越视野，环境将发生足够的变化，以至于无法预测未来。

时间步数

模型预测控制的目标是优化控制输入：[δ，a]。优化器将调整这些输入，直到找到控制输入的低成本向量为止。

此向量的长度由 N 确定：

$$[\delta_1, a_1, \delta_2, a2, ..., \delta N-1, a_{N-1}]$$

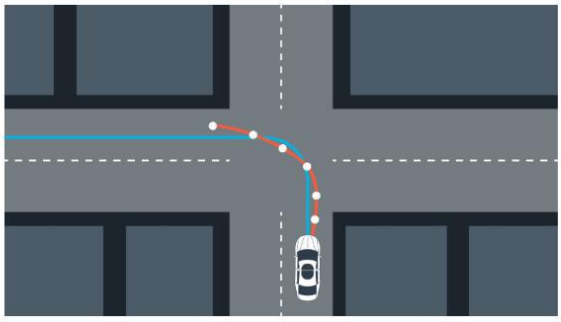
Thus N determines the number of variables optimized by the MPC. This is also the major driver of computational cost.

因此，N 决定了由 MPC 优化的变量数量。这也是计算成本的主要驱动力。

时间步长

MPC 尝试通过驱动之间的离散路径来近似连续参考轨迹。 较大的 dt 值导致较少的动作，这使得更难以精确地逼近连续的参考轨迹。 有时称为“离散化错误”。

设置 N，dt 和 T 的一种好方法是，首先确定 T 的合理范围，然后适当调整 dt 和 N，同时牢记每个因素的影响。



蓝线是参考轨迹，红线是模型预测控制所计算的轨迹。 在此示例中，地平线有 7 个台阶 N，白色小卵石之间的空间表示经过的时间 dt。

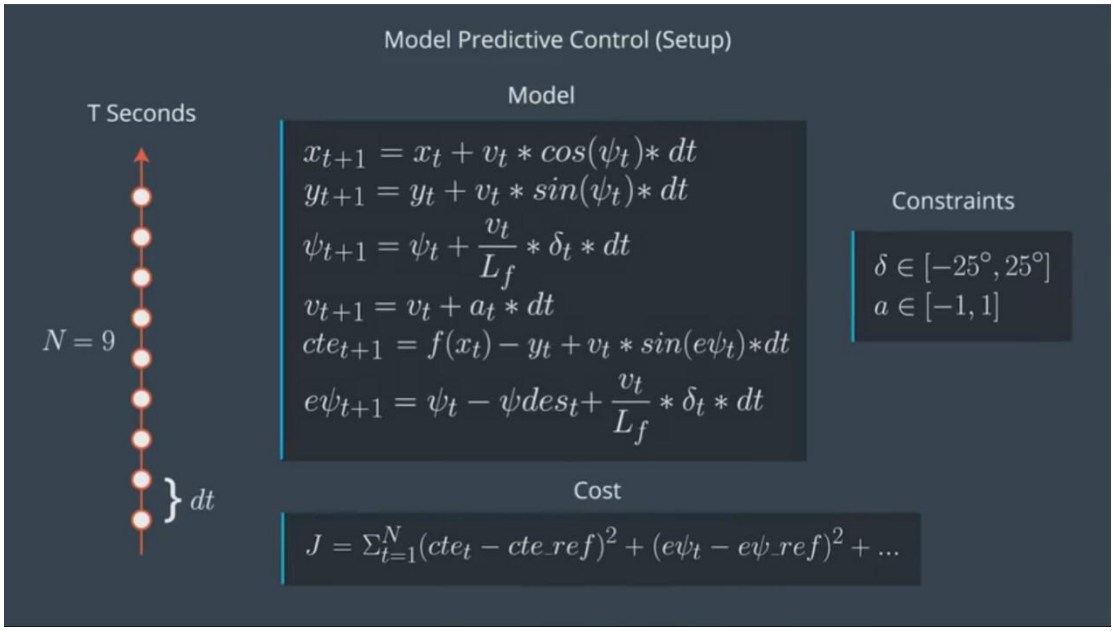
3、MPC 模型算法

MPC 使用优化器去寻找一个控制输入使其代价函数最小。实际上，我们只执行第一组控制输入，这会使车辆进行到一个新的状态，然后重复这个过程。

1、首先建立 MPC 闭环控制所需要的一切东西。这包括确定轨迹的持续时间 T(通过选择 dt 和 N 来确定)，

2、接下来，我们定义了车辆模型和约束条件，例如：实际的局限性。

3、最后我们确定代价函数。



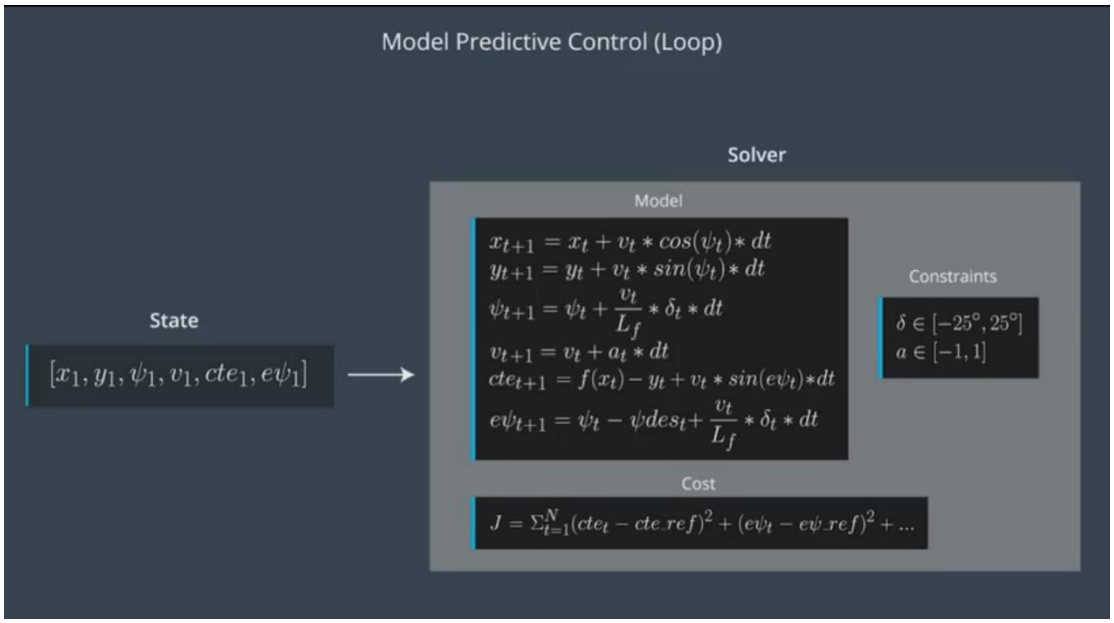
MPC（Loop）

1、首先把当前状态传递给 MPC 模型

2、调用优化求解器。求解器使用初始状态、模型限制条件、代价函数，返回一组控制向量，这个控制向量可以使代价函数最小。这个求解器，我们也称之为 IPOPT

3、然后使用第一组控制输入，并且不断循环这个过程。

见下图：



在本测验中，您将使用 MPC 沿直线跟踪轨迹。

步骤：

1 设置 N 和 dt。

2 使多项式适合航路点。

3 计算初始跨轨误差和方向误差值。

4 定义成本函数的组件（状态，执行器等）。 您可以使用前面讨论的方法，也可以根据自己的需要来编造一些东西！

5 定义模型约束。 这些是“车辆模型”模块中定义的状态更新方程式。

在开始之前，让我们看一下将用于本测验和以下项目的库。

(1)IPOPT https://github.com/coin-or/Ipopt

Ipopt 是我们将用于优化控制输入[δ]的工具

$[\delta_1, a1, ..., \delta N - 1, a_{N-1}]$.它能够找到局部最优值（非线性问题！），同时将约束直接设置为执行器和车辆模型定义的约束。

Ipopt 要求我们直接给它雅可比和海森-它不为我们计算它们。 因此，我们需要手动计算它们，或者让一个库来为我们做这些。 幸运的是，有一个名为 CppAD 的库就可以做到这一点。

（2）CppAD https://coin-or.github.io/CppAD/doc/cppad.htm

CppAD 是我们将用于自动区分的库。 通过使用 CppAD，我们不必手动计算导数，这很繁琐且容易出错。为了有效地使用 CppAD，我们必须使用其类型而不是常规的 double 或 std::vector 类型。此外，必须从 CppAD 调用数学函数。 这是调用 pow 的示例：

```
CppAD :: pow（x，2）；
// 代替
pow（x，2）；
```

幸运的是，大多数基本数学运算都已重载。因此，只要在 CppAD<double>而不是 double 上调用*，+，-，/就可以正常工作。 大多数操作都是为您完成的，并且在我们提供的代码中有一些示例可以借鉴。

(3)代码结构

我们已经为您填写了大部分测验入门代码。 该测验的目的实际上是使所有功能按预期工作。也就是说，解密入门代码的某些元素可能很棘手，因此我们将逐步引导您。

MPC.cpp 中有两个主要组件：

(1)vector<double> MPC::Solve（Eigen::VectorXd x0，Eigen::VectorXd coeffs）方法

（2）FG_eval class

（4）MPC ::solve

x0 是初始状态[x，y，ψ，v，cte，eψ]，coeffs 是拟合多项式的系数。该方法的主要工作是为 Ipopt 设置车辆模型约束（约束）和变量（变量）。

（5）变量

```
double x = x0[0];double y = x0[1];double psi = x0[2];double v = x0[3];double cte = x0[4];double epsi = x0[5];

...// Set the initial variable values
```

```
vars[x_start] = x;

vars[y_start] = y;

vars[psi_start] = psi;

vars[v_start] = v;

vars[cte_start] = cte;

vars[epsi_start] = epsi;
```

注意 Ipopt 希望所有约束和变量都是矢量。 例如，假设 N 为 5，则 vars 的结构为 38 个元素的向量：

vars[0],....,vars[4] -> $[x_1, ..., x_5]$

vars[5],....,vars[9] -> $[y_1, ..., y_5]$

vars[10],....,vars[14] -> $[\psi_1, ..., \psi_5]$

vars[15],....,vars[19] -> $[v_1, ..., v_5]$

vars[20],....,vars[24] -> $[cte_1, ..., cte_5]$

vars[25],....,vars[29] -> $[e\psi_1, ..., e\psi_5]$

vars[30],....,vars[33] -> $[\delta_1, ..., \delta_4]$

vars[34],....,vars[37] -> $[a_1, ..., a_4]$

然后，我们在变量上设置上下限。 在这里，我们将值 δ 的范围设置为以弧度为单位的[-25，25]：

```
    for (int i = delta_start; i < a_start; i++) {
    vars_lowerbound[i] = -0.436332;
    vars_upperbound[i] = 0.436332;
  }
```

约束条件

接下来，我们在约束上设置上下限。

考虑例如：

Xt+1= xt+ vt* cos（ψt）* dt

这表示 xt + 1 必须等于 xt+ vt* cos（ψt）* dt。 换句话说：Xt + 1-（xt+ vt* co（ψt*dt）= 0

上面的等式简化了约束的上限和下限：两者都必须为 0。

这也可以推广到其他方程式：

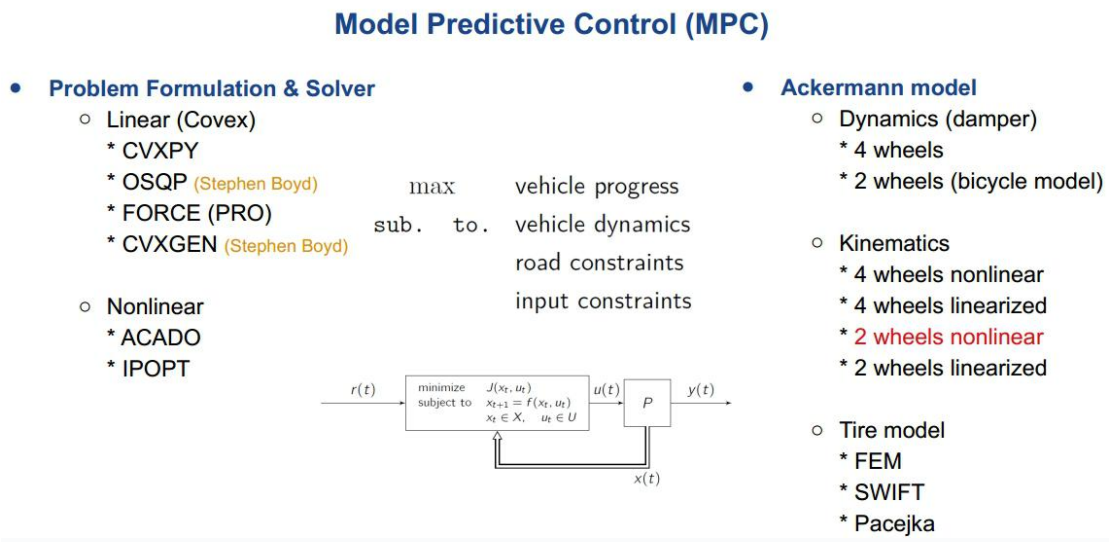
```
for (int i = 0; i < n_constraints; i++) {

  constraints_lowerbound[i] = 0;

  constraints_upperbound[i] = 0;

}
```

FG_eval

FG_eval 类具有构造函数：`FG_eval(Eigen::VectorXd coeffs) { this->coeffs = coeffs; }`

其中 coeffs 是拟合多项式的系数。 跨轨误差和航向误差方程将使用 coeffs。

FG_eval 类只有一种方法：



MPC 代码实现的优化原理：

优化目标函数：

$$J = \sum_{i=1}^{N=25}[(cte_i)^2 + (espi)^2 + (v - v\_ref)^2 + \delta^2 + a^2 + (a_k - a_{k-1})^2 + (\delta_k - \delta_{k-1})^2]$$

约束条件 1：

$$\begin{cases} -0.436332 \le \delta \le 0.436332 \\ -1.0 \le a \le 1.0 \end{cases}$$

其余 6*N 个约束条件如下：

$$\begin{cases} x(t+1) - (x(t) + v(t)\cos(\varphi(t))) = 0(\text{初始不为零}) \\ y(t+1) - (y(t) + v(t)\sin(\varphi(t))) = 0(\text{初始不为零}) \\ \varphi(t+1) - (\varphi(t) + \dfrac{v(t)}{L_f}\delta(t) \bullet dt) = 0(\text{初始值不为零}) \\ v(t+1) - (v(t) + a \bullet dt) = 0(\text{初始值不为零}) \\ cte(t+1) - (f(x(t)) - y(t) + v(t)\sin(\varphi(t)) \bullet dt) = 0(\text{初始值不为零}) \\ epsi(t+1) - (epsi(t) - psides(t) + v(t)\dfrac{\delta(t)}{L_f} \bullet dt) = 0(\text{初始值不为零}) \end{cases}$$

其中：$psides(t) = \arctan(f'(t))$