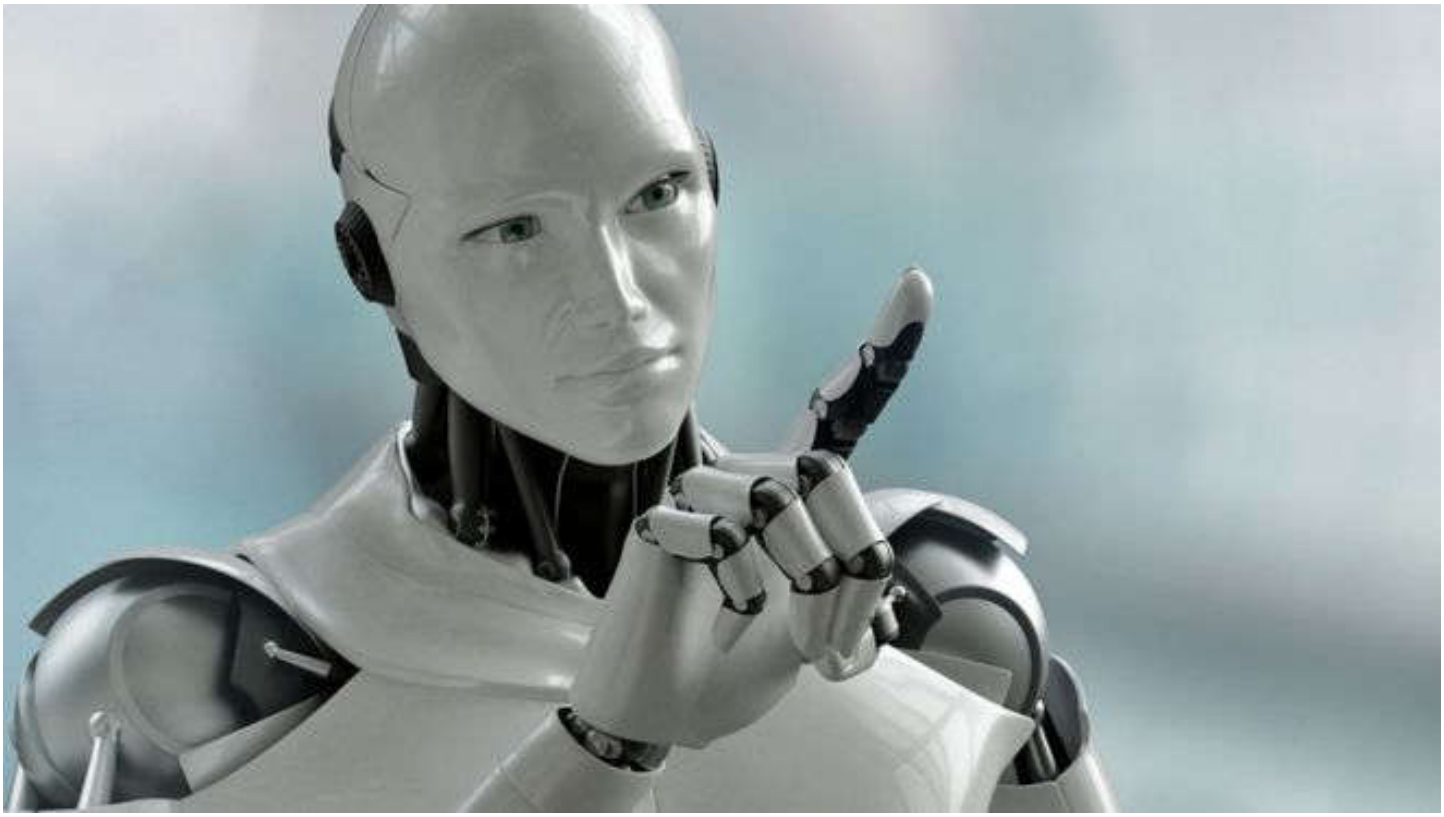


自动驾驶控制算法实例之模型预测控制（MPC）--从模型推导到代码实现（以Autoware为例）

随着自动驾驶技术的不断发展，模型预测控制（Model Predictive Control, MPC）作为一种非常有效的控制方法被广泛应用于车辆的横纵向控制研究中。不难发现，Autuware与Apollo的自动驾驶开源项目皆有利用MPC算法对车辆控制的应用与实现，而且只要是关于自动驾驶控制类JD都会强调需要了解MPC算法。因此，了解MPC算法在自动驾驶中的应用对于其控制原理的理解及车辆控制都能有一个比较深刻的认识。

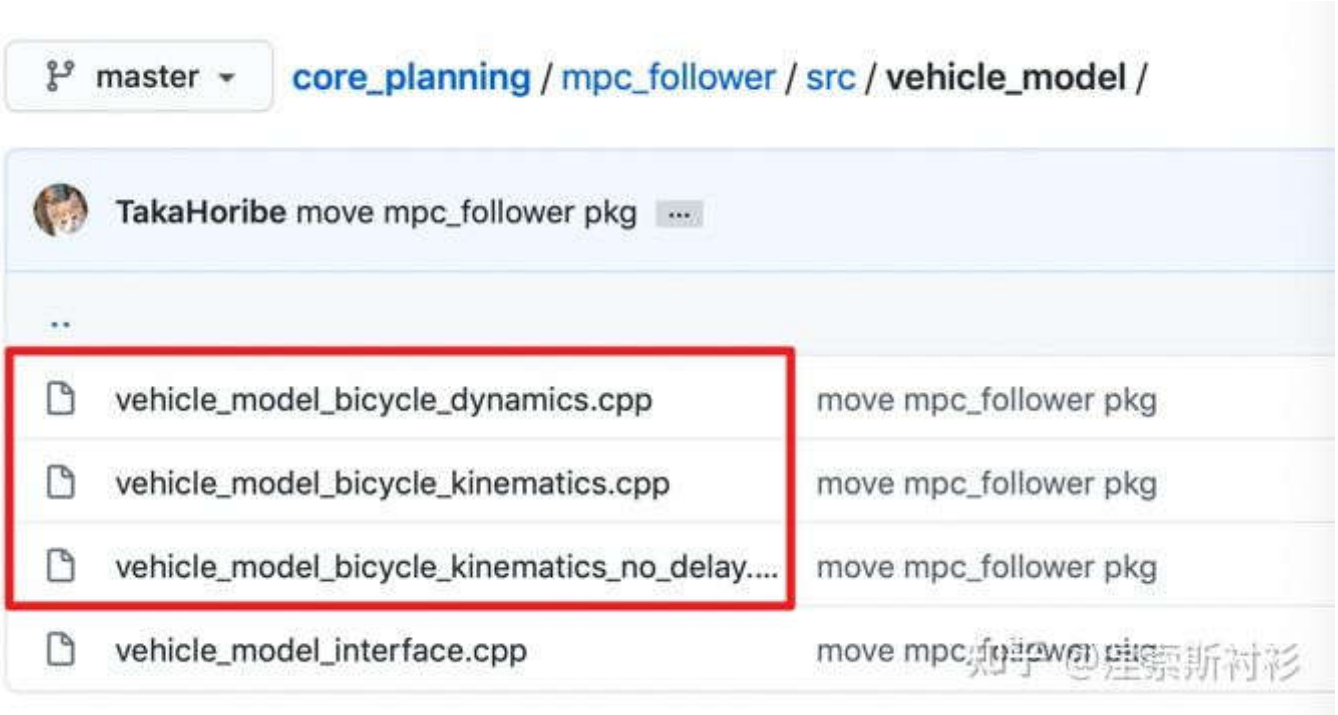
MPC控制算法在车辆控制中的应用主要分为如下几个步骤：

1. 控制模型的建立；
2. 控制模型的线性化；
3. 控制模型的离散化；
4. MPC控制算法的应用；

本篇以自动驾驶开源框架Autoware中的MPC控制算法为基础，系统介绍MPC控制算法的原理及实现流程，代码链接：Autoware-AI/**core_planning/mpc_follower/**。

一、 控制模型的建立（以运动学为例）

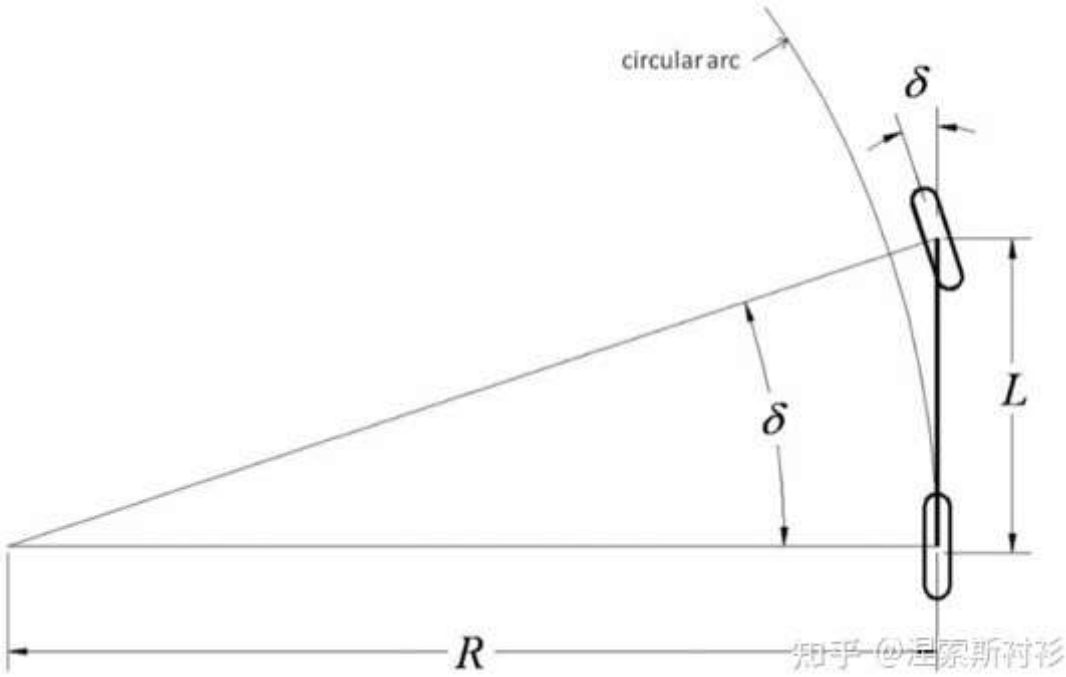
MPC（Model Predictive Control），顾名思义，模型预测控制，即模型是首要的。车辆模型的建立主要有动力学模型（ dynamic model）和运动学模型（ kinematics model ）两类。考虑到方向盘转角的执行滞后特性，在Autoware框架中，MPC算法运用的控制模型主要有三种：



在低速运行的场景中，运动学模型即可满足要求，且运动学模型相对简单，可以作为理解MPC算法很好的切入点。因此，本篇主要以运动学模型为基础，介绍MPC算法的实现流程。对于车辆运动学模型相关介绍有很多，在此不多做赘述，直接给出。

$$\begin{cases} \dot{x} = v * cos\theta \\ \dot{y} = v * sin\theta \\ \dot{\theta} = v * tan(\delta_f)/L \end{cases}$$

其中： (x, y) 分别代表车辆横纵向位置信息； θ 代表车辆的航向信息； δ_f 代表车辆的前轮转角； L 代表车辆的轴距； v 代表车速。



在vehicle_model_bicycle_kinematics_no_delay模型中，给出的模型如下：

$$\dot{x} = f(x,u) = \begin{cases} \dot{y} = v * sin\theta \\ \dot{\theta} = v * tan(\delta_f)/L \end{cases}$$

基于该模型不难得知，其状态变量为 $x = [y, \theta]$（横向位移，航向角），控制量 $u = [v, \delta_f]$ （车速，前轮转角）。**但是代码实现中状态量却为：[横向位移偏差，航向偏差]**，即**[err_lat, yaw_err]，主要是对该系统需要进行线性化处理。**

二、控制模型的线性化

根据上述车辆的运动学模型不难发现，该系统是一个非线性系统，无法运用线性时变的模型预测控制。因此，对于上述系统还得进行线性化处理，即需要进过如下的系统变换。

$$\dot{x} = f(x,u) \rightarrow \Delta\dot{x} = A * \Delta x + B * \Delta u + W * d$$

对于上述系统线性化处理的过程，可参考文章【扩展卡尔曼滤波(EKF)之非线性系统的线性化】，通过此文章可基本了解线性化的步骤。当然，为了与代码一

的模型相匹配，直接给出该非线性系统的线性化的Matlab实现过程，通过该代码即可求出上述的状态转移矩阵A与控制输入矩阵B以及W。

```
%%线性化模型
syms y theta delta_r;
syms v L
x = [y theta];
f1 = v*sin(theta);
f2 = v*tan(delta_r)/L;
f = [f1; f2];
F = jacobian(f,x)
G = jacobian(f,delta_r)
xp = [0 0]; %平衡点
A = subs(F, x , xp)
```

平衡点 $(0,0)$ 附近求导即可求得最后的线性化模型为：

$$\dot{x} = A * x + B * u + W * \delta_f ref$$

其中： $A = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix}$ ， $A = \begin{bmatrix} 0 \\ v/(L * cos^2(\delta_f)) \end{bmatrix}$ ， $W = \begin{bmatrix} 0 \\ -v/(L * cos^2(\delta_f)) \end{bmatrix}$ ， $\delta_f ref$ 为期望的前轮输入转角。

需要特别注意的是，经过线性化后的MPC系统状态量 $x = [err_{lat}, yaw_{err}]$ ，控制量 $u = [v, \delta_f]$ 。 但在该实例中，控制的输入量为 $u = [v, curvature]$ ，即[速度, 曲率]，因此 δ_f 还得利用阿克曼转角模型（如下所示）转换为期望的曲率。

三、控制模型的离散化

对于数字控制的系统，对于上述控制系统还得进行相应的离散化。对于控制系统的离散化，可参考文章【控制算法原理及实现之PID（以飞控为例）】。通过此文章，即可大致了解控制系统的离散化过程，控制系统离散化的形式有

种，在Autoware和Apollo开源框架中，主要是采用双线性（Tustin）变换以及欧拉法对系统进行离散化。

对于双线性变换，其变换公式如下：

$A(z) = A(s)|_{s=\frac{2z-1}{Tz+1}}$ ，其中，T为采样周期；

由此可知，其离散化后的状态转移矩阵 $A(z)$ 为：

```
Eigen::MatrixXd I = Eigen::MatrixXd::Identity(dim_x_, dim_x_);
   Az = (I - dt * 0.5 * As).inverse() * (I + dt * 0.5 * As); // bil
```

对于欧拉变化，其变换公式如下：

$A(z) = A(s)|_{s=\frac{z-1}{T}}$ ，其中，T为采样周期；

因此，可知 $A(z) = I + A(s) * T$ ， $B(z) = T * B(s)$ 。

在vehicle_model_bicycle_kinematics_no_delay模型中，主要采用欧拉变换来获得相应的离散化模型。最后，我们即可获取MPC所需的状态空间模型。

$$x(k+1) = A_d * x(k) + B_d * u(k) + W_d * \delta_f ref$$

其中，由阿克曼转角定理不难推知期望的前轮转角输入 $\delta_f ref$ 为：

```
void KinematicsBicycleModelNoDelay::calculateReferenceInput(Eigen::Ma
{
    Uref(0, 0) = std::atan(wheelbase_ * curvature_);
}
```

由上述的推导过程，不难理解其开源代码中模型的代码实现（如下所示）。至此，MPC算法所需的控制模型完成。

```
void KinematicsBicycleModelNoDelay::calculateDiscreteMatrix(Eigen::Ma
                                                            Eigen::MatrixXd
{
    auto sign = [](double x) { return (x > 0.0) - (x < 0.0); };

    /* Linearize delta around delta_r (referece delta) */
    double delta_r = atan(wheelbase_ * curvature_);
    if (abs(delta_r) >= steer_lim_)
        delta_r = steer_lim_ * (double)sign(delta_r);
    double cos_delta_r_squared_inv = 1 / (cos(delta_r) * cos(delta_r)

    Ad << 0.0, velocity_,
          0.0, 0.0;                  //线性化
    Eigen::MatrixXd I = Eigen::MatrixXd::Identity(dim_x_, dim_x_);
    Ad = I + Ad * dt;   //离散化

    Bd << 0.0, velocity_ / wheelbase_ * cos_delta_r_squared_inv;
    Bd *= dt;

    Cd << 1.0, 0.0,
          0.0, 1.0;

    Wd << 0.0,
        -velocity_ / wheelbase_ * delta_r * cos_delta_r_squared_inv;
    Wd *= dt;
}
```

四、 MPC控制算法的应用

未完待续。