Mini Project (Python)

Cheng shi

Table of contents:

1. Lateral control of driverless vehicle

1.1 Simplified two-wheeled bicycle model

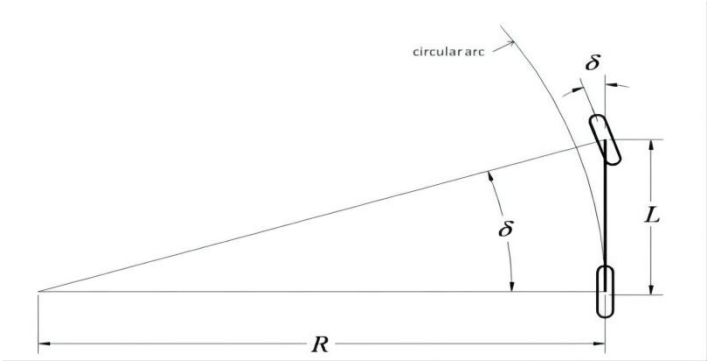
The following is a simplified two-wheeled bicycle model for later control.

First make assumptions about the model:

(1) The vehicle has no vertical motion, only the x-y plane motion is considered

(2) The vehicle movement is in line with the bicycle movement, and the center of the rear wheel is the research point

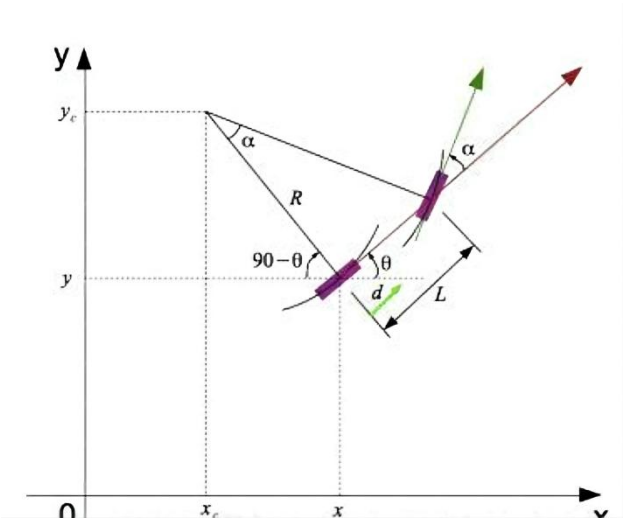
(3) The vehicle does not slip, and the direction of the body is the direction of speed



From simplified Ackerman to geometric relations:

$$\tan(\delta) = \frac{L}{R}$$

Create a Cartesian coordinate system:



The kinematic model state equation is obtained:

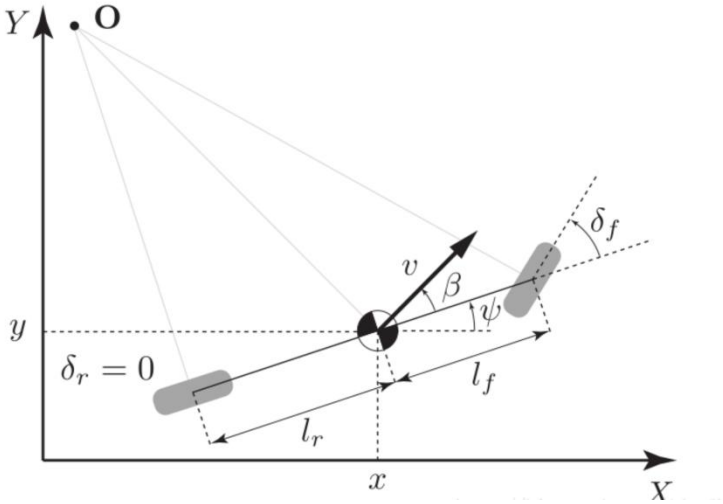
$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \sin\theta \\ \frac{\tan\delta}{L} \end{bmatrix} v$$

Python Code:

```
class vehicle:
  def __init__(self,x=0.0,y=0.0,yaw=0.0,v=0.0):
    self.x = x
    self.y = y
    self.yaw = yaw
    self.v = v
    self.dt = 0.1
  def update(self,a,delta):
    self.x = self.x + self.v*math.cos(self.yaw)*dt
    self.y = self.y + self.v*math.sin(self.yaw)*dt
    self.yaw = self.yaw + self.v/L*math.tan(delta)*dt
    self.v = self.v + a*dt
```

1.2 Vehicle kinematics model

The model with the center of mass of the vehicle as the research point is more in line with the motion characteristics of the vehicle.



Among them, O is the instantaneous center of rotation of the vehicle, β is the slip angle, which refers to the angle formed between the traveling direction of the vehicle and the direction pointed by the wheel rim; ψ is the heading angle, which refers to the angle between the vehicle body and the X-axis.

Model assumptions:

(1) The vehicle has no vertical motion, and only moves in the XY plane.

(2) The tire deflection angle on the left and right sides of the vehicle is the same, which can be simplified as a bicycle model.

(3) The vehicle is front-wheel drive, it can be assumed that the rear wheel slip angle δr=0.

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

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

$$\psi_{t+1} = \psi_t + \frac{v_t}{l_r} \sin(\beta) \times dt$$

$$v_{t+1} = v_t + a \times dt$$

where β can be calculated as follows:

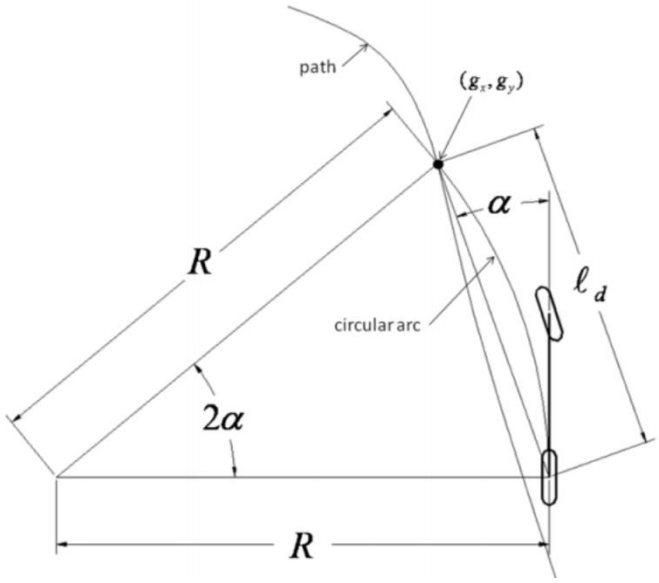
$$\beta = \tan^{-1}\left(\frac{l_r}{l_f + l_r} \tan(\delta_f)\right)$$

Python code:

```
class vehicle:
  def __init__(self,x=0.0,y=0.0,yaw=0.0,v=0.0):
    self.x = x
    self.y = y
    self.yaw = yaw
    self.v = v
    self.delta = 0
    self.dt = 0.1
  def update(self,a,steering):
    self.x = self.x + self.v*math.cos(self.yaw + self.delta)*dt
    self.y = self.y + self.v*math.sin(self.yaw + self.delta)*dt
    self.yaw = self.yaw + self.v/l_r*math.tan(self.delta)*dt
    self.v = self.v + a*dt
    self.delta = math.atan(l_r/(l_f+l_r)*math.tan(steering))
```

1.3 Vehicle tracking model

Starting from the bicycle model, the pure tracking algorithm takes the rear axle of the vehicle as the tangent point, and the longitudinal body of the vehicle as the tangent line. By controlling the front wheel angle, the vehicle can drive along an arc passing through the target waypoint, as shown in the following figure:



It is necessary to control the center point of the rear axle of the vehicle to pass through the point to be tracked. According to the law of sine, we get:

$$\frac{l_d}{\sin(2\alpha)} = \frac{R}{\sin(\frac{\pi}{2} - \alpha)}$$

$$\frac{l_d}{2\sin\alpha\cos\alpha} = \frac{R}{\cos\alpha}$$

$$\frac{l_d}{\sin\alpha} = 2R$$

Combined with the above derivation, the control variable expression of pure tracking control algorithm is obtained:

$$\delta(t) = tan^{-1}(\frac{2Lsin(\alpha(t))}{l_d})$$

The time is taken into consideration here. In the case of knowing the angle α(t) between the vehicle body and the target waypoint at time t and the forward-looking distance ld from the target waypoint, since the vehicle wheelbase L is fixed, we can use the above formula to estimate The front wheel rotation angle δ that should be made, in order to better understand the principle of pure tracking controller, we define a new quantity: el The error between the current attitude of the vehicle and the target waypoint in the lateral direction, from which the sine of the angle can be obtained:

$$sin(\alpha) = \frac{e_l}{l_d}$$

The arc's radian can then be rewritten as:

$$\kappa = \frac{2}{l_d^2} e_l$$

Considering that it is essentially a horizontal CTE, it can be seen from the above formula that the pure tracking controller is actually a P controller with a lateral angle. This P controller is greatly affected by the foresight distance. How to adjust the foresight distance becomes a pure tracking algorithm The key, generally speaking, is considered that the forward sight distance is a function of the vehicle speed, and different forward sight distances need to be selected at different vehicle speeds.One of the most common ways to adjust the front sight distance is to express the front sight distance as a linear function of the longitudinal speed of the vehicle, that is, l=kv, then the front wheel angle formula becomes:

$$\delta(t) = tan^{-1}(\frac{2Lsin(\alpha(t))}{kv_x(t)})$$

Then the adjustment of the pure tracking controller becomes the adjustment coefficient k. Generally speaking, the maximum and minimum foresight distances are used to constrain the foresight distance. The apparent distance will make the tracking more accurate (of course it will also bring about the oscillation of the control), here we use Python to implement a simple pure tracking controller.

Python implements pure trajectory tracking algorithm

In this practice, the steering angle is controlled by tracking control, and a simple P controller is used to control the speed. First, we define the parameter values as follows:

```
import numpy as np
import math
import matplotlib.pyplot as plt
k = 0.1  # foresight distance factor
Lfc = 2.0  # foresight distance
Kp = 1.0  # speed P controller coefficient
dt = 0.1  # time interval, unit: s
L = 2.9  # vehicle wheelbase, unit: m
```

Here we set the minimum forward distance to 2, the coefficient k of the forward distance with respect to the vehicle speed is set to 0.1, the proportional coefficient Kp of the speed P controller is set to 1.0, the time interval is 0.1 seconds, and the wheelbase of the vehicle is set to 2.9 Meter.
Define the vehicle state class. In the simple bicycle model, we only consider the current position (x, y) of the vehicle, the yaw angle yaw of the vehicle and the speed v of the vehicle. In order to simulate in software, we define the state update function of the vehicle To simulate the status update of a real vehicle:

```
class VehicleState:
    def __init__(self, x=0.0, y=0.0, yaw=0.0, v=0.0):
        self.x = x
        self.y = y
        self.yaw = yaw
        self.v = v
def update(state, a, delta):
    state.x = state.x + state.v * math.cos(state.yaw) * dt
    state.y = state.y + state.v * math.sin(state.yaw) * dt
    state.yaw = state.yaw + state.v / L * math.tan(delta) * dt
    state.v = state.v + a * dt
    return state
```

In this practice, we use a simple P controller for longitudinal control and a pure tracking controller for lateral control (ie corner control). These two controllers are defined as follows:

```
def PControl(target, current):
    a = Kp * (target - current)
    return a
def pure_pursuit_control(state, cx, cy, pind):
    ind = calc_target_index(state, cx, cy)
    if pind >= ind:
        ind = pind
    if ind < len(cx):
        tx = cx[ind]
        ty = cy[ind]
    else:
        tx = cx[-1]
        ty = cy[-1]
        ind = len(cx) - 1
    alpha = math.atan2(ty - state.y, tx - state.x) - state.yaw
    if state.v < 0:  # back
        alpha = math.pi - alpha
    Lf = k * state.v + Lfc
    delta = math.atan2(2.0 * L * math.sin(alpha) / Lf, 1.0)
    return delta, ind
```

Define a function to search for the nearest waypoint:

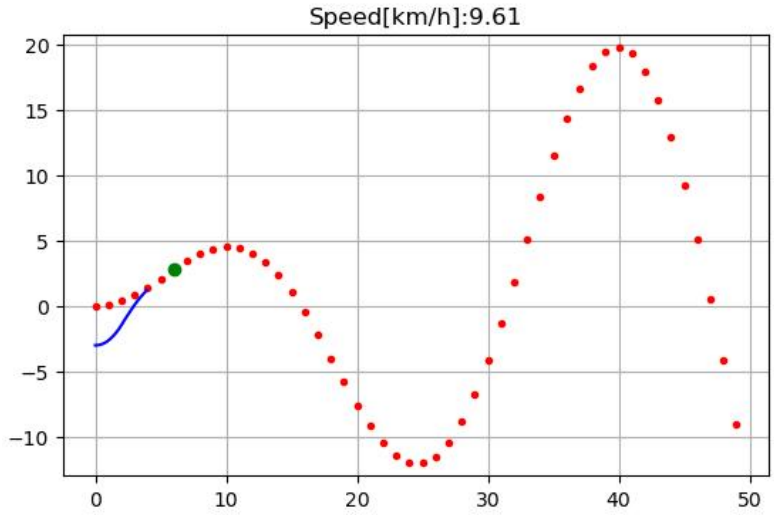
```
def calc_target_index(state, cx, cy):
    # search for the nearest waypoint
    dx = [state.x - icx for icx in cx]
    dy = [state.y - icy for icy in cy]
    d = [abs(math.sqrt(idx ** 2 + idy ** 2)) for (idx, idy) in zip(dx, dy)]
    ind = d.index(min(d))
    L = 0.0
    Lf = k * state.v + Lfc
    while Lf > L and (ind + 1) < len(cx):
        dx = cx[ind + 1] - cx[ind]
        dy = cx[ind + 1] - cx[ind]
        L += math.sqrt(dx ** 2 + dy ** 2)
        ind += 1
    return ind
```

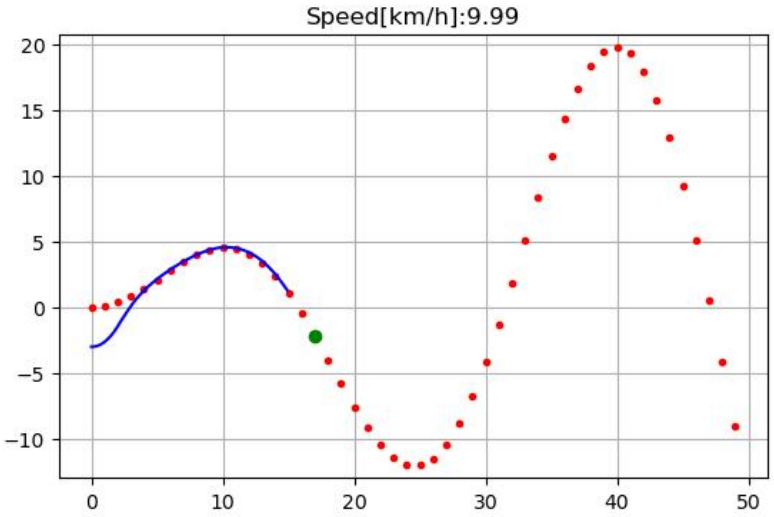
Main function:

```
def main():
    #  set target waypoint
    cx = np.arange(0, 50, 1)
    cy = [math.sin(ix / 5.0) * ix / 2.0 for ix in cx]
    target_speed = 10.0 / 3.6  # [m/s]
    T = 100.0  # maximum simulation time
    # set the accident status of the vehicle
    state = VehicleState(x=-0.0, y=-3.0, yaw=0.0, v=0.0)
    lastIndex = len(cx) - 1
    time = 0.0
    x = [state.x]
    y = [state.y]
    yaw = [state.yaw]
    v = [state.v]
    t = [0.0]
```

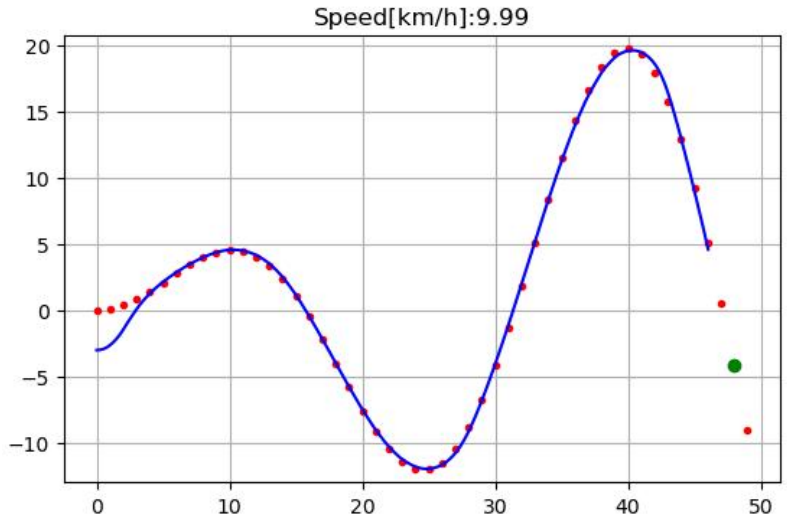
```
    target_ind = calc_target_index(state, cx, cy)
    while T >= time and lastIndex > target_ind:
        ai = PControl(target_speed, state.v)
        di, target_ind = pure_pursuit_control(state, cx, cy, target_ind)
        state = update(state, ai, di)
        time = time + dt
        x.append(state.x)
        y.append(state.y)
        yaw.append(state.yaw)
        v.append(state.v)
        t.append(time)
        plt.cla()
        plt.plot(cx, cy, ".r", label="course")
        plt.plot(x, y, "-b", label="trajectory")
        plt.plot(cx[target_ind], cy[target_ind], "go", label="target")
        plt.axis("equal")
        plt.grid(True)
        plt.title("Speed[km/h]:" + str(state.v * 3.6)[:4])
        plt.pause(0.001)
if __name__ == '__main__':
    main()
```

Operation effect:





在此处键入公式。



In the running diagram, the red dots represent the planned waypoints, the blue lines represent the actual trajectory of the vehicle, and the green dots in front represent the current forward looking distance.

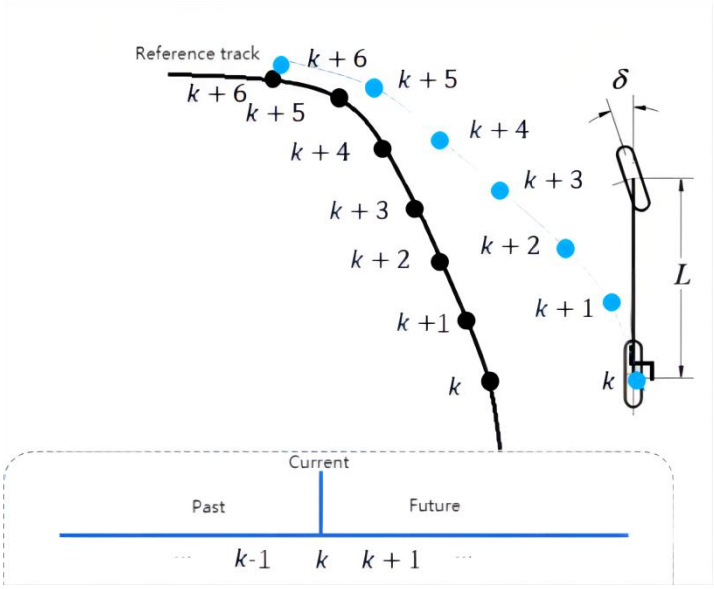
1.4 Summary

Geometric path tracers represented by pure tracking controllers are easy to understand and implement. Here I have achieved basic path following performance using geometric methods, but bottlenecks are encountered when there are significant speed changes, pure tracking methods use look-ahead distance to consider path information, this method is almost unaffected by path shape at low speeds influences. However, the method of choosing the best look-ahead distance is not clear. Expressing foresight distance as a function of velocity is a common approach, however, foresight distance may also be a function of path curvature and may even be related to CTE other than longitudinal velocity. Therefore, additional attention should be paid to the adjustment of the forward viewing distance of the pure tracking controller. A very short forward viewing distance will cause instability or even oscillation of the vehicle control. In order to ensure the stability of the vehicle, setting a longer forward viewing distance will cause the vehicle to be at a large corner. understeer problem.

2. Implementation of MPC for Vehicle Trajectory Tracking

2.1 MPC control principle

The core idea of Model Predictive Control (MPC) is to use a three-dimensional space model plus time to form a four-dimensional space-time model, and then solve the optimal controller based on this space-time model. The MPC controller is based on a space-time model over a period of time, so the obtained control output is also the control sequence of the system at finite time steps in the future. Because the theoretically constructed model has errors with the real model of the system; thus, the control output in the far future is of very low value to the system control, and the MPC only executes the first control output in the output sequence.



Assuming that the forward prediction step size is T, then the spatiotemporal model at step T is much larger than the original spatial model. In each control cycle, MPC needs to re-use the model calculation of the next T steps to obtain the currently executed control instructions. MPC utilizes a much larger model (higher computational cost) than the original spatial model just to get the optimal controller for the current step.

2.2 Linear model

When the model is in linear form (or a non-linear model is linearized using the same method), the solution of the MPC controller can be transformed into a quadratic programming problem,

Assume a linear model of the form:

$$x_{k+1} = Ax_k + Bu_k + C \tag{1}$$

Assuming that the control input of the next T steps is known, which is $u_k, u_{k+1}, u_{k+2}, \ldots, u_{k+T}$, according to the above model and input, we can calculate the state of the next T steps:

$$x_{k+1} = Ax_k + Bu_k + C$$
$$x_{k+2} = Ax_{k+1} + Bu_{k+1} + C = A(Ax_k + Bu_k + C) + Bu_{k+1} + C = A^2x_{k+1} + ABu_k + Bu_{k+1} + AC + C$$
$$x_{k+3} = A^3x_k + A^2Bu_k + ABu_{k+1} + Bu_{k+2} + A^2C + AC + C$$
$$\ldots$$
$$x_{k+T} = A^Tx_k + A^{T-1}Bu_k + A^{T-2}Bu_{k+1} + \ldots + A^{T-i}Bu_{k+i-1} + \ldots + Bu_{k+T-1} + A^{T-1}C + A^{T-2}C + \ldots + C$$

Plan the above T into a matrix-vector form:

$$\mathcal{X} = \mathcal{A}x_k + \mathcal{B}\mathbf{u} + \mathcal{C} \tag{2}$$

In the formula,

$$\mathcal{X} = \begin{bmatrix} x_{k+1} & x_{k+2} & x_{k+3} & \ldots & x_{k+T} \end{bmatrix}^T,$$
$$\mathbf{u} = \begin{bmatrix} u_k & u_{k+1} & u_{k+2} & \ldots & u_{k+T-1} \end{bmatrix}^T,$$
$$\mathcal{A} = \begin{bmatrix} A & A^2 & A^3 & \ldots & A^T \end{bmatrix}^T,$$
$$\mathcal{B} = \begin{bmatrix} B & 0 & 0 & \ldots & 0 \\ AB & B & 0 & \ldots & 0 \\ A^2B & AB & B & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ A^{t-1}B & A^{T-2}B & A^{T-3}B & \ldots & B \end{bmatrix},$$
$$\mathcal{C} = \begin{bmatrix} C \\ AC + C \\ A^2C + AC + C \\ \ldots \\ A^{k+T-1}C + \ldots + C \end{bmatrix}$$

Assuming the reference trajectory :

$$\overline{\mathcal{X}} = \begin{bmatrix} \overline{x}_{k+1} & \overline{x}_{k+2} & \overline{x}_{k+3} & \ldots & \overline{x}_{k+T} \end{bmatrix}^T$$

A simple objective cost function of MPC is as follows:

$$\begin{aligned} \min \mathcal{J} &= \mathcal{E}^T Q \mathcal{E} + \mathbf{u}^T R \mathbf{u} \\ \text{s.t. } & u_{min} \le \mathbf{u} \le u_{max} \end{aligned} \tag{3}$$

In the formula,

$$\mathcal{E} = \mathcal{X} - \overline{\mathcal{X}} = \begin{bmatrix} x_{k+1} - \overline{x}_{k+1} & x_{k+2} - \overline{x}_{k+2} & \ldots & x_{k+T} - \overline{x}_{k+T} \end{bmatrix}^T$$

Substituting equation (2) into equation (3), the optimization variable is only multiplied by **u**. The above optimization problem can be solved by quadratic programming method, and the optimal control sequence **u** that satisfies the objective cost function is obtained as :

$$\begin{Bmatrix} u_k & u_{k+1} & u_{k+2} & \ldots & u_{k+T-1} \end{Bmatrix}$$

When converted into Equation (3), it can be solved by using a convex optimization library, such as

python's [cvxopt], OSQP: An Operator Splitting Solver for Quadratic Programs (https://github.com/cvxopt/cvxopt).

2.3 Nonlinear model

If the model is nonlinear and we don't want to linearize it (the linearization process loses model accuracy). Here is an example of the geometric kinematics model of an unmanned vehicle:

$$\begin{aligned}\dot{x} &= v\cos(\theta)\\ \dot{y} &= v\sin(\theta)\\ \dot{\theta} &= v\frac{\tan(\delta)}{L}\\ \dot{v} &= a\end{aligned} \tag{1}$$

First, we need to discretize the above continuous differential model into a difference model (the difference interval is dt):

$$\begin{aligned}x_{k+1} &= x_k + v_k\cos(\theta_k)d_t\\ y_{k+1} &= y_k + v_k\sin(\theta_k)d_t\\ \theta_{k+1} &= \theta_k + v_k\frac{\tan(\delta_k)}{L}d_t\\ v_{k+1} &= v_k + a_k d_t\\ cte_{k+1} &= cte_k + v_k\sin(\theta_k)d_t\\ epsi_{k+1} &= epsi_k + v_k\frac{\tan(\delta_k)}{L}d_t\end{aligned} \tag{2}$$

The trajectory that the unmanned vehicle needs to track is formed by the discrete points {($\bar{x}_1$, $\bar{y}_1$), ($\bar{x}_2$, $\bar{y}_2$), ..., ($\bar{x}_M$, $\bar{y}_M$)} through cubic curve fitting, which can be represented by the x-axis is a function of the independent variable: y =f(x) = $c_0x^3 + c_1x^2 + c_2x + c_3$. It is worth noting that the discrete points representing the trajectory need to be transformed into the vehicle body coordinate system.

Therefore, at each prediction step, we can calculate the lateral tracking error cte_k and heading deviation $epsi_k$ according to the x_k and y_k values of the unmanned vehicle, The specific calculation formula is as follows:

$$\begin{aligned}cte_k &= f(x_k) - y_k\\ epsi_k &= \arctan(f'(x_k)) - \theta\end{aligned} \tag{3}$$

For an MPC controller with a prediction step size of N to solve the problem, the following optimization objective function can be designed:

$$\begin{aligned}\min \quad \mathcal{J} = &\sum_{k=1}^{N}(\omega_{cte}||cte_t||^2 + \omega_{epsi}||epsi_k||^2 + \omega_v||v_k - v_{ref}||^2)\\ &+ \sum_{k=1}^{N-1}(\omega_\delta||\delta_k||^2 + \omega_a||a_k||^2)\\ &+ \sum_{k=1}^{N-2}(\omega_{rate_\delta}||\delta_{k+1} - \delta_k||^2 + \omega_{rate_a}||a_{k+1} - a_k||^2)\end{aligned} \tag{4}$$

Satisfy dynamic model constraints:

$$\begin{aligned}\text{s.t.} \quad x_{k+1} &= x_k + v_k\cos(\theta_k)d_t, k = 1, 2, ..., N-1\\ y_{k+1} &= y_k + v_k\sin(\theta_k)d_t, k = 1, 2, ..., N-1\\ \theta_{k+1} &= \theta_k + v_k\frac{\tan(\delta_k)}{L}d_t, k = 1, 2, ..., N-1\\ v_{k+1} &= v_k + a_k d_t, k = 1, 2, ..., N-1\\ cte_{k+1} &= f(x_k) - y_k + v_k\sin(\theta_k)d_t\\ epsi_{k+1} &= \arctan(f'(x_k)) - \theta + v_k\frac{\tan(\delta_k)}{L}d_t\end{aligned} \tag{5}$$

Actuator constraints:

$$\begin{aligned}\delta &\in [\delta_{min}, \delta_{max}]\\ a &\in [a_{min}, a_{max}]\end{aligned} \tag{6}$$

Equations (4), (5) and (6) constitute the complete control problem of trajectory tracking of unmanned vehicles.

2.4 Solving the MPC problem of unmanned vehicle trajectory tracking

Implementation based on Python+Pyomo

Code:

```
import numpy as np
from pyomo.environ import *
from pyomo.dae import *

N = 9 # forward predict steps
ns = 6  # state numbers / here: 1: x, 2: y, 3: psi, 4: v, 5: cte, 6: epsi
na = 2  # actuator numbers /here: 1: steering angle, 2: throttle
```

```
class MPC(object):
    def __init__(self):
        m = ConcreteModel()
        m.sk = RangeSet(0, N-1)
        m.uk = RangeSet(0, N-2)
        m.uk1 = RangeSet(0, N-3)
        # Parameters
        m.wg       = Param(RangeSet(0, 3), initialize={0:1., 1:10., 2:100., 3:130000}, mutable=True)
        m.dt       = Param(initialize=0.1, mutable=True)
        m.Lf       = Param(initialize=2.67, mutable=True)
        m.ref_v    = Param(initialize=75., mutable=True)
        m.ref_cte  = Param(initialize=0.0, mutable=True)
        m.ref_epsi = Param(initialize=0.0, mutable=True)
        m.s0       = Param(RangeSet(0, ns-1), initialize={0:0., 1:0., 2:0., 3:0., 4:0., 5:0.}, mutable=True)
        m.coeffs   = Param(RangeSet(0, 3),
                           initialize={0:-0.000458316, 1:0.00734257, 2:0.0538795, 3:0.080728}, mutable=True)
```

```
        # Variables
        m.s      = Var(RangeSet(0, ns-1), m.sk)
        m.f      = Var(m.sk)
        m.psides = Var(m.sk)
        m.ua     = Var(m.uk, bounds=(-1.0, 1.0))
        m.ud     = Var(m.uk, bounds=(-0.436332, 0.436332))
```

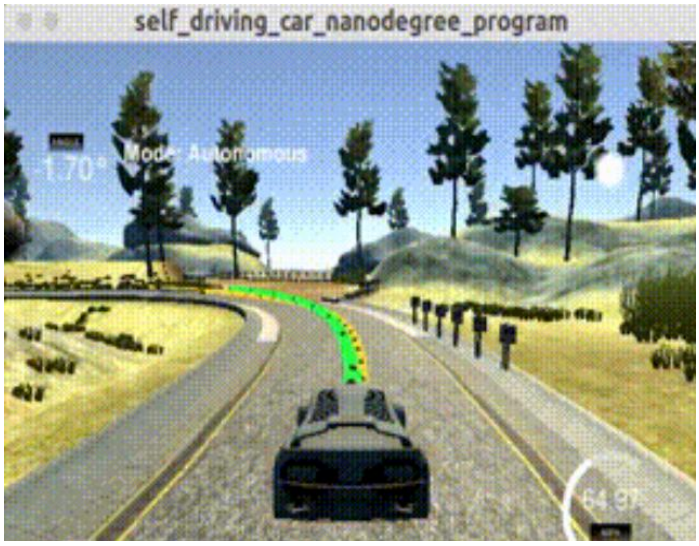
```
        # 0: x, 1: y, 2: psi, 3: v, 4: cte, 5: epsi
        # Constrainsts
        m.s0_update      = Constraint(RangeSet(0, ns-1), rule = lambda m, i: m.s[i,0] == m.s0[i])
        m.x_update       = Constraint(m.sk, rule=lambda m, k:
                                      m.s[0,k+1]==m.s[0,k]+m.s[3,k]*cos(m.s[2,k])*m.dt
                                      if k<N-1 else Constraint.Skip)
        m.y_update       = Constraint(m.sk, rule=lambda m, k:
                                      m.s[1,k+1]==m.s[1,k]+m.s[3,k]*sin(m.s[2,k])*m.dt
                                      if k<N-1 else Constraint.Skip)
        m.psi_update     = Constraint(m.sk, rule=lambda m, k:
                                       m.s[2,k+1]==m.s[2,k]-m.s[3,k]*m.ud[k]/m.Lf*m.dt
                                       if k<N-1 else Constraint.Skip)
        m.v_update       = Constraint(m.sk, rule=lambda m, k:
                                       m.s[3,k+1]==m.s[3,k]+m.ua[k]*m.dt
                                       if k<N-1 else Constraint.Skip)
        m.f_update      = Constraint(m.sk, rule=lambda m, k:
                                       m.f[k]==m.coeffs[0]*m.s[0,k]**3+m.coeffs[1]*m.s[0,k]**2+
                                       m.coeffs[2]*m.s[0,k]+m.coeffs[3])
        m.psides_update = Constraint(m.sk, rule=lambda m, k:
                                           m.psides[k]==atan(3*m.coeffs[0]*m.s[0,k]**2
                                                             +2*m.coeffs[1]*m.s[0,k]+m.coeffs[2]))
        m.cte_update     = Constraint(m.sk, rule=lambda m, k:
                                        m.s[4,k+1]==(m.f[k]-m.s[1,k]+m.s[3,k]*sin(m.s[5,k])*m.dt)
                                       if k<N-1 else Constraint.Skip)
        m.epsi_update    = Constraint(m.sk, rule=lambda m, k:
                                  m.s[5, k+1]==m.s[2,k]-m.psides[k]-m.s[3,k]*m.ud[k]/m.Lf*m.dt
                                        if k<N-1 else Constraint.Skip)
```

```
        # Objective function
        m.cteobj  = m.wg[2]*sum((m.s[4,k]-m.ref_cte)**2 for k in m.sk)
        m.epsiobj = m.wg[2]*sum((m.s[5,k]-m.ref_epsi)**2 for k in m.sk)
        m.vobj    = m.wg[0]*sum((m.s[3,k]-m.ref_v)**2 for k in m.sk)
        m.udobj   = m.wg[1]*sum(m.ud[k]**2 for k in m.uk)
        m.uaobj   = m.wg[1]*sum(m.ua[k]**2 for k in m.uk)
        m.sudobj  = m.wg[3]*sum((m.ud[k+1]-m.ud[k])**2 for k in m.uk1)
        m.suaobj  = m.wg[2]*sum((m.ua[k+1]-m.ua[k])**2 for k in m.uk1)
        m.obj = Objective(expr = m.cteobj+m.epsiobj+m.vobj+m.udobj+m.uaobj+m.sudobj+m.suaobj, sense=minimize)

        self.iN = m#.create_instance()
```

```
    def Solve(self, state, coeffs):
        self.iN.s0.reconstruct({0:state[0], 1: state[1], 2:state[2], 3:state[3], 4:state[4], 5:state[5]})
        self.iN.coeffs.reconstruct({0:coeffs[0], 1:coeffs[1], 2:coeffs[2], 3:coeffs[3]})
        self.iN.f_update.reconstruct()
        self.iN.s0_update.reconstruct()
        self.iN.psides_update.reconstruct()
        SolverFactory('ipopt').solve(self.iN)
        x_pred_vals = [self.iN.s[0,k]() for k in self.iN.sk]
        y_pred_vals = [self.iN.s[1,k]() for k in self.iN.sk]
        steering_angle = self.iN.ud[0]()
        throttle = self.iN.ua[0]()
        return x_pred_vals, y_pred_vals, steering_angle, throttle
```

Operation effect:



The yellow curve in the above figure is the trajectory to be tracked by the unmanned vehicle, and the green curve is the optimal tracking trajectory calculated by MPC. Each time the MPC is solved, the control commands of N steps are obtained, but only the results of the current step are used, and the subsequent steps are thrown away. Therefore, the update cycle of the MPC is consistent with the update cycle of the actuator.

2.5 Summary

For unmanned vehicle trajectory tracking MPC control, in order to minimize the loss of the model, we use the unmanned vehicle geometric kinematics model to construct the MPC problem model. Implementation based on Pyomo. We found that the unmanned vehicle can track the trajectory well at a speed of nearly 70 MPH (about 113 km/h, about 30 m/s), which is enough to demonstrate the stability and effectiveness of the MPC trajectory tracking control.