第 3 章 参考线的离散化与匹配点的选择

在 Frenet 坐标系中，参考线 $r(t)$可以是任意随时间变化的曲线，然而，在 Lattice 规划算法中，参考线 $r(t)$一般是指结构化道路的车道中心线，由高精地图提供。本章将重点从理论与程序实现的角度介绍参考线的描述与计算、离散化与匹配点的计算等问题。

3.1 参考线的描述与计算

参考线 $s(t)$是 t 时刻的累计弧长，定义如下：

$$s(t)=\int_0^t \| r'(x) \|_2 \mathrm{d}x \tag{3-1}$$

式中，$r'(t)=\dfrac{\partial r}{\partial t}$，$\| \cdot \|_2$ 表示 2-范数。$r(t)$为欧氏空间随时间不断变化的非退化曲线，如图 3-1 所示。此处的非退化指的是 $r(t)$的曲率不为 0，即不是一条直线。

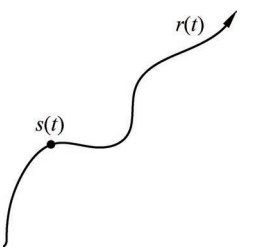


图 3-1 非退化曲线与参考线

在理论上，虽然 $r(t)$是非退化曲线，但是在工程实践中，参考线一般为结构化道路的道路中心线。因此，这就要求参考线无论在曲线还是直线的条件下都可以计算累计弧长。该问题的解决办法，也就是后续 3.2 节提到的参考线离散化问题。

由前述 $s(t)\equiv s \Rightarrow t=t(s)$，$s(t)\equiv s$，可知 s 不仅是时间 t 的函数，t 也是累计弧长 s 的函数，因此，参考线 $r(t)$也可以记为 $r(s)$。

在实际的工程实践中，参考线是离散化的，$s(t)$的计算会与式(3-1)略有差异。计算过程可以描述如下：

(1) 设 t_0 时刻的累计弧长 $s(t_0)=0$。

(2) 设$(x(t_0),y(t_0))$，$(x(t_1),y(t_1))$，…，$(x(t_i),y(t_i))$，…，$(x(t_{n-1}),y(t_{n-1}))$为参考线上的 n 个离散点。

(3) $s(t)$的计算如下所示。

$$s(t_k)=\sqrt{(x(t_1)-x(t_0))^2+(y(t_1)-y(t_0))^2}+\cdots+$$

$$\sqrt{(x(t_i)-x(t_{i-1}))^2+(y(t_i)-y(t_{i-1}))^2}+\cdots+$$
$$\sqrt{(x(t_k)-x(t_{k-1}))^2+(y(t_k)-y(t_{k-1}))^2}$$
$$=s(t_{k-1})+\sqrt{(x(t_k)-x(t_{k-1}))^2+(y(t_k)-y(t_{k-1}))^2} \tag{3-2}$$

3.2 参考线的离散化与 $s(t)$的计算

3.2.1 离散化与计算过程

在 Lattice 算法中,参考线的离散化需要借助结构体或类 PathPoint,以及类 reference_point 和 MapPathPoint 存储高精地图中的路点。为了便于说明问题,书中对 Path、reference_point 和 MapPathPoint 等类进行了简化处理。具体的代码可以参看 Apollo 6.0。

Lattice 算法中对于参考线的离散化计算过程可以归纳为以下步骤:

(1) 构建存储离散点的“容器”。

(2) 将参考线中各个路点的属性值(包括笛卡儿坐标(x,y)、航向角、曲率、曲率变化率)存入“容器”。

(3) 计算相邻路点之间的欧氏距离。

(4) 计算累计弧长。

3.2.2 实例分析

对参考线进行离散化与累计弧长计算是 Lattice 算法的前提与基础。为了让读者对该过程的理论分析过程与实现有更好的理解与认知,书中对该过程从 Python 和 C++的角度进行编程实现与可视化显示。

1. 基于 Python 的参考线计算代码实现

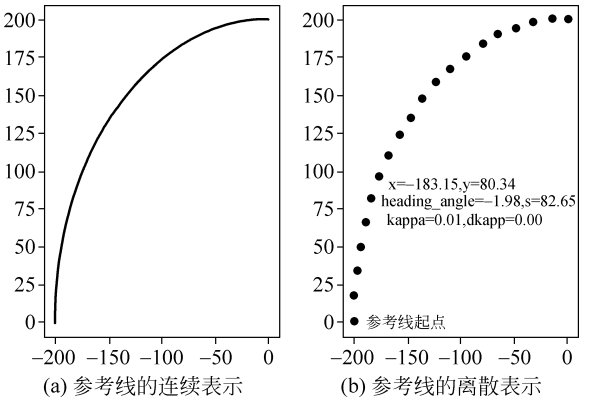
代码如下:

```
//第 3 章/DiscretizedReference.py
import numpy as np
import math
import matplotlib.pyplot as plt

#生成参考线,此处的参考线为半径为 R 的一段弧长
def generate_referenceline(start_angle, end_angle, radius):
    #弧长所对应的圆心角的范围为[start_angle,end_angle]
    theta = np.linspace(start_angle, end_angle, 20)

    #该段弧长上点的坐标(x,y),其中 x = R * cos(theta),y = R * sin(theta)
    x = radius * np.cos(theta)
```

```
    y = radius * np.sin(theta)

    #计算每点处的 dx / dtheta 和 dy / dtheta
    dx_dtheta = -radius * np.sin(theta)
    dy_dtheta = radius * np.cos(theta)

    #计算每点的航向角,即斜率为 dy / dx
    heading_angle = np.arctan2(dy_dtheta, dx_dtheta)

    #由于是一段圆弧,所以曲率为 1 / R, 曲率变化率为 0
    kappa = np.ones(20) * 1 / radius
    dkappa = np.zeros(20)

    #用 zip 函数将每个点的坐标、航向角、曲率和曲率变化率封装
    reference_line = zip(x, y, heading_angle, kappa, dkappa)

    return reference_line
#此函数主要用于体现参考线累计弧长及其他参数的计算过程
def to_discretized_referenceline(reference_line):

    s = 0.0                          #参考线的累计弧长
    s_set = []                       #list 容器,存放每个点的 s 值
    x_set = []                       #list 容器,存放每个点的 x 坐标
    y_set = []                       #list 容器,存放每个点的 y 坐标
    heading_angle_set = []           #list 容器,存放每个点的航向角
    kappa_set = []                   #list 容器,存放每个点的曲率
    dkappa_set = []                  #list 容器,存放每个点的曲率变化率

    #遍历参考线
    for x, y, heading_angle, kappa, dkappa in reference_line:
        x_set.append(x)
        y_set.append(y)
        heading_angle_set.append(heading_angle)
        kappa_set.append(kappa)
        dkappa_set.append(dkappa)

        if len(s_set) != 0:
            #计算相邻两点的欧氏距离,并累加弧长
            dx = x_set[-2] - x_set[-1]
            dy = y_set[-2] - y_set[-1]
            s += math.sqrt(dx * dx + dy * dy)

        s_set.append(s)

    path_points = zip(x_set, y_set, heading_angle_set, kappa_set, dkappa_set, s_set)
    return path_points

if __name__ == '__main__':
```

```
    #参考线的起始弧度为 PI,终止弧度为 0.5 * PI,半径为 200
    reference_line = generate_referenceline(start_angle = math.pi, end_angle = 0.5 *
math.pi, radius = 200.0)
    reference_line_frenet = generate_referenceline(start_angle = math.pi, end_angle = 0.5 *
math.pi, radius = 200.0)
    x_set, y_set,_,_,_ = zip( * reference_line)#利用 zip( * )解包数据
    PathPoints = to_discretized_referenceline(reference_line_frenet)
    x_discretized,y_discretized,heading_angle_set,kappa_set,dkappa_set,s_set = zip( *
PathPoints)#同上

    plt.subplot(121)
    plt.plot(x_set, y_set, color = 'black')
    plt.title('(a)', y = -0.2)

    plt.subplot(122)
    #下述两句主要是为了显示"汉字"
    plt.rcParams['font.sans-serif'] = ['SimHei']
    plt.rcParams['axes.unicode_minus'] = False
    plt.scatter(x_discretized, y_discretized, color = 'black', marker = "o")
    plt.text(x_discretized[0] + 10, y_discretized[0] - 5,'参考线起点')
    plt.text(x_discretized[5] + 15, y_discretized[5] - 15, 'x = ' +
            str(round(x_discretized[5],2)) + ', y = ' +
            str(round(y_discretized[5],2)) + '\n' + 'heading_angle = '
            + str(round(heading_angle_set[5],2)) + ',s = ' +
            str(round(s_set[5],2)) + '\n' + 'kappa = ' + '%.2f' %kappa_set[5]
            + ',dkapp = ' + '%.2f' %dkappa_set[5])
    plt.title('(b)', y = -0.2)
    plt.show() #显示效果如图 3-2 所示
```

图 3-2(a)所示的是以连续的形式显示参考线,而在实际的工程应用中需要采取离散化的形式进行计算,因此,在图 3-2(b)中,书中以散点的形式显示,并同时显示了某点的诸多属性,例如坐标、累计弧长、航向角、曲率和曲率变化率等,如图 3-2(b)所示。



(a) 参考线的连续表示　　(b) 参考线的离散表示

图 3-2　参考线离散化计算

2. 基于 C++的参考线计算代码实现

对于参考线中的各个离散点，在实现时，Apollo 中利用了 message 表示各个路点的信息，具体表示与实现代码如下：

```
message PathPoint {
  //三维坐标的类型
  optional double x = 1;
  optional double y = 2;
  optional double z = 3;

  //方向的表示是以 XOY 平面为基础的
  optional double theta = 4;
  //XOY 平面的曲率
  optional double kappa = 5;
  //从起点开始累计的线段长度之和，即 Frenet 坐标系下的坐标 s
  optional double s = 6;

  //此处表示曲率的导数
  optional double dkappa = 7;
  //表示对曲率的二次微分
  optional double ddkappa = 8;
  //表示路点所在车道的 ID
  optional string lane_id = 9;

  //表示对坐标点(x,y)的导数(dx/dt, dy/dt)
  optional double x_derivative = 10;
  optional double y_derivative = 11;
}
```

```
//第 3 章/Discretized.cc

std::vector<PathPoint> ToDiscretizedReferenceLine(
    const std::vector<ReferencePoint>& ref_points) {
  double s = 0.0;
  std::vector<PathPoint> path_points;
  for (const auto& ref_point : ref_points) {
    PathPoint path_point;
    path_point.set_x(ref_point.x());
    path_point.set_y(ref_point.y());
    path_point.set_theta(ref_point.heading());
    path_point.set_kappa(ref_point.kappa());
    path_point.set_dkappa(ref_point.dkappa());

    if (!path_points.empty()) {
      double dx = path_point.x() - path_points.back().x();
      double dy = path_point.y() - path_points.back().y();
      s += std::sqrt(dx * dx + dy * dy);
```

```
        }
        path_point.set_s(s);
        path_points.push_back(std::move(path_point));
      }
      return path_points;
```

3.3 匹配点的选择

3.3.1 匹配点选择的描述

$p(x(t),y(t))$表示车辆当前的轨迹点，如图 3-3 所示 $v_j(j=0,1,\cdots,i-1,i,i+1,\cdots,n)$表示参考线上的离散点。匹配点的选择就是在参考线的各个离散点中找出距离车辆当前轨迹点最近的点。选择过程描述如下：

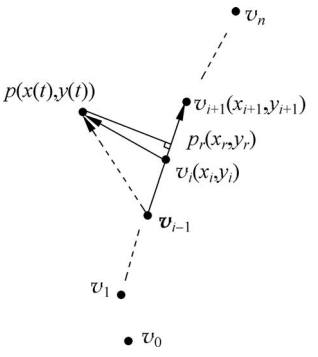


图 3-3 最佳匹配点的选择

(1) 分别计算车辆当前轨迹点 $p(x(t),y(t))$与参考线各个离散点的欧氏距离，公式如下：

$$l_j = \| pv_j \|_2 = \sqrt{(x(t)-x_j(t))^2+(y(t)-y_j(t))^2} \quad (j=0,1,\cdots,n) \tag{3-3}$$

(2) 求出参考线上距离车辆当前轨迹点的距离最小点 $v_{\text{index}} = \underset{j=0,1,\cdots,n}{\operatorname{argmin}} l_j$，并记录该点的索引(如图 3-3 中的点 $v_i(x_i,y_i)$，索引为 i，坐标为(x_i,y_i))。

(3) 根据距离最小值的点的索引，找出该索引的前一个点的索引和后一个点的索引(如图 3-3 所示，$v_i(x_i,y_i)$为距离最小值的点，前一个点则为 v_{i-1}，索引为 $i-1$，后一个点则为 v_{i+1}，索引为 $i+1$)。

(4) 根据距离最小点的前一个索引(如图 3-3 中的点$\boldsymbol{v}_{i-1}$)与车辆当前轨迹点(如图 3-3 中的点 $p(x(t),y(t))$)，计算向量$\boldsymbol{v}_{i-1}\boldsymbol{p}$ 在向量$\boldsymbol{v}_{i-1}\boldsymbol{v}_{i+1}$ 上的投影点 p_r(如图 3-3 所示)，公式如下：

$$\| \boldsymbol{v}_{i-1}\boldsymbol{p}_r \|_2 = \frac{\boldsymbol{v}_{i-1}\boldsymbol{p}\cdot\boldsymbol{v}_{i-1}\boldsymbol{v}_{i+1}}{\| \boldsymbol{v}_{i-1}\boldsymbol{v}_{i+1} \|_2} = \| \boldsymbol{v}_{i-1}\boldsymbol{p} \| \cos(pv_{i-1}v_{i+1}) \tag{3-4}$$

式中，·表示向量$\boldsymbol{v}_{i-1}\boldsymbol{p}$与向量$\boldsymbol{v}_{i-1}\boldsymbol{v}_{i+1}$的点积，两个向量之间的夹角表示为$pv_{i-1}v_{i+1}$。此时投影点$p_r$即为车辆当前轨迹点在参考线的最佳匹配点。

(5) 根据3.1节参考线上各点s计算过程的描述，可知最佳匹配点p_r的s值计算如下：

$$s_{p_r}=s_{v_{i-1}}+\|\boldsymbol{v}_{i-1}\boldsymbol{p}_r\|_2 \tag{3-5}$$

式中，$s_{v_{i-1}}$表示参考线上点v_{i-1}的s值。

(6) 根据参考线上点v_{i-1}、v_{i+1}的s值$s_{v_{i-1}}$和$s_{v_{i+1}}$，以及最佳匹配点p_r的s_{p_r}，利用线性插值计算匹配点p_r的笛卡儿坐标和曲率等属性值。

注意：之所以没有把图3-3中的点v_i作为车辆当前轨迹点的最佳匹配点，主要原因在于参考线各点是离散的，因此，如果把v_i作为最佳匹配点，则最短距离点的确定是带有误差的。例如图3-3中无论是线段v_ip的长度，还是线段$v_{i-1}p$的长度必然大于或等于它的投影线段p_rp的长度。这也是为什么需要找出最短距离点的前一个点和后一个点，我们再做投影的原因。同时，读者也应该注意，v_i的前一个点和后一个点可能不存在。例如，如果v_i是参考线的第1个离散点或者最后一个离散点，则在这种情况下，v_i也同时代表它的前一个点或后一个点。对于这种特殊情况，读者也需要注意。在具体处理时会有差异，在代码中读者也可以看到。

3.3.2　线性插值计算过程的描述

线性插值的基本原理比较简单，很多资料中都有详细的介绍。本书主要以Apollo 6.0中的代码实现为依据，对其基本原理或计算过程进行简要描述。

在笛卡儿坐标系XOY中，原点O与P_0、P、P_1共线，相互间的位置关系如图3-4所示，且P_0和P_1的笛卡儿坐标分别为(x_0,y_0)和(x_1,y_1)，P_0、P、P_1距离原点O的距离分别为$\|OP_0\|_2$、$\|OP\|_2$、$\|OP_1\|_2$，此处的距离可以类比Frenet坐标系下各点的s坐标。P_0'、P'和P_1'分别为它们在X轴上的投影。在上述条件下，可以用线性插值的方法来计算P点的坐标(x,y)。

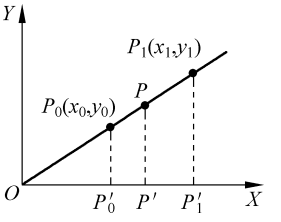


图3-4　线性插值示意图

以点P的坐标值x的求解为例。根据三角形相似可知：

$$\frac{x}{x_0}=\frac{\|OP\|_2}{\|OP_0\|_2} \tag{3-6}$$

$$\frac{x_1}{x_0}=\frac{\|OP_1\|_2}{\|OP_0\|_2} \tag{3-7}$$

由式(3-6)和式(3-7)可知：

$$\frac{x-x_0}{x_0}=\frac{\|OP\|_2-\|OP_0\|_2}{\|OP_0\|_2} \tag{3-8}$$

$$\frac{x_1 - x_0}{x_0} = \frac{\| OP_1 \|_2 - \| OP_0 \|_2}{\| OP_0 \|_2} \tag{3-9}$$

根据式(3-8)和式(3-9)可得

$$\frac{x - x_0}{x_1 - x_0} = \frac{\| OP \|_2 - \| OP_0 \|_2}{\| OP_1 \|_2 - \| OP_0 \|_2} \tag{3-10}$$

令$\frac{\| OP \|_2 - \| OP_0 \|_2}{\| OP_1 \|_2 - \| OP_0 \|_2} = w$,式(3-10)可得

$$x = (1 - w)x_0 + wx_1 \tag{3-11}$$

式(3-11)即为线性插值的计算公式。此计算方法可推广至参考线匹配点各属性值的计算。关于这一点,在实例分析中会结合 Apollo 6.0 中的代码与计算方法相互印证。

3.3.3 实例分析

为了验证匹配点的选择与线性插值的计算方法,本书对前面的理论部分进行了代码验证。所有代码读者可以通过扫描本书目录上方的二维码获得。运行环境为 Ubuntu 20.04,代码如下:

```
//第 3 章/main.cc
#include "../include/path_matcher.h"
#include <vector>
#include <cmath>

int main(int argc, char* argv[]){

  std::vector<PathPoint> reference_line; //构建存储离散参考线的“容器”

  float R(20.0f), x(10.0f), y(0.0f);
//设置离散参考线的半径与圆心,为了方便描述,假设离散参考线为圆的一部分
  double start_arc(M_PI), end_arc(0.5 * M_PI);
//设定参考线的起始圆心角
  double planning_x(-4.5), planning_y(14.14);
//设定参考线的圆心坐标
  PathPoint match_point;
  std::cout << "M_PI = " << M_PI << std::endl;

  for ( float angle = start_arc; angle >= end_arc; angle -= 0.15){
    //循环获取参考线上的各点
    PathPoint path_point;
    double x_point = x + R * std::cos(angle);
    double y_point = y + R * std::sin(angle);
    //计算圆弧上各点的坐标
    double theta_point = std::atan2(y_point, x_point);
    //计算圆弧上各点的切角,即航向角
    double kappa_point = 1 / R;
    //计算圆弧上各点的曲率,由于所有点都在圆弧上,所以曲率为 1/R
```

```
    double dkappa_point = 0;
    double ddkappa_point = 0;
    //由于曲率是常数,所以圆弧上各点的曲率的一阶和二阶导数为0
    double s = R * (M_PI - angle);
    //各点的s值根据弧长计算公式可求

    //把各点的笛卡儿坐标、航向角、曲率、曲率的一阶导和二阶导赋值给每个参考点
    path_point.set_x(x_point);
    path_point.set_y(y_point);
    path_point.set_theta(theta_point);
    path_point.set_kappa(kappa_point);
    path_point.set_dkappa(dkappa_point);
    path_point.set_ddkappa(ddkappa_point);
    path_point.set_s(s);

    //把各个参考点压入"容器"
    reference_line.push_back(path_point);

    }

    std::cout << "Current PathPoint is (-4.5,14.14)" << std::endl;
    match_point = apollo::common::math::PathMatcher::MatchToPath(reference_line, planning_x,
planning_y);
    std::cout << "The Match Point is (" << match_point.x() << "," << match_point.y() << ")" << ";
s = " << match_point.s() << std::endl;
    std::cout << "The Match Point theta = " << match_point.theta() << ";kappa = " << match_
point.kappa() << std::endl;
    std::cout << "The Match Point dkappa = " << match_point.dkappa() << ";ddkappa = " << match_
point.dkappa() << std::endl;


  return 0;
}
```

代码运行后,结果显示如下:

```
M_PI = 3.14159
Current PathPoint is (-4.5,14.14)
The Match Point is (-4.16313,13.8085);s = 15.4511
The Match Point theta = 1.88145;kappa = 0.05
The Match Point dkappa = 0;ddkappa = 0
```

可以看到,代码运行后获得了车辆当前轨迹点的匹配点的笛卡儿坐标、航向角、曲率等值,其中,匹配点选择的代码放在了 path_match.cc 文件中,代码如下:

```
//第3章/path_match.cc

#include "../include/path_matcher.h"
```

```
#include <algorithm>
#include <cmath>
#include <vector>

#include "glog/logging.h"
#include "../include/modules/math/linear_interpolation.h"

PathPoint PathMatcher::MatchToPath(const std::vector<PathPoint>& reference_line,
                                   const double x, const double y) {

  CHECK_GT(reference_line.size(), 0U);
  //判断参考点“容器”的大小是否为空,如果为空,则报错
  auto func_distance_square = [](const PathPoint& point, const double x,
                                 const double y) {
    double dx = point.x() - x;
    double dy = point.y() - y;
    return dx * dx + dy * dy;
  };
  //计算参考线“容器”内各点与当前车辆轨迹点(x,y)的欧氏距离,即式(3-2)所示。函数形
  //参 const double x 和 const double y。此处是通过 lambda 函数实现的

  double distance_min = func_distance_square(reference_line.front(), x, y);
  std::size_t index_min = 0;
  //将参考点“容器”内的第 1 个点与车辆当前轨迹点的距离作为最小距离,并将它的索引号作为
  //最小距离的索引号。这会在后续比较过程中进行更新

  for (std::size_t i = 1; i < reference_line.size(); ++i) {
    double distance_temp = func_distance_square(reference_line[i], x, y);
    if (distance_temp < distance_min) {
      distance_min = distance_temp;
      index_min = i;
    }
  }
  //遍历参考线上的各点,更新最小距离及其索引,如匹配点选择的步骤(2)

  std::size_t index_start = (index_min == 0) ? index_min : index_min - 1;
  std::size_t index_end =
      (index_min + 1 == reference_line.size()) ? index_min : index_min + 1;
  //检查最小距离点是否为特殊点,例如是否为第 1 个点或者为最后一个点,具体解释见 3.1.1 节的
  //注意中的描述

  if (index_start == index_end) {
    return reference_line[index_start];
  }
  //这是一种特殊情况,表明参考点“容器”内只有一个点

  return FindProjectionPoint(reference_line[index_start],
                             reference_line[index_end], x, y);
PathPoint PathMatcher::FindProjectionPoint(const PathPoint& p0,
```

```
                                                const PathPoint& p1, const double x,
                                                const double y) {
  double v0x = x - p0.x();
  double v0y = y - p0.y();
  //p0 点代表最小距离点的前一个点(如图 3-3 中的 v_{i-1}),此处计算车辆当前轨迹点与该点的
  //向量坐标

  double v1x = p1.x() - p0.x();
  double v1y = p1.y() - p0.y();
  //p1 点代表最小距离点的后一个点(如图 3-3 中的 v_{i+1}),此处计算车辆当前轨迹点与该点的
  //向量坐标

  double v1_norm = std::sqrt(v1x * v1x + v1y * v1y);
  double dot = v0x * v1x + v0y * v1y;

  double delta_s = dot/v1_norm;
  //具体含义及解释见式(3-4)
  return InterpolateUsingLinearApproximation(p0, p1, p0.s() + delta_s); }
  //p0.s() + delta_s 的含义及解释见式(3-4)
```

在匹配点的计算过程中主要用到了线性插值的方法。方法的理论解释在 3.3.2 节中进行了详细描述,代码如下:

```
//第 3 章/linear_interpolation.cc

#include "linear_interpolation.h"
#include <cmath>
#include "../cyber/common/log.h"
#include "math_utils.h"


PathPoint InterpolateUsingLinearApproximation(const PathPoint &p0,
                                              const PathPoint &p1,
                                              const double s) {
  double s0 = p0.s();
  double s1 = p1.s();
  //获取如图 3-3 中的 v_{i-1} 和 v_{i+1} 的 s 值,准备线性插值计算

  PathPoint path_point;
  double weight = (s - s0) / (s1 - s0);
  //此公式的具体说明见 3.3.2 节

  double x = (1 - weight) * p0.x() + weight * p1.x();
  double y = (1 - weight) * p0.y() + weight * p1.y();
  //计算过程的解释见式(3-11)

  double theta = slerp(p0.theta(), p0.s(), p1.theta(), p1.s(), s);
  double kappa = (1 - weight) * p0.kappa() + weight * p1.kappa();
```

```
    double dkappa = (1 - weight) * p0.dkappa() + weight * p1.dkappa();
    double ddkappa = (1 - weight) * p0.ddkappa() + weight * p1.ddkappa();
    //计算过程的解释同上

    path_point.set_x(x);
    path_point.set_y(y);
    path_point.set_theta(theta);
    path_point.set_kappa(kappa);
    path_point.set_dkappa(dkappa);
    path_point.set_ddkappa(ddkappa);
    path_point.set_s(s);
    return path_point;
  }
```

3.4 小结

车辆当前轨迹点的匹配点主要在参考线上寻找。需要读者注意的是匹配点并不是在所有离散参考点中找与车辆当前轨迹点距离最小的点。主要原因有两点：第一点是每个参考线点的 s 值的计算都是“以直代曲”的叠加；第二点则在于参考线在进行离散化时，由于采样点间隔的原因，所以也会存在误差。基于上面两点考虑，与当前车辆轨迹点的最小距离点的确定只是判定了匹配点的存在范围或区域，即在该点的前一个点和后一个点之间，而后，再计算车辆当前轨迹点的投影。这种计算方法的理解需要读者注意并加以灵活运用。