二次规划（`QP`）样条路径优化

1 目标函数

1.1 获得路径长度

路径定义在station-lateral坐标系中。**s**的变化区间为从车辆当前位置点到默认路径的长度。

1.2 获得样条段

将路径划分为**n**段，每段路径用一个多项式来表示。

1.3 定义样条段函数

每个样条段 ***i*** 都有沿着参考线的累加距离d_i。每段的路径默认用5介多项式表示。

$$l = f_i(s) = a_{i0} + a_{i1} \cdot s + a_{i2} \cdot s^2 + a_{i3} \cdot s^3 + a_{i4} \cdot s^4 + a_{i5} \cdot s^5 (0 \leq s \leq d_i) \tag{1}$$

1.4 定义每个样条段优化目标函数

$$cost = \sum_{i=1}^{n} \left(w_1 \cdot \int\limits_{0}^{d_i} (f_i')^2(s)ds + w_2 \cdot \int\limits_{0}^{d_i} (f_i'')^2(s)ds + w_3 \cdot \int\limits_{0}^{d_i} (f_i''')^2(s)ds \right) \tag{2}$$

1.5 将开销（cost）函数转换为 `QP`公式

`QP`公式:

$$\begin{aligned} minimize & \frac{1}{2} \cdot x^T \cdot H \cdot x + f^T \cdot x \\ s.t. \qquad & LB \leq x \leq UB \\ & A_{eq}x = b_{eq} \\ & Ax \geq b \end{aligned} \tag{3}$$

下面是将开销（cost）函数转换为 `QP`公式的例子:

$$f_i(s) = \begin{vmatrix} 1 & s & s^2 & s^3 & s^4 & s^5 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} \tag{4}$$

且

$$f_i'(s) = \begin{vmatrix} 0 & 1 & 2s & 3s^2 & 4s^3 & 5s^4 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} \tag{5}$$

且

$$f_i'(s)^2 = \begin{vmatrix} a_{i0} & a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} \end{vmatrix} \cdot \begin{vmatrix} 0 \\ 1 \\ 2s \\ 3s^2 \\ 4s^3 \\ 5s^4 \end{vmatrix} \cdot \begin{vmatrix} 0 & 1 & 2s & 3s^2 & 4s^3 & 5s^4 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} \tag{6}$$

然后得到,

$$\int\limits_{0}^{d_i} f_i'(s)^2 ds = \int\limits_{0}^{d_i} \begin{vmatrix} a_{i0} & a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} \end{vmatrix} \cdot \begin{vmatrix} 0 \\ 1 \\ 2s \\ 3s^2 \\ 4s^3 \\ 5s^4 \end{vmatrix} \cdot \begin{vmatrix} 0 & 1 & 2s & 3s^2 & 4s^3 & 5s^4 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} ds \tag{7}$$

从聚合函数中提取出常量得到,

$$\int\limits_{0}^{d_i} f'(s)^2 ds = \begin{vmatrix} a_{i0} & a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} \end{vmatrix} \cdot \int\limits_{0}^{d_i} \begin{vmatrix} 0 \\ 1 \\ 2s \\ 3s^2 \\ 4s^3 \\ 5s^4 \end{vmatrix} \cdot \begin{vmatrix} 0 & 1 & 2s & 3s^2 & 4s^3 & 5s^4 \end{vmatrix} ds \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} \tag{8}$$

$$\begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{vmatrix} \quad \begin{vmatrix} a_{i0} \end{vmatrix}$$

$$
= \begin{vmatrix} a_{i0} & a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} \end{vmatrix} \cdot \int\limits_0^{d_i} \begin{vmatrix} 0 & 1 & 2s & 3s^2 & 4s^3 & 5s^4 \\ 0 & 2s & 4s^2 & 6s^3 & 8s^4 & 10s^5 \\ 0 & 3s^2 & 6s^3 & 9s^4 & 12s^5 & 15s^6 \\ 0 & 4s^3 & 8s^4 & 12s^5 & 16s^6 & 20s^7 \\ 0 & 5s^4 & 10s^5 & 15s^6 & 20s^7 & 25s^8 \end{vmatrix} ds \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix}
$$

最后得到,

$$
\int\limits_0^{d_i} f_i'(s)^2 ds = \begin{vmatrix} a_{i0} & a_{i1} & a_{i2} & a_{i3} & a_{i4} & a_{i5} \end{vmatrix} \cdot \begin{vmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d_i & d_i^2 & d_i^3 & d_i^4 & d_i^5 \\ 0 & d_i^2 & \frac{4}{3}d_i^3 & \frac{6}{4}d_i^4 & \frac{8}{5}d_i^5 & \frac{10}{6}d_i^6 \\ 0 & d_i^3 & \frac{6}{4}d_i^4 & \frac{9}{5}d_i^5 & \frac{12}{6}d_i^6 & \frac{15}{7}d_i^7 \\ 0 & d_i^4 & \frac{8}{5}d_i^5 & \frac{12}{6}d_i^6 & \frac{16}{7}d_i^7 & \frac{20}{8}d_i^8 \\ 0 & d_i^5 & \frac{10}{6}d_i^6 & \frac{15}{7}d_i^7 & \frac{20}{8}d_i^8 & \frac{25}{9}d_i^9 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} \tag{9}
$$

请注意我们最后得到一个6介的矩阵来表示5介样条插值的衍生开销。
应用同样的推理方法可以得到2介，3介样条插值的衍生开销。

2 约束条件

2.1 初始点约束

假设第一个点为 (s_0, l_0), (s_0, l_0') and (s_0, l_0'')，其中l_0 , l_0' and l_0''表示横向的偏移，并且规划路径的起始点的第一，第二个点的衍生开销可以从$f_i(s)$, $f_i'(s)$, $f_i(s)''$计算得到。

将上述约束转换为 `QP` 约束等式，使用等式:

$$
A_{eq}x = b_{eq} \tag{10}
$$

下面是转换的具体步骤:

$$
f_i(s_0) = \begin{vmatrix} 1 & s_0 & s_0^2 & s_0^3 & s_0^4 & s_0^5 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} = l_0 \tag{11}
$$

且

$$
f_i'(s_0) = \begin{vmatrix} 0 & 1 & 2s_0 & 3s_0^2 & 4s_0^3 & 5s_0^4 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} = l_0' \tag{12}
$$

且

$$
f_i''(s_0) = \begin{vmatrix} 0 & 0 & 2 & 3 \times 2s_0 & 4 \times 3s_0^2 & 5 \times 4s_0^3 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} = l_0'' \tag{13}
$$

其中，i是包含s_0的样条段的索引值。

2.2 终点约束

和起始点相同，终点(s_e, l_e) 也应当按照起始点的计算方法生成约束条件。

将起始点和终点组合在一起，得出约束等式为:

$$
\begin{vmatrix} 1 & s_0 & s_0^2 & s_0^3 & s_0^4 & s_0^5 \\ 0 & 1 & 2s_0 & 3s_0^2 & 4s_0^3 & 5s_0^4 \\ 0 & 0 & 2 & 3 \times 2s_0 & 4 \times 3s_0^2 & 5 \times 4s_0^3 \\ 1 & s_e & s_e^2 & s_e^3 & s_e^4 & s_e^5 \\ 0 & 1 & 2s_e & 3s_e^2 & 4s_e^3 & 5s_e^4 \\ 0 & 0 & 2 & 3 \times 2s_e & 4 \times 3s_e^2 & 5 \times 4s_e^3 \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix} = \begin{vmatrix} l_0 \\ l_0' \\ l_0'' \\ l_e \\ l_e' \\ l_e'' \end{vmatrix} \tag{14}
$$

2.3 平滑节点约束

该约束的目的是使样条的节点更加平滑。假设两个段seg_k 和seg_{k+1}互相连接，且seg_k的累计值s为s_k。计算约束的等式为:

$$
f_k(s_k) = f_{k+1}(s_0) \tag{15}
$$

下面是计算的具体步骤:

$$
\begin{vmatrix} 1 & s_k & s_k^2 & s_k^3 & s_k^4 & s_k^5 \end{vmatrix} \cdot \begin{vmatrix} a_{k0} \\ a_{k1} \\ a_{k2} \\ \vdots \end{vmatrix} = \begin{vmatrix} 1 & s_0 & s_0^2 & s_0^3 & s_0^4 & s_0^5 \end{vmatrix} \cdot \begin{vmatrix} a_{k+1,0} \\ a_{k+1,1} \\ a_{k+1,2} \\ \vdots \end{vmatrix} \tag{16}
$$

$$\begin{vmatrix}1 & s_k & s_k & s_k & s_k & s_k\end{vmatrix} \cdot \begin{vmatrix} a_{k3} \\ a_{k4} \\ a_{k5}\end{vmatrix} = \begin{vmatrix}1 & s_0 & s_0 & s_0 & s_0 & s_0\end{vmatrix} \cdot \begin{vmatrix} a_{k+1,3} \\ a_{k+1,4} \\ a_{k+1,5}\end{vmatrix} \tag{16}$$

移项合并为:

$$\begin{vmatrix}1 & s_k & s_k^2 & s_k^3 & s_k^4 & s_k^5 & -1 & -s_0 & -s_0^2 & -s_0^3 & -s_0^4 & -s_0^5\end{vmatrix} \cdot \begin{vmatrix} a_{k0} \\ a_{k1} \\ a_{k2} \\ a_{k3} \\ a_{k4} \\ a_{k5} \\ a_{k+1,0} \\ a_{k+1,1} \\ a_{k+1,2} \\ a_{k+1,3} \\ a_{k+1,4} \\ a_{k+1,5}\end{vmatrix} = 0 \tag{17}$$

将$s_0 = 0$代入等式。

同样地，可以为下述等式计算约束等式:

$$\begin{aligned} f_k'(s_k) &= f_{k+1}'(s_0) \\ f_k''(s_k) &= f_{k+1}''(s_0) \\ f_k'''(s_k) &= f_{k+1}'''(s_0) \end{aligned} \tag{18}$$

三阶连续的表达式:

$$\begin{vmatrix}0 & 0 & 2 & 6s_k & 12s_k^2 & 20s_k^3 & 0 & 0 & -2 & -6s_0 & -12s_0^2 & -20s_0^3\end{vmatrix} \cdot \begin{vmatrix} a_{k0} \\ a_{k1} \\ a_{k2} \\ a_{k3} \\ a_{k4} \\ a_{k5} \\ a_{k+1,0} \\ a_{k+1,1} \\ a_{k+1,2} \\ a_{k+1,3} \\ a_{k+1,4} \\ a_{k+1,5}\end{vmatrix} = 0 \tag{19}$$

代码赋值:

```
bool Spline2dConstraint::AddSecondDerivativeSmoothConstraint() {
  if (t_knots_.size() < 3) {
    return true;
  }
  // 6个等式, affine_equality是系数, affine_boundary是值。约束函数数量:
  // 6 * (n-1), n=t_knots_.size()-1
  Eigen::MatrixXd affine_equality =
      Eigen::MatrixXd::Zero(6 * (t_knots_.size() - 2), total_param_);
  Eigen::MatrixXd affine_boundary =
      Eigen::MatrixXd::Zero(6 * (t_knots_.size() - 2), 1);
  // 相邻两个knots对之间的多项式拟合函数进行约束
  for (uint32_t i = 0; i + 2 < t_knots_.size(); ++i) {
    // 计算第一个曲线的自变量: t_knots[i+1].s-t_knots[i].s
    const double rel_t = t_knots_[i + 1] - t_knots_[i];
    const uint32_t num_params = spline_order_ + 1;
    const uint32_t index_offset = 2 * i * num_params;
    // 函数值系数: [1, s, s^2, s^3, s^4, s^5]
    std::vector<double> power_t = PolyCoef(rel_t);
    // 一阶导系数: [0, 1, 2s, 3s^2, 4s^3, 5s^4]
    std::vector<double> derivative_t = DerivativeCoef(rel_t);
    // 二阶导系数: [0, 0, 2, 6s, 12s^2,20s^3]
    std::vector<double> second_derivative_t = SecondDerivativeCoef(rel_t);
    for (uint32_t j = 0; j < num_params; ++j) {
      affine_equality(6 * i, j + index_offset) =
          power_t[j];  // 第一个多项式x曲线终点函数值
      affine_equality(6 * i + 1, j + index_offset) =
          derivative_t[j];  // 第一个多项式x曲线终点一阶导
      affine_equality(6 * i + 2, j + index_offset) =
          second_derivative_t[j];  // 第一个多项式x曲线终点二阶导
      affine_equality(6 * i + 3, j + index_offset + num_params) =
          power_t[j];  // 第二个多项式y曲线终点函数值
      affine_equality(6 * i + 4, j + index_offset + num_params) =
          derivative_t[j];  // 第二个多项式y曲线终点一阶导
      affine_equality(6 * i + 5, j + index_offset + num_params) =
          second_derivative_t[j];  // 第二个多项式y曲线终点二阶导
    }
    //后一段曲线的起始点 s=0
    affine_equality(6 * i, index_offset + 2 * num_params) =
```

```
            -1.0;  // 第一个多项式x曲线终点函数值 - 第二个多项式x曲线起点函数值
        affine_equality(6 * i + 1, index_offset + 2 * num_params + 1) =
            -1.0;  // 第一个多项式x曲线终点一阶导 -
                   // 第二个多项式x曲线起点一阶导(速度一致)
        affine_equality(6 * i + 2, index_offset + 2 * num_params + 2) =
            -2.0;  // 第一个多项式x曲线终点二阶导 -
                   // 第二个多项式x曲线起点二阶导(加速度一致)
        affine_equality(6 * i + 3, index_offset + 3 * num_params) =
            -1.0;  // 第一个多项式y曲线终点函数值 - 第二个多项式y曲线起点函数值
        affine_equality(6 * i + 4, index_offset + 3 * num_params + 1) =
            -1.0;  // 第一个多项式y曲线终点一阶导 -
                   // 第二个多项式y曲线起点一阶导(速度一致)
        affine_equality(6 * i + 5, index_offset + 3 * num_params + 2) =
            -2.0;  // 第一个多项式y曲线终点二阶导 -
                   // 第二个多项式y曲线起点二阶导(加速度一致)
    }
    return AddEqualityConstraint(affine_equality, affine_boundary);
}
```

2.4 点采样边界约束

在路径上均匀的取样**m**个点，检查这些点上的障碍物边界。将这些约束转换为 `QP` 约束不等式，使用不等式：

$$Ax \geq b \tag{20}$$

应该考虑的约束

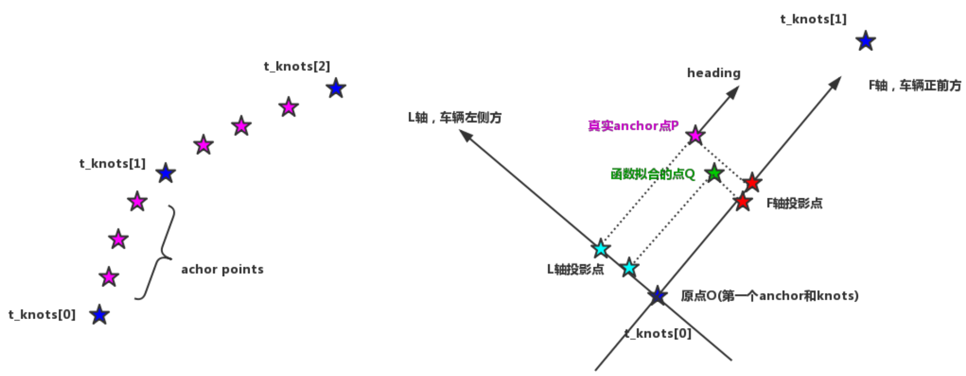
1. 预瞄点的x',y'应该保证在真实x,y的L轴lateral_bound、F轴longitudinal_bound范围内
2. 第一个预瞄点的heading和函数的一阶导方向需要一致，大小可以不一致
3. x和y的n段函数之间，两端曲线连接的部分应该是平滑的，两个函数值(位置)、一阶导(速度)、二阶导(加速度)必须一致

1.横纵向边界约束

每个anchor point相对第一个点的相对参考系坐标为(x,y)，方向为heading。而该点坐标在的段拟合出来的相对参考系坐标为(x',y')，坐标的计算方式为:

$$\begin{aligned} x' &= f_i(s) = a_{i0} + a_{i1}s + a_{i2}s^2 + a_{i3}s^3 + a_{i4}s^4 + a_{i5}s^5 \\ y' &= g_i(s) = b_{i0} + b_{i1}s + b_{i2}s^2 + b_{i3}s^3 + b_{i4}s^4 + b_{i5}s^5 \end{aligned} \tag{21}$$

其中i是anchor point所在的knots段，i=1,2,...,n(n=num_spline)，确定了i也就确定了这段曲线的函数表达式



1.1 确定真实点在FL轴上的投影

投影得到真实点在FL轴上的投影，确定预瞄点的边界范围，即预瞄点在真实点(图中蓝色点)的范围内

```
double Spline2dConstraint::SignDistance(const Vec2d& xy_point,
                                        const double angle) const {
  //点乘内积
  return common::math::InnerProd(
      xy_point.x(), xy_point.y(),
      -common::math::sin(common::math::Angle16::from_rad(angle)),
      common::math::cos(common::math::Angle16::from_rad(angle)));
}

double InnerProd(const double x0, const double y0, const double x1,
                 const double y1) {
  return x0 * x1 + y0 * y1;
}
```

先计算真实点坐标在前方FL轴上的投影，投影点到原点的距离，也就是前方距离计算方式为:

$$\begin{aligned} x_{p,later} &= (cos(\theta+\pi/2), sin(\theta+\pi/2)) \cdot (x,y) = (-sin\theta, cos\theta) \cdot (x,y) \\ y_{p,longi} &= (cos\theta, sin\theta) \cdot (x,y) \end{aligned} \tag{22}$$

1.2 确定预瞄点在FL轴上的投影

由五次多项式可以得到预瞄点的坐标x',y'，每个点的取值由采样间隔s约定(也就是代码中的t_coord[i])，那么通过FindIndex(i)得到所属的段的曲线函数表达式(知道了ai,bi)，带入采样间隔s就可以得到坐标x',y'.

总结：FindIndex(i) ==> 得到曲线表达式 ==>   带入s ==> 得到(x',y')

$$
\begin{aligned}
x' &= SA \\
y' &= SB
\end{aligned} \tag{23}
$$

其中：

$$
\begin{aligned}
S &= [1, s, s^2, s^3, s^4, s^5] \\
A &= [a_{i0}, a_{i1}, a_{i2}, a_{i3}, a_{i4}, a_{i5}]^T \\
B &= [b_{i0}, b_{i1}, b_{i2}, b_{i3}, b_{i4}, b_{i5}]^T
\end{aligned} \tag{24}
$$

```
const uint32_t index = FindIndex(t_coord[i]);
const double rel_t = t_coord[i] - t_knots_[index];
const uint32_t index_offset = 2 * index * (spline_order_ + 1);
```

上述代码中：

1. i是采样间隔
2. index是计算n个拟合段中anchor point所属的段。rel_t是anchor point累积距离s相对于之前的knots累积距离s的相对差，说白了就是自变量归一化到[0,1]之间
3. index_offset是该段拟合函数对应的参数位置，我们可以知道n段拟合多项式函数的参数总和为 2*(spline_order+1) * n。所以第i个拟合函数的参数偏移位置为2*(spline_order+1) * i

那么：

- [2*(spline_order+1) * i, 2*(spline_order+1) * i+(spline_order+1)]是x多项式函数的参数，共(spline_order+1)个，即向量A;
- [2*(spline_order+1) * i + (spline_order+1), 2*(spline_order+1) * (i+1)]是y多项式函数的参数，共(spline_order+1)个，即向量B

建立系数矩阵代码：

```
std::vector<double> Spline2dConstraint::AffineCoef(const double angle,
                                                    const double t) const {
  const uint32_t num_params = spline_order_ + 1;
  std::vector<double> result(num_params * 2, 0.0);
  double x_coef = -common::math::sin(common::math::Angle16::from_rad(angle));
  double y_coef = common::math::cos(common::math::Angle16::from_rad(angle));
  for (uint32_t i = 0; i < num_params; ++i) {
    result[i] = x_coef;
    result[i + num_params] = y_coef;
    x_coef *= t;
    y_coef *= t;
  }
  return result;
}
/* result = [-sina, -t*sina, -t^2*sina, -t^3*sina, -t^4*sina, -t^5*sina,
              cosa,  t*cosa,  t^2*cosa,  t^3*cosa,  t^4*cosa,  t^5*cosa]
*/
```

横纵向偏移函数的系数矩阵为：

$$
\begin{aligned}
lateralcoef &= [-sin\theta S, cos\theta S] \\
longitudinalcoef &= [cos\theta S, sin\theta S]
\end{aligned} \tag{25}
$$

根据系数矩阵得到预瞄点在FL上的投影距离为：

$$
\begin{aligned}
x'_{p,later} &= (-sin\theta, cos\theta) \cdot (x', y') = (-sin\theta, cos\theta) \cdot (SA, SB) = [-sin\theta S, cos\theta S].(A, B) = lateralcoef \cdot (A, B) \\
y'_{q,longi} &= (cos\theta, sin\theta) \cdot (x', y') = (cos\theta, sin\theta) \cdot (SA, SB) = [cos\theta S, sin\theta S].(A, B) = longitudinalcoef \cdot (A, B)
\end{aligned} \tag{26}
$$

矩阵形式为：

$$
x'_{p,longi} = (\begin{vmatrix} -sin\theta & -s_i \cdot sin\theta & -s_i^2 \cdot sin\theta & -s_i^3 \cdot sin\theta & -s_i^4 \cdot sin\theta & -s_i^5 \cdot sin\theta \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix}, |cos\theta \quad s_i \cdot cos\theta \quad s_i^2 \cdot cos\theta \quad s_i^3 \cdot cos\theta \quad s_i^4 \cdot cos
$$

$$
= lateralcoef \cdot (A, B)
$$

$$
y'_{p,longi} = (\begin{vmatrix} cos\theta & s_i \cdot cos\theta & s_i^2 \cdot cos\theta & s_i^3 \cdot cos\theta & s_i^4 \cdot cos\theta & s_i^5 \cdot cos\theta \end{vmatrix} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \end{vmatrix}, |sin\theta \quad s_i \cdot sin\theta \quad s_i^2 \cdot sin\theta \quad s_i^3 \cdot sin\theta \quad s_i^4 \cdot sin\theta \quad s_i^5
$$

$$
= longitudinalcoef \cdot (A, B)
$$

根据上面得到的真实点在FL轴上的投影和预瞄点在FL上的投影，得到预瞄点的上下边界为：

$$
x - x' \le lateralBound ==> \quad x' \ge x - lateralBound
$$

$$y_{p,later} - y'_{p,later} \le longitudinalBound ==> \quad y'_{p,later} \ge y_{p,later} - longitudinalBound \tag{28}$$

其中: `lateralBound` 和 `longitudinalBound` 默认为0.2

不等式约束矩阵为:

```
Eigen::MatrixXd affine_inequality =
      Eigen::MatrixXd::Zero(4 * t_coord.size(), total_param_);
  Eigen::MatrixXd affine_boundary =
      Eigen::MatrixXd::Zero(4 * t_coord.size(), 1);
  for (uint32_t i = 0; i < t_coord.size(); ++i) {
    const double d_lateral = SignDistance(ref_point[i], angle[i]);
    const double d_longitudinal =
        SignDistance(ref_point[i], angle[i] - M_PI / 2.0);
    const uint32_t index = FindIndex(t_coord[i]);
    const double rel_t = t_coord[i] - t_knots_[index];
    const uint32_t index_offset = 2 * index * (spline_order_ + 1);
    std::vector<double> longi_coef = AffineDerivativeCoef(angle[i], rel_t);
    std::vector<double> longitudinal_coef =
        AffineDerivativeCoef(angle[i] - M_PI / 2, rel_t);
    for (uint32_t j = 0; j < 2 * (spline_order_ + 1); ++j) {
      // upper longi 设置L轴上界不等式系数
      affine_inequality(4 * i, index_offset + j) = longi_coef[j];
      // lower longi 设置L轴下界不等式系数
      affine_inequality(4 * i + 1, index_offset + j) = -longi_coef[j];
      // upper longitudinal 设置F轴上界不等式系数
      affine_inequality(4 * i + 2, index_offset + j) = longitudinal_coef[j];
      // lower longitudinal 设置F轴下界不等式系数
      affine_inequality(4 * i + 3, index_offset + j) = -longitudinal_coef[j];
    }

    affine_boundary(4 * i, 0) =
        d_lateral - lateral_bound[i];  //设置L轴上界不等式的边界
    affine_boundary(4 * i + 1, 0) =
        -d_lateral - lateral_bound[i];  //设置L轴下界不等式的边界
    affine_boundary(4 * i + 2, 0) =
        d_longitudinal - longitudinal_bound[i];  //设置F轴上界不等式的边界
    affine_boundary(4 * i + 3, 0) =
        -d_longitudinal - longitudinal_bound[i];  //设置F轴下界不等式的边界
  }

//total_param_ =
//      2 * (spline_order_ + 1) * (static_cast<uint32_t>(t_knots.size()) - 1);
```

不等式约束矩阵为:

$$affineInequality = \begin{bmatrix} longi_coef \\ -longi_coef \\ longitudinal_coef \\ -longitudinal_coef \end{bmatrix}_{4*tcoord\times totalparam} \cdot \begin{vmatrix} a_{i0} \\ a_{i1} \\ a_{i2} \\ a_{i3} \\ a_{i4} \\ a_{i5} \\ b_{i0} \\ b_{i1} \\ b_{i2} \\ b_{i3} \\ b_{i4} \\ b_{i5} \\ \dots \end{vmatrix}_{totalparam\times 1} \ge \begin{vmatrix} l_{lb,0} \\ l_{lb,1} \\ \dots \\ l_{lb,m} \end{vmatrix}_{totalparam\times 1} \tag{29}$$

式中:

$$\begin{bmatrix} longi_coef \\ -longi_coef \\ longitudinal_coef \\ -longitudinal_coef \end{bmatrix}_{4*tcoord\times totalparam} = \begin{bmatrix} -sin\theta & -s_i\cdot sin\theta & -s_i^2\cdot sin\theta & -s_i^3\cdot sin\theta & -s_i^4\cdot sin\theta & -s_i^5\cdot sin\theta & cos\theta & s_i\cdot cos\theta \\ sin\theta & s_i\cdot sin\theta & s_i^2\cdot sin\theta & s_i^3\cdot sin\theta & s_i^4\cdot sin\theta & s_i^5\cdot sin\theta & -cos\theta & -s_i\cdot cos\theta \\ cos\theta & s_i\cdot cos\theta & s_i^2\cdot cos\theta & s_i^3\cdot cos\theta & s_i^4\cdot cos\theta & s_i^5\cdot cos\theta & sin\theta & s_i\cdot sin\theta \\ -cos\theta & -s_i\cdot cos\theta & -s_i^2\cdot cos\theta & -s_i^3\cdot cos\theta & -s_i^4\cdot cos\theta & -s_i^5\cdot cos\theta & -sin\theta & -s_i\cdot sin\theta \\ \dots \end{bmatrix}$$

其中:

- tcoord = 采样点的个数(每个采样点分为**横向上下边界**和**纵向上下边界**，所以要乘以4)
- t_knots= 控制点的个数 = 分段数量 +1
- totalparam = 2 * (spline_order_ + 1) * ((t_knots.size()) - 1); (每段的约束参数 * 分段数量)

1.3 约束条件设置

现在可以计算真实点和拟合点在F轴L轴的投影，那么就有约束条件:

`|d_lateral - longi_coef·(A, B)| <= lateral_bound`

`|d_longitudinal - longitudinal_coef(A, B)| <= longitudinal_bound`

最后得到四个约束不等式:

L轴上界不等式
`d_lateral - longi_coef·(A, B) <= lateral_bound`

整理得到：`longi_coef·(A, B) >= d_lateral - lateral_bound`

L轴下界不等式
`d_lateral - longi_coef·(A, B) >= -lateral_bound`

整理得到：`-longi_coef·(A, B) >= -d_lateral - lateral_bound`

F轴上界不等式
`d_longitudinal - longitudinal_coef·(A, B) <= longitudinal_bound`

整理得到：`longitudinal_coef·(A, B) >= d_longitudinal - longitudinal_bound`

F轴下界不等式
`d_longitudinal - longitudinal_coef·(A, B) >= -longitudinal_bound`

整理得到：`-longitudinal_coef·(A, B) >= -d_longitudinal - longitudinal_bound`

```
 for (uint32_t j = 0; j < 2 * (spline_order_ + 1); ++j) {
      // upper longi 设置L轴上界不等式系数
      affine_inequality(4 * i, index_offset + j) = longi_coef[j];
      // lower longi 设置L轴下界不等式系数
      affine_inequality(4 * i + 1, index_offset + j) = -longi_coef[j];
      // upper longitudinal 设置F轴上界不等式系数
      affine_inequality(4 * i + 2, index_offset + j) = longitudinal_coef[j];
      // lower longitudinal 设置F轴下界不等式系数
      affine_inequality(4 * i + 3, index_offset + j) = -longitudinal_coef[j];
    }

    affine_boundary(4 * i, 0) =
        d_lateral - lateral_bound[i];  //设置L轴上界不等式的边界
    affine_boundary(4 * i + 1, 0) =
        -d_lateral - lateral_bound[i];  //设置L轴下界不等式的边界
    affine_boundary(4 * i + 2, 0) =
        d_longitudinal - longitudinal_bound[i];  //设置F轴上界不等式的边界
    affine_boundary(4 * i + 3, 0) =
        -d_longitudinal - longitudinal_bound[i];  //设置F轴下界不等式的边界
  }
```

配合代码和上述的公式可以不难看出不等式系数的设置和边界设置。经过上述赋值:

`affine_inequality`等同于: `[longi_coef, -longi_coef, longitudinal_coef, -longitudinal_coef]`

最后不等式约束:

`affine_inequality * [A1,B1,A2,B2,..An,Bn] >= affine_boundary`

等式约束同理

2.方向约束

- L轴分量为0，保证方向相同或者相反
- 验证同向性

2.1 L轴分量为0，保证方向相同或者相反

```
bool Spline2dConstraint::AddPointAngleConstraint(const double t,
                                                 const double angle) {
  // add equality constraint
  Eigen::MatrixXd affine_equality = Eigen::MatrixXd::Zero(1, total_param_);
  Eigen::MatrixXd affine_boundary = Eigen::MatrixXd::Zero(1, 1);
  std::vector<double> line_derivative_coef = AffineDerivativeCoef(angle, rel_t);
  for (uint32_t i = 0; i < line_derivative_coef.size(); ++i) {
    affine_equality(0, i + index_offset) = line_derivative_coef[i];
  }
  //可以得到L轴方向分量的计算方式为 line_derivative_coef · (A, B) = 0,  表示斜率在L轴方向上的分量为0
 if (!AddEqualityConstraint(affine_equality, affine_boundary)) {
    return false;
  }
}

std::vector<double> Spline2dConstraint::AffineDerivativeCoef(
    const double angle, const double t) const {
  const uint32_t num_params = spline_order_ + 1;
  std::vector<double> result(num_params * 2, 0.0);
  double x_coef = -std::sin(angle);
  double y_coef = std::cos(angle);
  std::vector<double> power_t = PolyCoef(t);
  for (uint32_t i = 1; i < num_params; ++i) {
    result[i] = x_coef * power_t[i - 1] * i;
    result[i + num_params] = y_coef * power_t[i - 1] * i;
  }
  return result;
}
```

第一个点的方向heading和多项式曲线在该点的斜率(一阶导)方向必须一致，大小可以不一致。**方向一致等价于：斜率在L轴方向上的分量为0**

代码中通过`Spline2dConstraint::AffineDerivativeCoef`函数计算得到的系数矩阵`line_derivative_coef`为:

微分矩阵 $D = [0, 1, 2s, 3s^2, 4s^3, 5s^4]$

$linederivativecoef = [-sin(\theta)D, cos(\theta)D]$

可以得到L轴方向分量的计算方式为 $linederivativecoef \cdot (A, B) = 0$

从代码我们可以看到一个问题：只是限制了L轴分量为零，但是不保证同向性。

2.2 验证同向性

真实点的方向为heading，拟合多项式在该点的一阶导数为 `(D·A, D·B)`。代码把heading做一规则化到[0, 2*pi]

计算heading的方向向量sgn = [x_sign, y_sign]，计算方法为：

- 如果正则化heading在[0, pi/2]: sgn = [1, 1]
- 如果正则化heading在[pi/2, pi]: sgn = [-1, 1]
- 如果正则化heading在[pi,3*pi/2]: sgn = [-1, -1]
- 如果正则化heading在[3* pi/2, 2*pi]: sgn = [1, -1]

只需要最后的内积 `sgn·(D·A, D·B) > 0`表明方向一致。

```
 // add inequality constraint
  Eigen::MatrixXd affine_inequality = Eigen::MatrixXd::Zero(2, total_param_);
  const Eigen::MatrixXd affine_inequality_boundary =
      Eigen::MatrixXd::Zero(2, 1);
  std::vector<double> t_coef = DerivativeCoef(rel_t);
  int x_sign = 1;
  int y_sign = 1;
  //角度归一化处理 将角度限制在(-π, π)之间
  double normalized_angle = fmod(angle, M_PI * 2);

  if (normalized_angle < 0) {
    normalized_angle += M_PI * 2;
  }

  if (normalized_angle > (M_PI / 2) && normalized_angle < (M_PI * 1.5)) {
    x_sign = -1;
  }

  if (normalized_angle >= M_PI) {
    y_sign = -1;
  }

  for (uint32_t i = 0; i < t_coef.size(); ++i) {
    affine_inequality(0, i + index_offset) = t_coef[i] * x_sign;
    affine_inequality(1, i + index_offset + num_params) = t_coef[i] * y_sign;
  }
...
//内积 sgn·(D·A, D·B) > 0表明方向一致
  return AddInequalityConstraint(affine_inequality, affine_inequality_boundary);
```