Apollo6.0_ReferenceLine_Smoother解析与子方法对比

以视觉检测的车道线为引导线，比如Mobileye输出的车道线，因其三次多项式形式，曲率切向较连续；而Hdmap、众包地图或者slam建图所给出的引导线，往往精度、位置存在“毛刺”，这样的引导线不能直接用于后续的运动规划，影响至下游控制模块；因此，对于原始引导线加上一层平滑处理尤为重要。

本文主要为apollo referenceLineSmooth模块的解析，主要包含模块代码的入口以及执行流程等；此外列举了子方法之间的相同点与差异点对比；最后通过实例来测试这几种子方法的时间和效果对比，可供各位开发者依据场景选用合适的方法去适配各自项目开发需求做参考。

在此，感谢本文中索引的博客大佬们将知识share，感谢Baidu apollo开源算法!。

如有错误之处，恳请各位大佬指正！。

一、模块函数入口与执行流程

已有同学对参考线平滑算法架构进行解析，没接触的同学可以先看下这位同学的文章《Apollo 6.0 参考线平滑算法解析》。

引导线平滑方法在planning/reference_line目录下reference_line_provider.cc文件中调用实现;

以OnLanePlanning class 为例，引导线平滑流程如下：

- 1. 在OnLanePlanning::Init()函数中实例化

ReferenceLineProvider指针对象;

-

2. 在ReferenceLineProvider构造函数中通过GFLAGS读取proto配置文件(`FLAGS_smoother_config_filename`)，apollo默认采用qp_spline smooth方法;

```
DEFINE_string(smoother_config_filename,
              "/apollo/modules/planning/conf/qp_spline_smoother_config.pb.txt",
              "The configuration file for qp_spline smoother");
```

-

3. 执行ReferenceLineProvider::Start()函数，依据GFLAGS变量定义决定是否另开线程;

-

4. 若另开线程，则通过ReferenceLineProvider::GenerateThread()，否则，通过ReferenceLineProvider::GetReferenceLines()调用CreateReferenceLine();

-

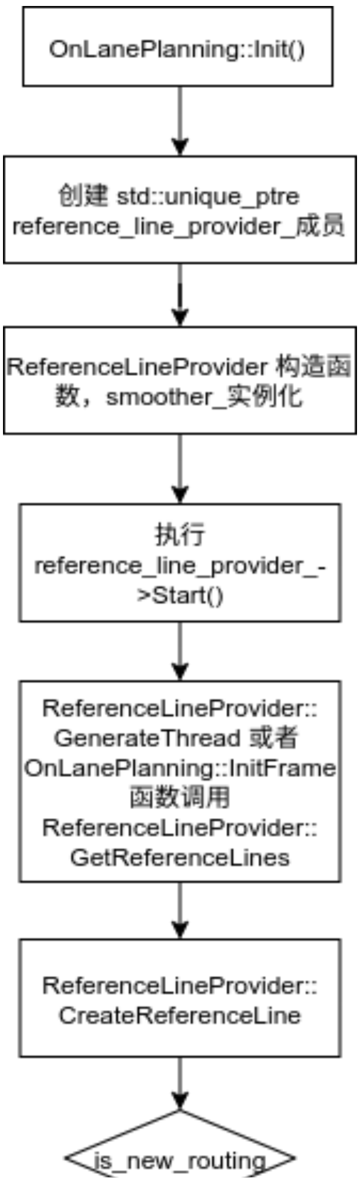
5. 依据is_new_routing是否为新的地图route,调用SmoothRouteSegments()函数;

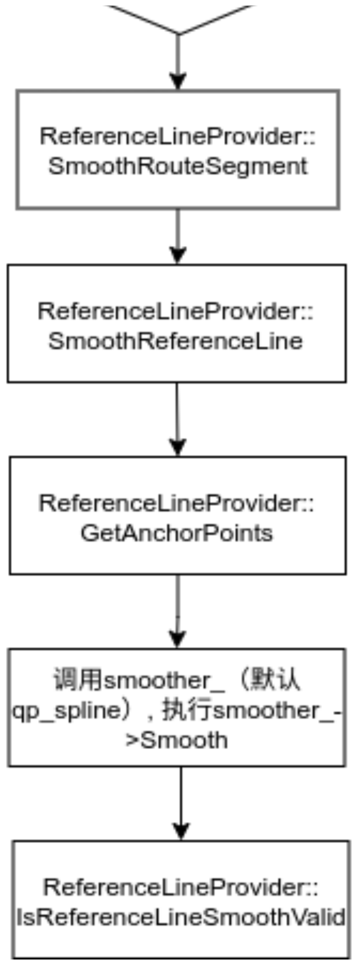
-

6. 调用执行ReferenceLineProvider::SmoothReferenceLine()
   - GetAnchorPoints:
     - 降采样：将引导线等分(按max_constraint_interval间隔);
     - 依据s在reference_line上插值生成anchor_point;
   - SetAnchorPoints:将anhor_points传入值smoother_中;
   - 调用实例化smoother_ Smooth()函数，qp_spline每段长度为max_spline_length；
   - IsReferenceLineSmoothValid判断平滑后的引导线与原始引导线的偏差是否满足要求;

```
bool ReferenceLineProvider::SmoothReferenceLine(
    const ReferenceLine &raw_reference_line, ReferenceLine *reference_line) {
  if (!FLAGS_enable_smooth_reference_line) {
    *reference_line = raw_reference_line;
    return true;
  }
  // generate anchor points:
  std::vector<AnchorPoint> anchor_points;
  GetAnchorPoints(raw_reference_line, &anchor_points);
  smoother_->SetAnchorPoints(anchor_points);
  if (!smoother_->Smooth(raw_reference_line, reference_line)) {
    AERROR << "Failed to smooth reference line with anchor points";
    return false;
  }
  if (!IsReferenceLineSmoothValid(raw_reference_line, *reference_line)) {
    AERROR << "The smoothed reference line error is too large";
    return false;
  }
  return true;
}
```


具体流程图如下





二、模块子方法

上一章节，主要介绍了引导线平滑算法架构与具体执行流程。接下来，主要对子方法的差异进行对比。

1) 子算法简述与对比

Apollo 6.0 参考线平滑方法具体有三大种方法，其中离散点平滑方法包含两种方法

-
  1. QpSplineReferenceLineSmoother
-
  2. SpiralReferenceLineSmoother
-
  3. DiscretePointsReferenceLineSmoother

    -
        1. COS_THETA_SMOOTHING
    -
        2. FEM_POS_DEVIATION_SMOOTHING

在这里，这几种平滑方法的详细公式推导不做重复讲解，网上已有同学写过这几种方法公式推导的文章，大家可以通过以下链接去阅读一下，感兴趣的同学还可以自己推导一下，加强印象。

- 离散点平滑原理及公式推导
- QpSpline平滑原理及公式推导
- spiral平滑原理及公式推导

平滑方法汇总对比

- 优化变量
  n段spline x和y五次多项式系数
- 目标函数cost:
    -
        1. x和y的二阶导代价
    -
        2. x和y的三阶到代价
    -
        3. x和y的L2正则项代价
           $cost=\sum_{i=1}^{n}\left(\int_0^{t_i} f''(x)^2 dt+\int_0^{t_i} g''(y)^2 dt+\int_0^{t_i} f'''(x)^2 dt+\int_0^{t_i} g'''(y)^2 dt\right)$

转成二次型后再加上正则项代价;L2范数正则化，对参数变量进行惩罚,防止过拟合;处理矩阵求逆困难;

代码见`qp_spline_reference_line_smoother.cc`中`AddKernel`函数

```
bool QpSplineReferenceLineSmoother::AddKernel() {
  Spline2dKernel* kernel = spline_solver_->mutable_kernel();
```

```
  // add spline kernel
  if (config_.qp_spline().second_derivative_weight() > 0.0) {
    kernel->AddSecondOrderDerivativeMatrix(
        config_.qp_spline().second_derivative_weight());
  }
  if (config_.qp_spline().third_derivative_weight() > 0.0) {
    kernel->AddThirdOrderDerivativeMatrix(
        config_.qp_spline().third_derivative_weight());
  }

  kernel->AddRegularization(config_.qp_spline().regularization_weight(
  return true;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16

- 约束条件
  - 优化后点与原始点的位置偏差约束;
  - 起始点航向要与原始点航向一致;
  - 前后spline在交接点处位置、一阶导、二阶导连续;

代码见qp_spline_reference_line_smoother.cc中AddConstraint函数

```
bool QpSplineReferenceLineSmoother::AddConstraint() {
  // Add x, y boundary constraint
  std::vector<double> headings;
  std::vector<double> longitudinal_bound;
  std::vector<double> lateral_bound;
  std::vector<common::math::Vec2d> xy_points;
  for (const auto& point : anchor_points_) {
    const auto& path_point = point.path_point;
    headings.push_back(path_point.theta());
    longitudinal_bound.push_back(point.longitudinal_bound);
    lateral_bound.push_back(point.lateral_bound);
    xy_points.emplace_back(path_point.x() - ref_x_, path_point.y() - r
  }
  const double scale = (anchor_points_.back().path_point.s() -
                        anchor_points_.front().path_point.s()) /
                       (t_knots_.back() - t_knots_.front());
  std::vector<double> evaluated_t;
  for (const auto& point : anchor_points_) {
    evaluated_t.emplace_back(point.path_point.s() /
                             scale);  //每段内t属于[a, a+1]
  }

  auto* spline_constraint = spline_solver_->mutable_constraint();

  // 1. all points (x, y) should not deviate anchor points by a boundi
  if (!spline_constraint->Add2dBoundary(evaluated_t, headings, xy_poin
                                        longitudinal_bound, lateral_bc
    AERROR << "Add 2d boundary constraint failed.";
    return false;
  }

  // 2. the heading of the first point should be identical to the anch
  if (FLAGS_enable_reference_line_stitching &&
```

```
      !spline_constraint->AddPointAngleConstraint(evaluated_t.front(),
                                                  headings.front())) {
    AERROR << "Add 2d point angle constraint failed.";
    return false;
  }
  // 3. all spline should be connected smoothly to the second order de
  if (!spline_constraint->AddSecondDerivativeSmoothConstraint()) {
    AERROR << "Add jointness constraint failed.";
    return false;
  }

  return true;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46

- 优化变量
  n段五次多项式回旋线的坐标点信息$\theta, \dot{\theta}, \ddot{\theta}, x, y, \Delta s$
- 目标函数
  - 引导线长度
  - 曲率
  - 曲率变化率
    $cost = w_{length}\sum_{i=1}^{n-1}\Delta s_i + w_{kappa}\sum_{i=1}^{n-1}\sum_{i=0}^{m-1}(\dot{\theta}(s_j))^2 + w_{dkappa}\sum_{i=1}^{n-1}\sum_{i=0}^{m-1}(\ddot{\theta}(s_j))^2$

代码见`spiral_problem_interface.cc`中`eval_f`函数

此处有一点需要大家留意一下，每一段spiral线段 其实都等分成num_of_internal_points_份,以此来计算曲率、曲率变化率代价

```
bool SpiralProblemInterface::eval_f(int n, const double* x, bool new_x
                                    double& obj_value) {
  CHECK_EQ(n, num_of_variables_);
  if (new_x) {
    update_piecewise_spiral_paths(x, n);
  }

  obj_value = 0.0;
  for (int i = 0; i + 1 < num_of_points_; ++i) {
    const auto& spiral_curve = piecewise_paths_[i];
    double delta_s = spiral_curve.ParamLength();

    obj_value += delta_s * weight_curve_length_;
    for (int j = 0; j < num_of_internal_points_; ++j) {
      double ratio =
          static_cast<double>(j) / static_cast<double>(num_of_internal
      double s = ratio * delta_s;

      double kappa = spiral_curve.Evaluate(1, s);
      obj_value += kappa * kappa * weight_kappa_;

      double dkappa = spiral_curve.Evaluate(2, s);
      obj_value += dkappa * dkappa * weight_dkappa_;
    }
  }
  return true;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7

- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27

- 约束条件
  - 连接点等式约束
    $$\theta_{i+1}=\theta_i(\Delta s_i) \quad \dot{\theta}_{i+1}=\dot{\theta}_i(\Delta s_i) \quad \ddot{\theta}_{i+1}=\ddot{\theta}_i(\Delta s_i) \quad x_{i+1}=x_i+\int_0^{\Delta s_i}\cos(\theta(s))ds \quad i\in[0,n-2] \quad y_{i+1}=y_i+\int_0^{\Delta s_i}\sin(\theta(s))ds \quad i\in[0,n-2]$$
  - 起点约束
    $$\begin{cases}\theta_0=\theta_{start} \\ \dot{\theta}_0=\dot{\theta}_{start} \\ \ddot{\theta}_0=\ddot{\theta}_{start} \\ x_0=x_{start} \\ y_0=y_{start}\end{cases}$$
  - 终点约束
    $$\begin{cases}\theta_n=\theta_{start} \\ \dot{\theta}_n=\dot{\theta}_{start} \\ \ddot{\theta}_n=\ddot{\theta}_{start} \\ x_n=x_{start} \\ y_n=y_{start}\end{cases}$$
  - 中间优化变量约束
    $$\begin{cases}\theta_i^{relative}-\pi/2<=\theta_i<=\theta_i^{relative}+\pi/2 \\ -0.25<=\dot{\theta}_i<=0.25 \\ -0.02<=\ddot{\theta}_i<=0.02 \\ x_i^{ref}-r_i<=x_i<=x_i^{ref}+r_i \\ y_i^{ref}-r_i<=y_i<=y_i^{ref}+r_i \\ y_i^{ref}\end{cases}$$

$-r_i <= \Delta s_i <= distance_i * \pi/2$

- 中间点位置偏差约束
  $(x_i - x_{iref})^2 + (y_i - y_{iref})^2 <= r_i^2, i \in [0, n-1]$, i为整数

约束条件代码见`spiral_problem_interface.cc`中`get_bounds_info`函数和`eval_g`函数

```
bool SpiralProblemInterface::get_bounds_info(int n, double* x_l, doubl
                                             int m, double* g_l, doubl
  CHECK_EQ(n, num_of_variables_);
  CHECK_EQ(m, num_of_constraints_);

  // variables
  // a. for theta, kappa, dkappa, x, y
  for (int i = 0; i < num_of_points_; ++i) {
    int index = i * 5;

    double theta_lower = 0.0;
    double theta_upper = 0.0;
    double kappa_lower = 0.0;
    double kappa_upper = 0.0;
    double dkappa_lower = 0.0;
    double dkappa_upper = 0.0;
    double x_lower = 0.0;
    double x_upper = 0.0;
    double y_lower = 0.0;
    double y_upper = 0.0;
    if (i == 0 && has_fixed_start_point_) {
      theta_lower = start_theta_;
      theta_upper = start_theta_;
      kappa_lower = start_kappa_;
      kappa_upper = start_kappa_;
      dkappa_lower = start_dkappa_;
      dkappa_upper = start_dkappa_;
      x_lower = start_x_;
      x_upper = start_x_;
```

```
    y_lower = start_y_;
    y_upper = start_y_;

  } else if (i + 1 == num_of_points_ && has_fixed_end_point_) {
    theta_lower = end_theta_;
    theta_upper = end_theta_;
    kappa_lower = end_kappa_;
    kappa_upper = end_kappa_;
    dkappa_lower = end_dkappa_;
    dkappa_upper = end_dkappa_;
    x_lower = end_x_;
    x_upper = end_x_;
    y_lower = end_y_;
    y_upper = end_y_;
  } else if (i + 1 == num_of_points_ && has_fixed_end_point_position
    theta_lower = relative_theta_[i] - M_PI * 0.2;
    theta_upper = relative_theta_[i] + M_PI * 0.2;
    kappa_lower = -0.25;
    kappa_upper = 0.25;
    dkappa_lower = -0.02;
    dkappa_upper = 0.02;
    x_lower = end_x_;
    x_upper = end_x_;
    y_lower = end_y_;
    y_upper = end_y_;
  } else {
    theta_lower = relative_theta_[i] - M_PI * 0.2;
    theta_upper = relative_theta_[i] + M_PI * 0.2;
    kappa_lower = -0.25;
    kappa_upper = 0.25;
    dkappa_lower = -0.02;
    dkappa_upper = 0.02;
    x_lower = init_points_[i].x() - default_max_point_deviation_;
    x_upper = init_points_[i].x() + default_max_point_deviation_;
    y_lower = init_points_[i].y() - default_max_point_deviation_;
    y_upper = init_points_[i].y() + default_max_point_deviation_;
  }
```

```
    // theta
    x_l[index] = theta_lower;
    x_u[index] = theta_upper;

    // kappa
    x_l[index + 1] = kappa_lower;
    x_u[index + 1] = kappa_upper;

    // dkappa
    x_l[index + 2] = dkappa_lower;
    x_u[index + 2] = dkappa_upper;

    // x
    x_l[index + 3] = x_lower;
    x_u[index + 3] = x_upper;

    // y
    x_l[index + 4] = y_lower;
    x_u[index + 4] = y_upper;
  }

  // b. for delta_s
  int variable_offset = num_of_points_ * 5;
  for (int i = 0; i + 1 < num_of_points_; ++i) {
    x_l[variable_offset + i] =
        point_distances_[i] - 2.0 * default_max_point_deviation_;
    x_u[variable_offset + i] = point_distances_[i] * M_PI * 0.5;
  }

  // constraints
  // a. positional equality constraints
  for (int i = 0; i + 1 < num_of_points_; ++i) {
    // for x
    g_l[i * 2] = 0.0;
    g_u[i * 2] = 0.0;
```

```
      // for y
      g_l[i * 2 + 1] = 0.0;
      g_u[i * 2 + 1] = 0.0;
    }
    // b. positional deviation constraints
    int constraint_offset = 2 * (num_of_points_ - 1);
    for (int i = 0; i < num_of_points_; ++i) {
      g_l[constraint_offset + i] = 0.0;
      g_u[constraint_offset + i] =
          default_max_point_deviation_ * default_max_point_deviation_;
    }
    return true;
  }
```

- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60

- 61
- 62
- 63
- 64
- 65
- 66
- 67
- 68
- 69
- 70
- 71
- 72
- 73
- 74
- 75
- 76
- 77
- 78
- 79
- 80
- 81
- 82
- 83
- 84
- 85
- 86
- 87
- 88
- 89
- 90
- 91
- 92
- 93
- 94
- 95
- 96
- 97

```
bool SpiralProblemInterface::eval_g(int n, const double* x, bool new_x
                                    double* g) {
  CHECK_EQ(n, num_of_variables_);
  CHECK_EQ(m, num_of_constraints_);

  if (new_x) {
    update_piecewise_spiral_paths(x, n);
  }

  // first, fill in the positional equality constraints
  for (int i = 0; i + 1 < num_of_points_; ++i) {
    int index0 = i * 5;
    int index1 = (i + 1) * 5;

    const auto& spiral_curve = piecewise_paths_[i];
    double delta_s = spiral_curve.ParamLength();
```

```
    //x(i+1)和x(i) 等式约束
    double x_diff = x[index1 + 3] - x[index0 + 3] -
                    spiral_curve.ComputeCartesianDeviationX(delta_s);
    g[i * 2] = x_diff * x_diff;
    //y(i+1)和x(i) 等式约束
    double y_diff = x[index1 + 4] - x[index0 + 4] -
                    spiral_curve.ComputeCartesianDeviationY(delta_s);
    g[i * 2 + 1] = y_diff * y_diff;
  }

  // second, fill in the positional deviation constraints
  //位置平移 约束
  int constraint_offset = 2 * (num_of_points_ - 1);
  for (int i = 0; i < num_of_points_; ++i) {
    int variable_index = i * 5;
    double x_cor = x[variable_index + 3];
    double y_cor = x[variable_index + 4];

    double x_diff = x_cor - init_points_[i].x();
    double y_diff = y_cor - init_points_[i].y();

    g[constraint_offset + i] = x_diff * x_diff + y_diff * y_diff;
  }
  return true;
}
```

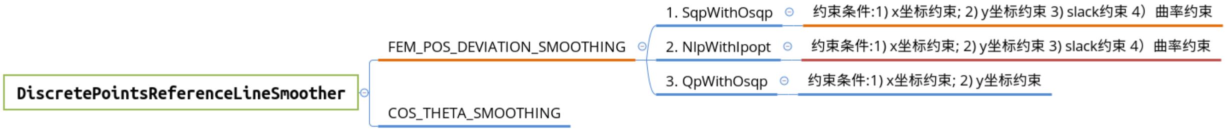
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12

- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41

三、discrete_points_smoother

FEM_POS_DEVIATION_SMOOTHING与COS_THETA_SMOOTHING的代价函数差异在于平滑度函数的计算方式，且其约束条件更为简单;
Apollo离散点平滑默认采用的是FEM_POS_DEVIATION_SMOOTHING

其中FEM_POS_DEVIATION_SMOOTHING有着三种实现方式，方法之间存在轻微差异，见下图：



COS_THETA_SMOOTHING

- 优化变量
  n个离散的(x_i,y_i)
- 目标函数
  - 曲线平滑度
  - 与参考点位置偏差
  - 轨迹长度代价

$$\begin{cases} cost1 = \dfrac{(x_i-x_{i-1})*(x_{i+1}-x_i)+(y_i-y_{i-1})*(y_{i+1}-y_i)}{\sqrt{(x_i-x_{i-1})^2+(y_i-y_{i-1})^2}\cdot\sqrt{(x_{i+1}-x_i)^2+(y_{i+1}-y_i)^2}} \\ cost2 = \sum_{i=0}^{n-2}((x_i-x_{i+1})^2+(y_i-y_{i+1})^2) \\ cost3 = \sum_{i=0}^{n-1}((x_i-x_{iref})^2+(y_i-y_{iref})^2) \\ cost = cost1+cost2+cost2 \end{cases}$$

目标函数代码见`cos_theta_ipopt_interface.cc`中`eval_f`函数

```
bool CosThetaIpoptInterface::eval_f(int n, const double* x, bool new_x
                                    double& obj_value) {
  CHECK_EQ(static_cast<size_t>(n), num_of_variables_);
  if (use_automatic_differentiation_) {
    eval_obj(n, x, &obj_value);
    return true;
  }
  obj_value = 0.0;
  for (size_t i = 0; i < num_of_points_; ++i) {
```

```
    size_t index = i << 1;
    obj_value +=
        (x[index] - ref_points_[i].first) * (x[index] - ref_points_[i]
        (x[index + 1] - ref_points_[i].second) *
            (x[index + 1] - ref_points_[i].second);
  }
  for (size_t i = 0; i < num_of_points_ - 2; i++) {
    size_t findex = i << 1;
    size_t mindex = findex + 2;
    size_t lindex = mindex + 2;
    obj_value -=
        weight_cos_included_angle_ *
        (((x[mindex] - x[findex]) * (x[lindex] - x[mindex])) +
         ((x[mindex + 1] - x[findex + 1]) * (x[lindex + 1] - x[mindex
        std::sqrt((x[mindex] - x[findex]) * (x[mindex] - x[findex]) +
                  (x[mindex + 1] - x[findex + 1]) *
                      (x[mindex + 1] - x[findex + 1])) /
        std::sqrt((x[lindex] - x[mindex]) * (x[lindex] - x[mindex]) +
                  (x[lindex + 1] - x[mindex + 1]) *
                      (x[lindex + 1] - x[mindex + 1]));
  }
  return true;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

- 约束条件
  - x约束
  - y约束

约束条件代码见cos_theta_ipopt_interface.cc中get_bounds_info函数

```
bool CosThetaIpoptInterface::get_bounds_info(int n, double* x_l, doubl
                                              int m, double* g_l, doubl
  CHECK_EQ(static_cast<size_t>(n), num_of_variables_);
  CHECK_EQ(static_cast<size_t>(m), num_of_constraints_);
  // variables
  // a. for x, y
  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i << 1;
    // x
    x_l[index] = -1e20;
```

```
      x_u[index] = 1e20;

      // y
      x_l[index + 1] = -1e20;
      x_u[index + 1] = 1e20;
    }

    // constraints
    // positional deviation constraints
    for (size_t i = 0; i < num_of_points_; ++i) {
      size_t index = i << 1;
      double x_lower = 0.0;
      double x_upper = 0.0;
      double y_lower = 0.0;
      double y_upper = 0.0;

      x_lower = ref_points_[i].first - bounds_[i];
      x_upper = ref_points_[i].first + bounds_[i];
      y_lower = ref_points_[i].second - bounds_[i];
      y_upper = ref_points_[i].second + bounds_[i];

      // x
      g_l[index] = x_lower;
      g_u[index] = x_upper;

      // y
      g_l[index + 1] = y_lower;
      g_u[index + 1] = y_upper;
    }
    return true;
  }
```

- 1
- 2
- 3
- 4
- 5
- 6

- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41

FEM_POS_DEVIATION_SMOOTHING

- 优化变量
  n个离散的(x_i,y_i)
- 目标函数
  - 曲线平滑度
  - 轨迹长度
  - 与参考位置偏移
    $$\begin{cases} cost1=\sum_{i=1}^{n-1}((x_{i-1}+x_{i+1}-2x_i)^2+(y_{i-1}+y_{i+1}-2y_i)^2) \\ cost2=\sum_{i=0}^{n-2}((x_i-x_{i+1})^2+(y_i-y_{i+1})^2) \\ cost3=\sum_{i=0}^{n-1}((x_i-x_{iref})^2+(y_i-y_{iref})^2) \\ cost=cost1+cost2+cost2 \end{cases}$$

以`SqpWithOsqp`方法为例,目标函数代码见

`fem_pos_deviation_ipopt_interface.cc`中`eval_obj`函数

```
bool FemPosDeviationIpoptInterface::eval_obj(int n, const T* x, T* obj
  *obj_value = 0.0;

  // Distance to refs
  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;
    *obj_value +=
        weight_ref_deviation_ *
        ((x[index] - ref_points_[i].first) * (x[index] - ref_points_[i
         (x[index + 1] - ref_points_[i].second) *
             (x[index + 1] - ref_points_[i].second));
  }

  // Fem_pos_deviation
  for (size_t i = 0; i + 2 < num_of_points_; ++i) {
    size_t findex = i * 2;
    size_t mindex = findex + 2;
    size_t lindex = mindex + 2;

    *obj_value += weight_fem_pos_deviation_ *
                  (((x[findex] + x[lindex]) - 2.0 * x[mindex]) *
```

```
                    ((x[findex] + x[lindex]) - 2.0 * x[mindex]) +
                ((x[findex + 1] + x[lindex + 1]) - 2.0 * x[mindex +
                    ((x[findex + 1] + x[lindex + 1]) - 2.0 * x[mind
  }

  // Total length
  for (size_t i = 0; i + 1 < num_of_points_; ++i) {
    size_t findex = i * 2;
    size_t nindex = findex + 2;
    *obj_value +=
        weight_path_length_ *
        ((x[findex] - x[nindex]) * (x[findex] - x[nindex]) +
         (x[findex + 1] - x[nindex + 1]) * (x[findex + 1] - x[nindex +
  }

  // Slack variable minimization
  for (size_t i = slack_var_start_index_; i < slack_var_end_index_; ++
    *obj_value += weight_curvature_constraint_slack_var_ * x[i];
  }

  return true;
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15

- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40
- 41
- 42
- 43

- 约束条件
  - x约束
  - y约束
  - 松弛变量约束
  - 曲率约束

    $$\begin{cases} xiref - ri <= xi <= xiref + ri \\ yiref - ri <= yi <= yiref + ri \\ 0.0 < slacki <= \infty \\ (xi-1 + xi+1 - 2xi)2 + \end{cases}$$

$$(y_{i-1}+y_{i+1}-2y_i)^2-slack_i<=(\Delta s^2*curvlimit)^2$$

以SqpWithOsqp方法为例,约束变量代码见fem_pos_deviation_ipopt_interface.cc中get_bounds_info函数

```
bool FemPosDeviationIpoptInterface::get_bounds_info(int n, double* x_l
                                                    double* x_u, int m
                                                    double* g_l, doubl
  CHECK_EQ(static_cast<size_t>(n), num_of_variables_);
  CHECK_EQ(static_cast<size_t>(m), num_of_constraints_);
  // variables
  // a. for x, y
  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;
    // x
    x_l[index] = -1e20;
    x_u[index] = 1e20;

    // y
    x_l[index + 1] = -1e20;
    x_u[index + 1] = 1e20;
  }
  // b. for slack var
  for (size_t i = slack_var_start_index_; i < slack_var_end_index_; ++
    x_l[i] = -1e20;
    x_u[i] = 1e20;
  }

  // constraints
  // a. positional deviation constraints
  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;
    // x
    g_l[index] = ref_points_[i].first - bounds_around_refs_[i];
    g_u[index] = ref_points_[i].first + bounds_around_refs_[i];
```

```
    // y
    g_l[index + 1] = ref_points_[i].second - bounds_around_refs_[i];
    g_u[index + 1] = ref_points_[i].second + bounds_around_refs_[i];
  }

  // b. curvature constraints
  double ref_total_length = 0.0;
  auto pre_point = ref_points_.front();
  for (size_t i = 1; i < num_of_points_; ++i) {
    auto cur_point = ref_points_[i];
    double x_diff = cur_point.first - pre_point.first;
    double y_diff = cur_point.second - pre_point.second;
    ref_total_length += std::sqrt(x_diff * x_diff + y_diff * y_diff);
    pre_point = cur_point;
  }
  double average_delta_s =
      ref_total_length / static_cast<double>(num_of_points_ - 1);
  double curvature_constr_upper =
      average_delta_s * average_delta_s * curvature_constraint_;

  for (size_t i = curvature_constr_start_index_;
       i < curvature_constr_end_index_; ++i) {
    g_l[i] = -1e20;
    g_u[i] = curvature_constr_upper * curvature_constr_upper;
  }

  // c. slack var constraints
  for (size_t i = slack_constr_start_index_; i < slack_constr_end_inde
    g_l[i] = 0.0;
    g_u[i] = 1e20;
  }

  return true;
}
```

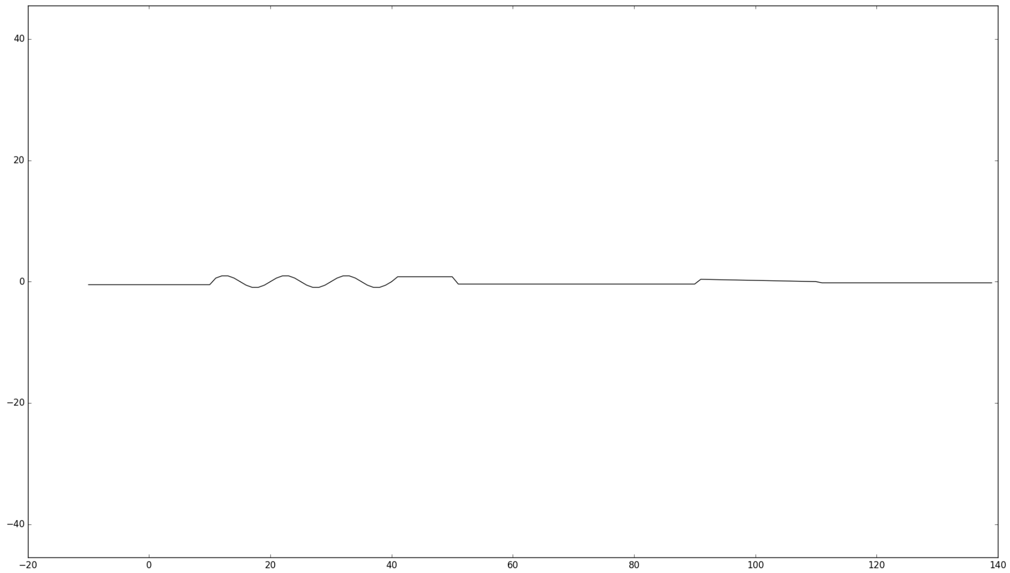
- 1
- 2
- 3

- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36
- 37
- 38
- 39
- 40

- 41
- 42
- 43
- 44
- 45
- 46
- 47
- 48
- 49
- 50
- 51
- 52
- 53
- 54
- 55
- 56
- 57
- 58
- 59
- 60
- 61
- 62
- 63
- 64
- 65

2) 实例测试

输入为一条由阶跃、正弦(+=1m偏差)、斜坡曲线组成的路径，总长150m，相邻点间隔约1m
路径如下图所示：



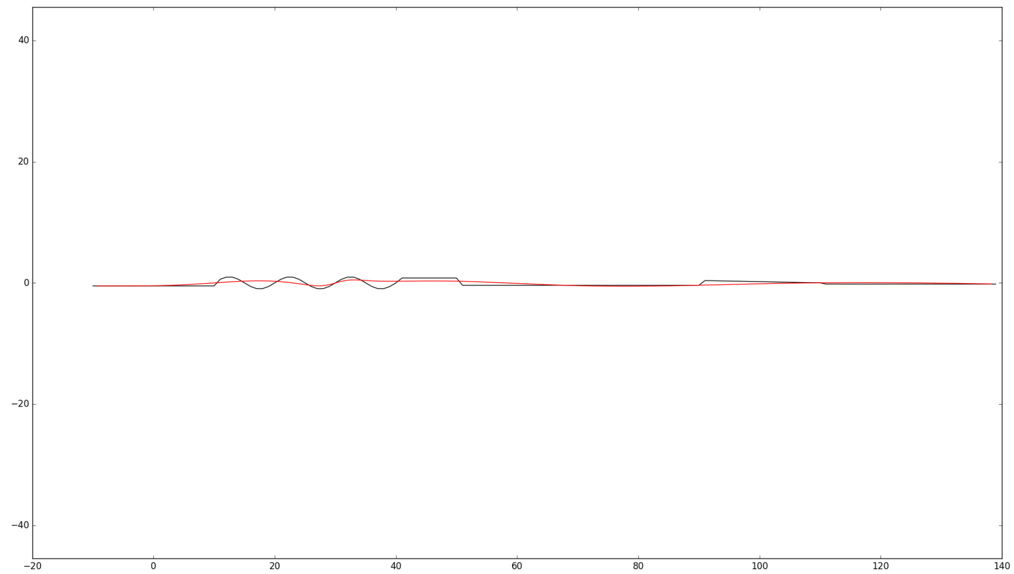
路径平滑耗时与结果质量很受等分间隔、侧向和纵向偏差范围、优化迭代次数以及原始路径形状等影响，以下测试结果仅供大家参考；如有差异，请大家反馈交流。

qp_spline_smoother

配置文件参数如下

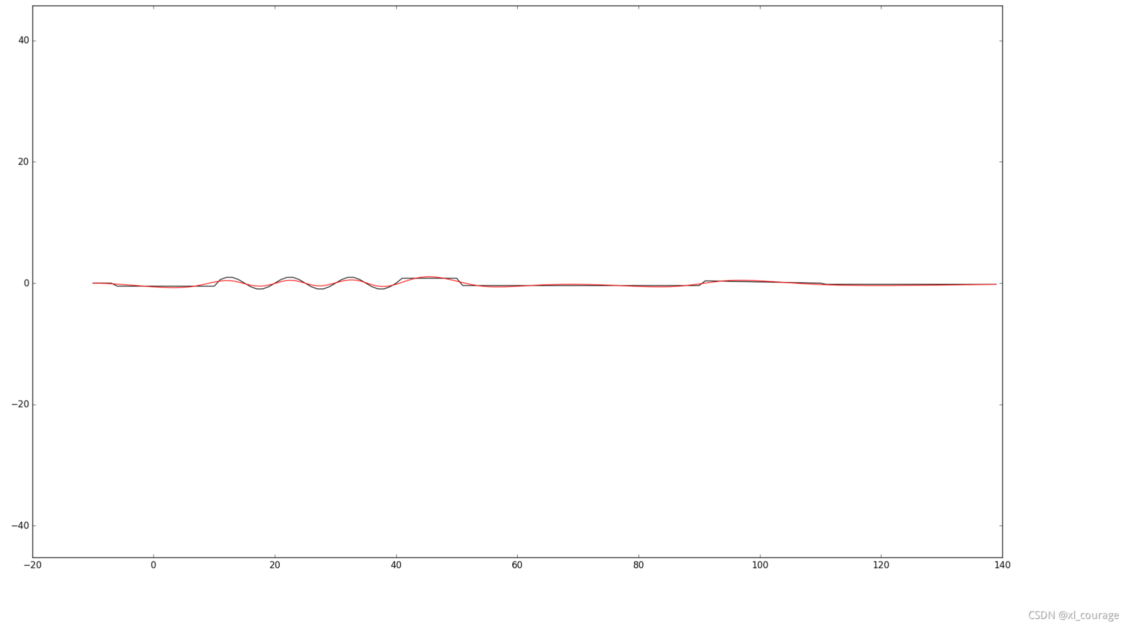
```
max_constraint_interval : 5.0
longitudinal_boundary_bound : 2.0
max_lateral_boundary_bound : 0.3
min_lateral_boundary_bound : 0.05
num_of_total_points : 200
curb_shift : 0.2
lateral_buffer : 0.2

qp_spline {
  spline_order: 5
  max_spline_length : 15.0 #25.0
  regularization_weight : 1.0e-5
  second_derivative_weight : 200.0
  third_derivative_weight : 1000.0
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15



如上图所示，因anchor_point间隔为5m,优化后的轨迹形状上很难与原始轨迹接近，局部轨迹偏差较大，但优化的曲线平滑度较高;耗时40-60ms左右;

将`max_constraint_interval`改成2m,`max_spline_length`改成6m，优化后的曲线与原始曲线较贴合;耗时50-90ms;结果见下图:



spiral_smoother

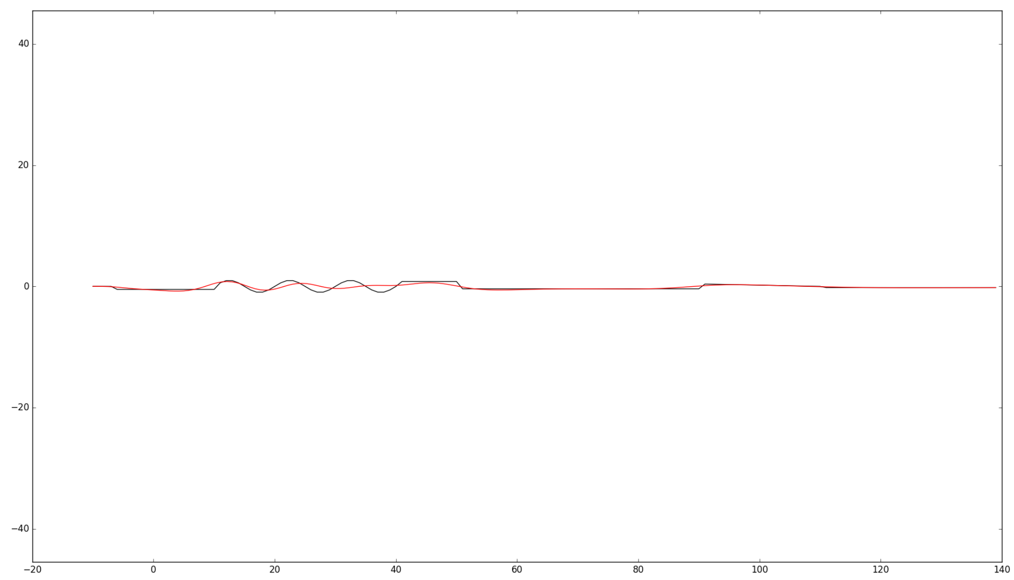
配置参数如下

```
max_constraint_interval : 6.0 #5.0
longitudinal_boundary_bound : 2.0
max_lateral_boundary_bound : 0.5
min_lateral_boundary_bound : 0.1
resolution : 1.0  #0.02
curb_shift : 0.2
lateral_buffer : 0.2

spiral {
  max_deviation: 0.2 #0.05
  piecewise_length : 10.0 #10.0
  max_iteration : 500
  opt_tol : 1.0e-3  #1e-6
  opt_acceptable_tol : 1e-2 #1e-4
  opt_acceptable_iteration : 15 #15
  weight_curve_length : 1.0
  weight_kappa : 1.0 #1.0
  weight_dkappa : 100.0 #100.0
```

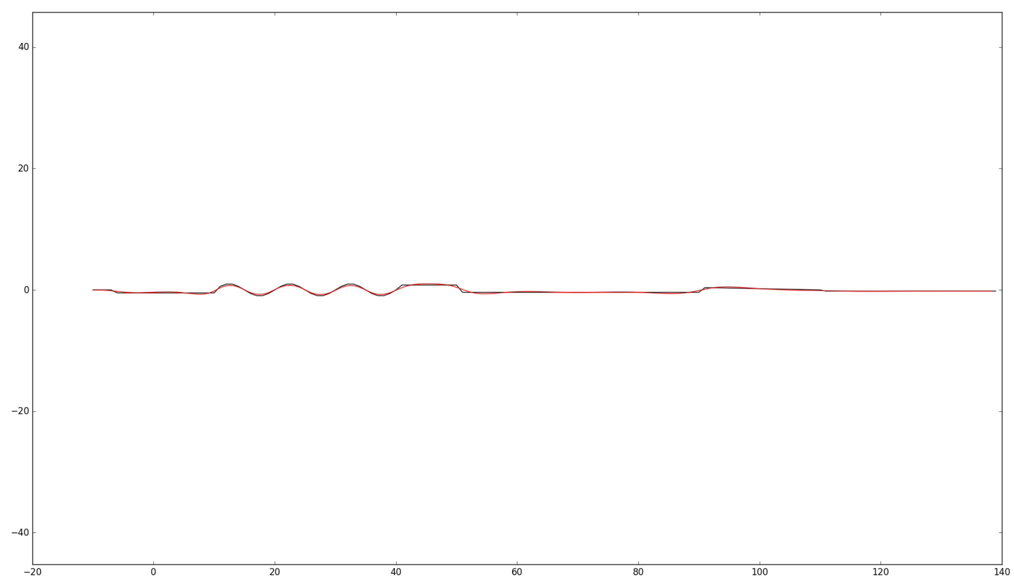
```
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19

如下图所示，spiral_smooth优化后的曲线整体位置与原始曲线偏差稍大，与qp_spline优化一样，受`max_constraint_interval`影响，`max_constraint_interval`越小，曲线越贴近原始曲线，平滑也会降低,该工况下耗时100-200ms;



将max_constraint_interval改成2m,耗时增至秒级，优化后的曲线贴近原始曲线;结果见下图。



discrete_points_smoother

配置参数如下

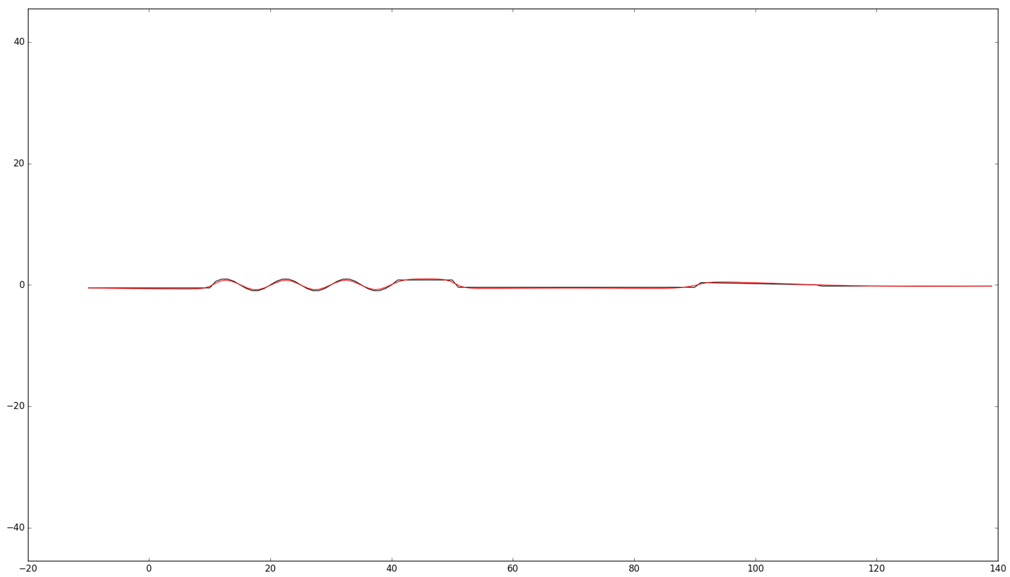
```
max_constraint_interval : 1.0
```

```
longitudinal_boundary_bound : 0.3
max_lateral_boundary_bound : 0.25
min_lateral_boundary_bound : 0.05
curb_shift : 0.2
lateral_buffer : 0.2

discrete_points {
  smoothing_method: FEM_POS_DEVIATION_SMOOTHING
  fem_pos_deviation_smoothing {
    weight_fem_pos_deviation: 1e9
    weight_ref_deviation: 1.0
    weight_path_length: 0.3
    apply_curvature_constraint: false
    curvature_constraint: 0.2
    use_sqp: true
    max_iter: 500
    time_limit: 0.0
    verbose: false
    scaled_termination: true
    warm_start: true
    print_level: 0
  }
}
```

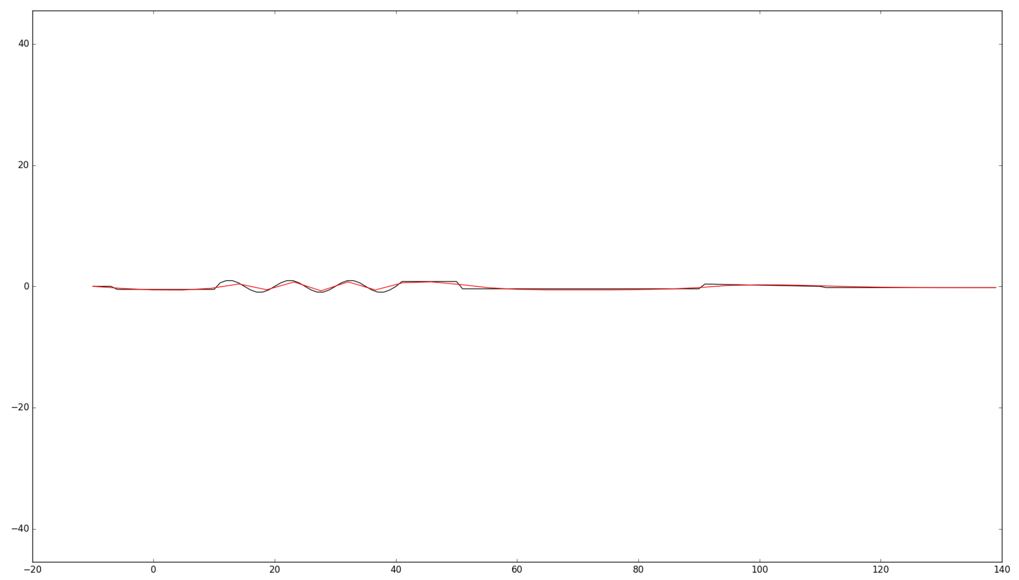
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14

- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24

如下图所示，优化的曲线非常贴近原始曲线，确保优化后的曲线不失真，同时该配置参数下，优化总耗时1-5ms;



将`max_constraint_interval`改成5m,优化后的曲线与原始曲线偏差较大，且曲率不连续，平滑度差；耗时小于1ms;效果如下图：



三、总结

以上对比，本人难以做到"绝对"的标准一致，参数配置只能尽力做到相对评价统一。**以下总结，仅代表个人从算法实践经验中的总结，如有偏颇，恳请指正。**

- 耗时: discrete_smoother < qp_spline_smoother <= spiral_spline
- 平滑度: qp_spline_smoother == spiral_smoother >= discrete_smoother
- qp_spline_smoother方法对与引导线切向突变大的工况时平滑效果稍差，需要大幅减小anchor_point之间的距离间隔以及spline长度，以此来达到优化后的轨迹贴合原始轨迹;算法耗时适中;平滑度高;贴和原始轨迹能力偏弱
- spiral_smoother方法贴合原始轨迹能力稍好于qp_spline_smoother;平滑度高;算法耗时高;
- discrete_smoother方法耗时很少;轨迹平滑度、曲率连续性差，但可通过缩小`max_constraint_interval`来接近"曲率连续";不受首尾点曲率、航向约束;方法简单

如果你觉得内容还不错，麻烦点赞支持下哈 😄 😄 ❤️

四、参考

- apollo
- 知乎-引导线平滑
- Ipopt优化实例