无人驾驶算法——Baidu Apollo代码解析之ReferenceLine Smoother参考线平滑

> Date: 2020/12/15
> Editor:萧潇子（Jesse）
> Contact: 1223167600@qq.com

Apollo 参考线平滑类

Apollo主要的参考线平滑类有三个：QpSplineReferenceLineSmoother、SpiralReferenceLineSmoother和DiscretePointsReferenceLineSmoother。

reference_line_provider.cc

```
ReferenceLineProvider::ReferenceLineProvider(
    const common::VehicleStateProvider *vehicle_state_provider,
    const hdmap::HDMap *base_map,
    const std::shared_ptr<relative_map::MapMsg> &relative_map)
    : vehicle_state_provider_(vehicle_state_provider) {
  if (!FLAGS_use_navigation_mode) {
    pnc_map_ = std::make_unique<hdmap::PncMap>(base_map);
    relative_map_ = nullptr;
  } else {
    pnc_map_ = nullptr;
    relative_map_ = relative_map;
  }

  ACHECK(cyber::common::GetProtoFromFile(FLAGS_smoother_config_filenam
                                         &smoother_config_))
      << "Failed to load smoother config file "
      << FLAGS_smoother_config_filename;
```

```
  if (smoother_config_.has_qp_spline()) {
    smoother_.reset(new QpSplineReferenceLineSmoother(smoother_config_
  } else if (smoother_config_.has_spiral()) {
    smoother_.reset(new SpiralReferenceLineSmoother(smoother_config_))
  } else if (smoother_config_.has_discrete_points()) {

    smoother_.reset(new DiscretePointsReferenceLineSmoother(smoother_c
  } else {
    ACHECK(false) << "unknown smoother config "
                  << smoother_config_.DebugString();
  }
  is_initialized_ = true;
}
```

- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32

其中DiscretePointsReferenceLineSmoother中包含了两个方法：CosThetaSmoother和FemPosDeviationSmoother。**本文主要介绍散点平滑的建模过程（代码中公式的物理意义以及推导）**

代价函数

cos_theta_ipopt_interface.cc

以n个点为例子$P0,P1,P2,P3,\cdots,Pn$

初始化点

```
bool CosThetaIpoptInterface::get_starting_point(int n, bool init_x, dc
                                                bool init_z, double* z
                                                double* z_U, int m,
                                                bool init_lambda,
                                                double* lambda) {
  CHECK_EQ(static_cast<size_t>(n), num_of_variables_);
  std::random_device rd;
  std::default_random_engine gen = std::default_random_engine(rd());
  std::normal_distribution<> dis{0, 0.05};

  for (size_t i = 0; i < num_of_points_; ++i) {
```

```
    size_t index = i << 1;
    x[index] = ref_points_[i].first + dis(gen);
    x[index + 1] = ref_points_[i].second + dis(gen);
  }
  return true;
}
```

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

代价函数包含3部分

```
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);
}
```

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

第1部分——参考点距离代价

$$obj_{value1} = \sum_{i=0}^{n-1} (x_i - x_{i-ref})^2 + (y_i - y_{i-ref})^2$$

```
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]));
  }
  }
```

第2部分——平滑性代价

$$obj_{value2} = -\sum_{i=0}^{n-2} (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2$$

$$\sqrt{(x_{i+2}-x_{i+1})^2+(y_{i+2}-y_{i+1})^2}$$

$$\sqrt{(x_{i+1}-x_i)\times(x_{i+2}-x_{i+1})+(y_{i+1}-y_i)\times(y_{i+2}-y_{i+1})}$$

```
  for (size_t i = 0; i < num_of_points_ - 1; ++i) {
    size_t findex = i << 1;
    size_t nindex = findex + 2;
    *obj_value +=
        weight_length_ *
        ((x[findex] - x[nindex]) * (x[findex] - x[nindex]) +
         (x[findex + 1] - x[nindex + 1]) * (x[findex + 1] - x[nindex +
  }

  return true;
}
```

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

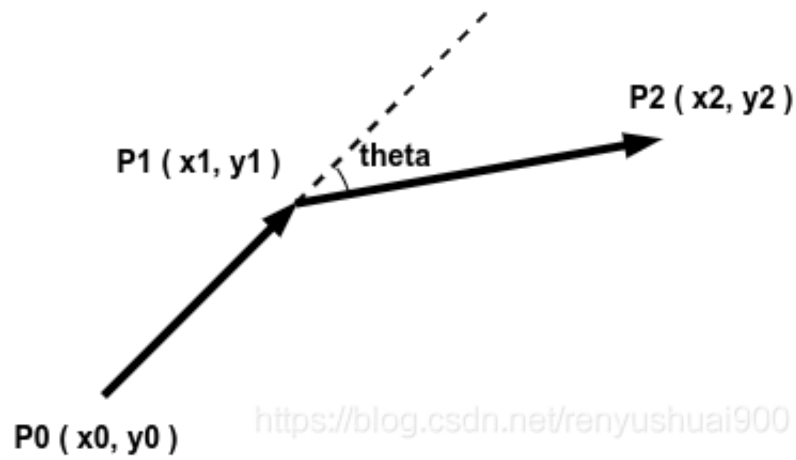
第3部分——总长度代价

$$objvalue3=\sum_{i=0}^{n-1}(x_i-x_{i+1})^2+(y_i-y_{i+1})^2$$

总代价函数：

$$objvalue=w1\times objvalue1+w2\times objvalue2+w3\times objvalue3$$

这里解释一下**第2部分代价函数代码中的公式的意义**：



cos_theta方法： = 前面是 - 号

连续3点 $P0,P1,P2$ 如上图所示，其中向量$\overrightarrow{P0P1}=(x1-x0,y1-y0)$和$\overrightarrow{P1P2}=(x2-x1,y2-y1)$之间的夹角$\theta$

$$cos\theta=\frac{\overrightarrow{P0P1}\cdot\overrightarrow{P1P2}}{|\overrightarrow{P0P1}||\overrightarrow{P1P2}|}=(x1-x0)\times(x2-x1)+(y1-y0)\times(y2-y1)/\sqrt{(x1-x0)2+(y1-y0)2}/\sqrt{(x2-x1)2+(y2-y1)2}$$

***cosθ*值越大，*θ*越小，*P0,P1,P2* 越接近直线，曲线越平滑**

---

fem_pos_deviation_ipopt_interface.cc

代价函数：

以n个点为例子$P0,P1,P2,P3,\cdots,Pn$

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

```
  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));
  }




  return true;
}
```

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

- 22

第1部分——参考点距离代价

$$obj_{value1}=\sum_{i=0}^{n-1}(x_i-x_{i-ref})^2+(y_i-y_{i-ref})^2$$

```
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
}
```

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

第2部分——平滑性代价

$$obj_{value2}=\sum_{i=0}^{n-2}(x_i+x_{i+2}-2\times x_{i+1})^2+(y_i+y_{i+2}-2\times y_{i+1})^2$$

```
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 +
}
```

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9

第3部分——总长度代价
$objvalue3=\sum_{i=0}^{n-1}(x_i-x_{i+1})^2+(y_i-y_{i+1})^2$

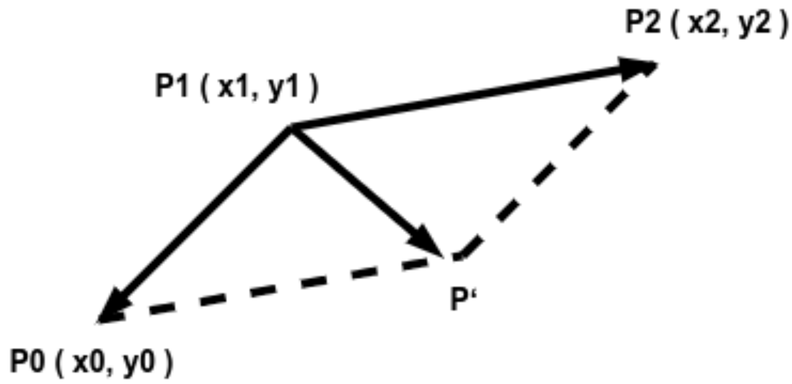
```
for (size_t i = slack_var_start_index_; i < slack_var_end_index_; ++
  *obj_value += weight_curvature_constraint_slack_var_ * x[i];
}
```

- 1
- 2
- 3
- 4
- 5

第4部分——对于松弛变量的惩罚也是常规做法，可以在后面约束的

部分看到松弛变量的意义。

总代价函数:
$objvalue=w1\times objvalue1+w2\times objvalue2+w3\times objvalue3$
这里解释一下第2部分代价函数代码中的公式的意义:



Fem_pos_deviation方法: = 前面是 + 号
$(xi+xi+2-2\times xi+1)2+(yi+yi+2-2\times yi+1)2$

其中上式的物理意义如上图中$\vec{P1P'}$向量的模的平方，它可以理解为：向量$\vec{P1P0}$和 向量$\vec{P1P2}$相加结果新的向量$\vec{P1P'}$的模平方。如果这三个点在一条直线上，那么这个值最小；三个点组成的两个线段的夹角θ越大，即曲线越平滑。

约束

约束条件:

```
template <class T>
bool FemPosDeviationIpoptInterface::eval_constraints(int n, const T* x
                                                     T* g) {

  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;
    g[index] = x[index];
```

```
    g[index + 1] = x[index + 1];
  }


  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;
    g[curvature_constr_start_index_ + i] =
        (((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 + 1]) *
             ((x[findex + 1] + x[lindex + 1]) - 2.0 * x[mindex + 1]))
        x[slack_var_start_index_ + i];
  }


  size_t slack_var_index = 0;
  for (size_t i = slack_constr_start_index_; i < slack_constr_end_inde
    g[i] = x[slack_var_start_index_ + slack_var_index];
    ++slack_var_index;
  }
  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

边界条件：

```
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_);


  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;
```

```
    x_l[index] = -1e20;
    x_u[index] = 1e20;


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

  for (size_t i = slack_var_start_index_; i < slack_var_end_index_; ++
    x_l[i] = -1e20;
    x_u[i] = 1e20;
  }



  for (size_t i = 0; i < num_of_points_; ++i) {
    size_t index = i * 2;

    g_l[index] = ref_points_[i].first - bounds_around_refs_[i];
    g_u[index] = ref_points_[i].first + bounds_around_refs_[i];


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


  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;
  }


  for (size_t i = slack_constr_start_index_; i < slack_constr_end_inde
    g_l[i] = 0.0;
    g_u[i] = 1e20;
  }

  return true;
}
```

- 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
- 66

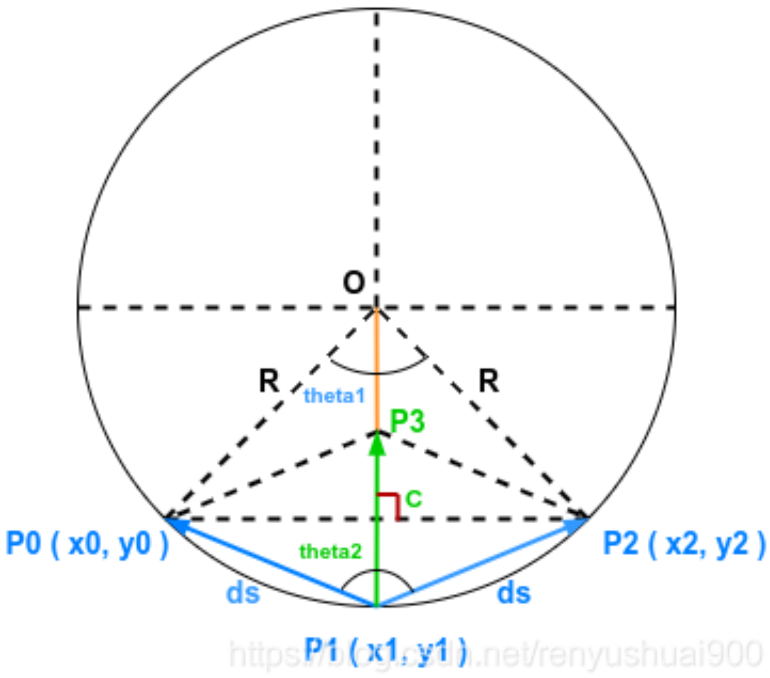
这里解释一下对曲率约束的上届：

对于除去首尾的每个anchor point（因为需要三个点计算曲率，故首尾无法计算），需要满足：

$$(x_i+x_{i+2}-2\times x_{i+1})^2+(y_i+y_{i+2}-2\times y_{i+1})^2-stack_i \le (curvature\_constr\_upper)^2$$

$$curvature\_constr\_upper=average\_delta\_s\times average\_delta\_s\times curvature\_constraint\_$$

该约束是之前提到的那个矢量的模的平方减去松弛变量。首先计算了原始的anchor points之间的平均距离average_delta_s，定义最大曲率curvature_constraint = 0.2，该约束的上界设置为$average_delta_s^2\times curvature_constraint$。该约束没有下界，而计算其上界的具体物理意义如下图所示，黄色线段可以理解为松弛变量。



如图所示，假设 $P0,P1,P2$三点处在同一个圆上，当$\theta1$较小时，向量$\overrightarrow{P1P0}$和 向量$\overrightarrow{P1P2}$的模近似等于弧长，因此有：

$\theta1=RdS$

根据$O-P0-P1$等腰三角形几何关系有：

$\theta2=2\pi-\theta1$

由于$|\overrightarrow{P1P0}|=|\overrightarrow{P1P2}|$，所以$C$是$P1P3$的中点，有$|\overrightarrow{P1C}|=|\overrightarrow{CP3}|$关系，由此得：

$|\overrightarrow{P1P3}|=2P1C$

根据$C-P0-P1$直角三角形几何关系有：

$P1C=dS\times cos(\theta2)$

把$P1C$带入$|\overrightarrow{P1P3}|$得：

$|\overrightarrow{P1P3}|=2P1C=2\times dS\times cos(\theta2)$

把$\theta2$带入得：

$|\overrightarrow{P1P3}|=2\times dS\times cos(\theta2)=2\times dS\times cos(2\pi-\theta1)=2\times dS\times cos(2\pi-2\theta1)=2\times dS\times sin(2\theta1)=2\times dS\times 2\theta1=dS\times\theta1$

把$\theta1$带入得：

$|\overrightarrow{P1P3}|=dS\times\theta1=dS\times RdS=dS2\times R1=dS2\times cur$

---

fem_pos_deviation_osqp_interface.cc和

fem_pos_deviation_sqp_osqp_interface.cc文件中二次规划代价函数矩阵形式如下：

```
void FemPosDeviationOsqpInterface::CalculateKernel(
    std::vector<c_float>* P_data, std::vector<c_int>* P_indices,
    std::vector<c_int>* P_indptr) {
  CHECK_GT(num_of_variables_, 4);
```

```
  std::vector<std::vector<std::pair<c_int, c_float>>> columns;
  columns.resize(num_of_variables_);
  int col_num = 0;

  for (int col = 0; col < 2; ++col) {
    columns[col].emplace_back(col, weight_fem_pos_deviation_ +
                                   weight_path_length_ +
                                   weight_ref_deviation_);
    ++col_num;
  }
```

```
for (int col = 2; col < 4; ++col) {
  columns[col].emplace_back(
      col - 2, -2.0 * weight_fem_pos_deviation_ - weight_path_length
  columns[col].emplace_back(col, 5.0 * weight_fem_pos_deviation_ +
                                     2.0 * weight_path_length_ +
                                     weight_ref_deviation_);
  ++col_num;
}

int second_point_from_last_index = num_of_points_ - 2;
for (int point_index = 2; point_index < second_point_from_last_index
     ++point_index) {
  int col_index = point_index * 2;
  for (int col = 0; col < 2; ++col) {
    col_index += col;
    columns[col_index].emplace_back(col_index - 4, weight_fem_pos_de
    columns[col_index].emplace_back(
        col_index - 2,
        -4.0 * weight_fem_pos_deviation_ - weight_path_length_);
    columns[col_index].emplace_back(
        col_index, 6.0 * weight_fem_pos_deviation_ +
                       2.0 * weight_path_length_ + weight_ref_deviat
    ++col_num;
  }
}

int second_point_col_from_last_col = num_of_variables_ - 4;
int last_point_col_from_last_col = num_of_variables_ - 2;
for (int col = second_point_col_from_last_col;
     col < last_point_col_from_last_col; ++col) {
  columns[col].emplace_back(col - 4, weight_fem_pos_deviation_);
  columns[col].emplace_back(
      col - 2, -4.0 * weight_fem_pos_deviation_ - weight_path_length
  columns[col].emplace_back(col, 5.0 * weight_fem_pos_deviation_ +
                                     2.0 * weight_path_length_ +
                                     weight_ref_deviation_);
  ++col_num;
```

```
  }

  for (int col = last_point_col_from_last_col; col < num_of_variables_
    columns[col].emplace_back(col - 4, weight_fem_pos_deviation_);
    columns[col].emplace_back(
        col - 2, -2.0 * weight_fem_pos_deviation_ - weight_path_length
    columns[col].emplace_back(col, weight_fem_pos_deviation_ +
                                       weight_path_length_ +
                                       weight_ref_deviation_);
    ++col_num;
  }

  CHECK_EQ(col_num, num_of_variables_);

  int ind_p = 0;
  for (int i = 0; i < col_num; ++i) {
    P_indptr->push_back(ind_p);
    for (const auto& row_data_pair : columns[i]) {


      P_data->push_back(row_data_pair.second * 2.0);
      P_indices->push_back(row_data_pair.first);
      ++ind_p;
    }
  }
  P_indptr->push_back(ind_p);
}
```

- 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
- 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
- 98

代价函数——二次规划代价函数矩阵形式

以n个点为例子$P_0,P_1,P_2,P_3,\cdots,P_n$

第1部分——参考点距离代价
$objvalue_1=\sum_{i=0}^{n-1}(x_i-x_{i-ref})^2+(y_i-y_{i-ref})^2$
第2部分——平滑性代价
$objvalue_2=\sum_{i=0}^{n-2}(x_i+x_{i+2}-2\times x_{i+1})^2+(y_i+y_{i+2}-2\times y_{i+1})^2$
第3部分——总长度代价
$objvalue_3=\sum_{i=0}^{n-1}(x_i-x_{i+1})^2+(y_i-y_{i+1})^2$

总代价函数：
$objvalue=w_1\times objvalue_1+w_2\times objvalue_2+w_3\times objvalue_3$
其中$w_1 w_2 w_3$分别代表上述三种代价的权重，经过化简得到矩阵形式：
$obsvalue=\frac{1}{2}x^TPx+q^Tx$
其中P和q矩阵形式如下：
$P=W2+W3+W1-2W2-W3W20000-2W2-W35W2+2W3+W1-4W2-W3\ddots.000W2-4W2-W36W2+2W3+W1\ddots.W2000W2-4W2-W3\ddots.-4W2-W3W2000W2\ddots.6W2+2W3+W1-4W2-W3W2000\ddots.-4W2$

$-W35W2+2W3+W1-2W2-W30000W2-2W2-W3W2+W3+W1$

$q=-2W1Xref$

约束方程矩阵形式：
$l\leq Ax\leq u$
A为单位矩阵$In\times n$，l和u为xy设置的上下界

purepursuit方法是基于几何追踪的路径追踪方法，基于几何的控制方法较为简单和直接，不用考虑车辆的运动学模型和动力学模型，控制时使用的参数少，能够较好的运用到实践使用中。最常用的两种方法是purepursuit方法和stanly方法。这里主要介绍purepursuit方法purepursuit建立在两个模型上，阿克曼转向几何模型和二维自行车模型。参数说明：δ：车辆的转向角；L:为车轴长度R：转弯半径K：是计算出来的圆弧的曲率ld:预瞄距离α：目标点方向与当前航向的夹角；(gx,gy)：目标点；根据阿克曼转向几何关系，可以建立车辆前轮转向角和后轮遵循的曲率之间的关系，轮的偏向角δ，与后轮划