CHENXXX - 数值优化第三章作业

1 完成情况

本次作业完成了task1中的推导（不确定是否需要写程序实现一下），task2中的代码补全和CDC问题，以及task3中的所有要求

2 作业1

2.1 推导

A **strictly convex QP** with only **equality constraints**:

$$\begin{aligned}&\min_{x\in\mathbb{R}^n}\frac{1}{2}x^\top Qx+c^\top x\\&s.t. Ax=b\end{aligned}$$

2.1.1 proof:

We can employ the KKT conditions to sovle above problem. First, calculate the derivative of the objective function and the equality constraints

$$\bigtriangledown f(x)=Qx+c,\bigtriangledown h(x)=A^\top$$

1. Stationarity: $Qx^*+c+A^\top v^*=0$
2. Primal feasibility: $Ax^*=b$
3. Dual feasibility: none
4. Complementary feasibility: none

Sinultaneous above conditions, we have

$$\begin{cases}Qx^*+c+A^\top v^*=0\\ Ax^*=b\end{cases}$$

Rewriting as a matrix

$$\begin{bmatrix}Q & A^\top\\ A & 0\end{bmatrix}\begin{bmatrix}x^*\\ v^*\end{bmatrix}=\begin{bmatrix}-c\\ b\end{bmatrix} \tag{1}$$

It should be note that Q is SPD matrix, the linear equation in (1) has an analytical solution via schur complement. The schur complement of Q can be written as

$$\begin{bmatrix}Q & A^\top\\ A & 0\end{bmatrix}=\begin{bmatrix}I & 0\\ AQ^{-1} & I\end{bmatrix}\begin{bmatrix}Q & 0\\ 0 & -AQ^{-1}A\end{bmatrix}\begin{bmatrix}I & Q^{-1}A^\top\\ 0 & I\end{bmatrix}$$

$$\begin{bmatrix}Q & A^\top\\ A & 0\end{bmatrix}^{-1}=\begin{bmatrix}I & -Q^{-1}A^\top\\ 0 & I\end{bmatrix}\begin{bmatrix}Q^{-1} & 0\\ 0 & -(AQ^{-1}A)^{-1}\end{bmatrix}\begin{bmatrix}I & 0\\ -AQ^{-1} & I\end{bmatrix}$$

where $AQ^{-1}A$ is PSD since Q is PSD matrix. Then, the analytical solution to the QP problem with only equality constraints can be written as

$$\begin{bmatrix}x^*\\ v^*\end{bmatrix}=\begin{bmatrix}I & -Q^{-1}A^\top\\ 0 & I\end{bmatrix}\begin{bmatrix}Q^{-1} & 0\\ 0 & -(AQ^{-1}A)^{-1}\end{bmatrix}\begin{bmatrix}I & 0\\ -AQ^{-1} & I\end{bmatrix}\begin{bmatrix}-c\\ b\end{bmatrix}$$

3 作业2

3.1 补全思路

按照课程ppt的思路，$u = g - ||g||e_i$。为了数值稳定性，选择投影方向为$-sgn(g_i)||g_i||e_i$，其中$i = \arg\max_k |g_k|$。所以原始公式变为$u = g + sgn(g_i)||g_i||e_i$。代码实现如下：

```
// 选择绝对值最大的元素 其index作为投影的方向
const int id = max_abs<d>(new_origin);
const double g_norm = std::sqrt(sqr_norm<d>(new_origin));
// u = g + sgn(g_i)*||g||*e_i
cpy<d>(new_origin, reflx);
if(new_origin[id] < 0.0){
  reflx[id] += -g_norm; // u = reflx
} else{
  reflx[id] += g_norm;
}
```

为了加快求解速度，下面推导直接求出投影后约束的公式。设不等式约束的个数为`n`，优化量的维度为d，$c = \frac{2}{\mathbf{u}^\top \mathbf{u}}$

$$\mathbf{u} = (u_1, u_2, u_3, \dots, u_d)^\top$$

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1d} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2d} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3d} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nd} \end{bmatrix}_{n\times d} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \mathbf{A}_3 \\ \vdots \\ \mathbf{A}_n \end{bmatrix}$$

$$\mathbf{H} = \mathbf{I}_d + c \begin{bmatrix} u_1u_1 & u_1u_2 & u_1u_3 & \dots & u_1u_d \\ u_2u_1 & u_2u_2 & u_2u_3 & \dots & u_2u_d \\ u_3u_1 & u_3u_2 & u_3u_3 & \dots & u_3u_d \\ \vdots & & & & \\ u_du_1 & u_du_2 & u_du_3 & \dots & u_du_d \end{bmatrix} = \mathbf{I}_d + c\mathbf{u}^\top\mathbf{u}$$

假设g中第二个元素的绝对值最大，那么矩阵$\mathbf{M}$为

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} + c \begin{bmatrix} u_1u_1 & u_3u_1 & \dots & u_du_1 \\ u_1u_2 & u_3u_2 & \dots & u_du_2 \\ u_1u_3 & u_3u_3 & \dots & u_du_3 \\ \vdots & \vdots & & \vdots \\ u_1u_d & u_3u_d & \dots & u_du_d \end{bmatrix}_{d\times(d-1)} = \mathbf{I}' + c[\mathbf{U}_1 \quad \mathbf{U}_3 \quad \dots \quad \mathbf{U}_d]$$

将之前的约束$\mathcal{I}$都进行投影有

$$
\begin{aligned}
\mathbf{A}' &= \mathbf{AM} \\
&= \begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1d} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2d} \\ \vdots & \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nd} \end{bmatrix} \left(\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix} + c \begin{bmatrix} u_1u_1 & u_3u_1 & \dots & u_du_1 \\ u_1u_2 & u_3u_2 & \dots & u_du_2 \\ u_1u_3 & u_3u_3 & \dots & u_du_3 \\ \vdots & \vdots & & \vdots \\ u_1u_d & u_3u_d & \dots & u_du_d \end{bmatrix} \right) \\
&= \begin{bmatrix} a_{11} & a_{13} & \dots & a_{1d} \\ a_{21} & a_{23} & \dots & a_{2d} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n3} & \dots & a_{nd} \end{bmatrix} + c \begin{bmatrix} a_{11}u_1u_1 + a_{12}u_1u_2 + a_{13}u_1u_3 + \cdots + a_{1d}u_1u_d & \dots \\ a_{21}u_1u_1 + a_{22}u_1u_2 + a_{23}u_1u_3 + \cdots + a_{2d}u_1u_d & \dots \\ \vdots & \\ a_{n1}u_1u_1 + a_{n2}u_1u_2 + a_{n3}u_1u_3 + \cdots + a_{nd}u_1u_d & \dots \end{bmatrix} \\
&= \begin{bmatrix} a_{11} & a_{13} & \dots & a_{1d} \\ a_{21} & a_{23} & \dots & a_{2d} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n3} & \dots & a_{nd} \end{bmatrix} + c \begin{bmatrix} \mathbf{A}_1\mathbf{U}_1 & \mathbf{A}_1\mathbf{U}_3 & \dots & \mathbf{A}_1\mathbf{U}_d \\ \mathbf{A}_2\mathbf{U}_1 & \mathbf{A}_2\mathbf{U}_3 & \dots & \mathbf{A}_2\mathbf{U}_d \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_n\mathbf{U}_n & \mathbf{A}_n\mathbf{U}_3 & \dots & \mathbf{A}_n\mathbf{U}_d \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{b}' &= \mathbf{b} - \mathbf{Au} \\
&= \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} - \begin{bmatrix} a_{11} & a_{13} & \dots & a_{1d} \\ a_{21} & a_{23} & \dots & a_{2d} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n3} & \dots & a_{nd} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_d \end{bmatrix} = \begin{bmatrix} b_1 - \mathbf{A}_1\mathbf{u} \\ b_2 - \mathbf{A}_2\mathbf{u} \\ \vdots \\ b_n - \mathbf{A}_n\mathbf{u} \end{bmatrix}
\end{aligned}
$$

以上推导代码实现如下：

```
const double c = -2.0 / sqr_norm<d>(reflx);
// 将先前的约束都投影到约束i平面上 所以遍历到i平面马上退出
for(int j=0; j!=i; j=next[j]){
   double* new_plane = new_halves + j*d;
   const double* old_plane = halves + j*(d+1);
   // 按ppt上的思路 投影后的约束变成AM和b-Av
   // 其中 M的列向量为H的(d-1)个行向量 为了加速 这里直接算出AM的结果
   const double cAiu = c * dot<d>(old_plane,reflx);
   for(int k=0; k<d; k++){
      if(k<id){
         new_plane[k] = old_plane[k] + reflx[k]*cAiu;
      }
      else if(k>id){
         new_plane[k-1] = old_plane[k] + reflx[k]*cAiu;
      }
   }
   // 正常是b`=b-Av 但new_plane[d-1]=-b` old_plane[d]=-b
   new_plane[d-1] = dot<d>(new_origin,old_plane) + old_plane[d];
}
```

3.1.1 实验结果

从输出结果来看，代码是填写正确的。**详细代码补全请看task_2_1文件夹。**

```
optimal sol: 4.11111 9.15556 4.50022
optimal obj: 201.14
cons precision: 1.77636e-15
```

3.2 Collision Distance Computation

使用c++中stl库生成随机的障碍物集合$(v_1, v_2, \ldots, v_n)^\top$，用户通过rviz给定机器人的位置$x_{robot}$。计算以下低维度带约束二次规划问题

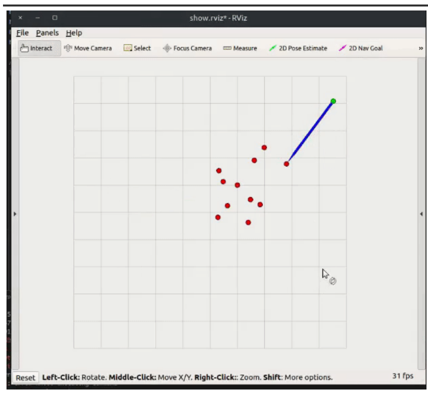
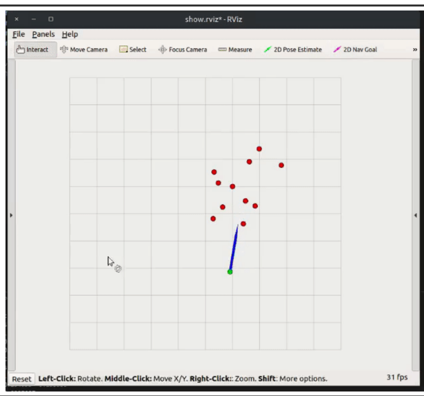
$$
\begin{aligned}
&\min_{z\in\mathbb{R}^2} y^\top y \\
&s.t. (x_{robot} - v_i)^\top z \le -1, \forall i \in \{1, 2, \ldots, n\}
\end{aligned}
$$

最后，碰撞向量$x = z/(z^\top z) + x_{robot}$。**详细代码请看task_2_2文件夹**，下面展示代码重点部分：

```cpp
int m = 2;
Eigen::Vector2d z = Eigen::Vector2d::Zero();
Eigen::Matrix2d Q = Eigen::Matrix2d::Identity();
Eigen::Vector2d c = Eigen::Vector2d::Zero();
Eigen::MatrixXd A;
A.resize(obs_lists.size(), 2);
for(size_t i=0; i<obs_lists.size(); i++){
  // obs_lists是随机生成的
  // robot_center是用户指定的
  A.row(i) = (robot_center - obs_lists.at(i)).transpose();
}
Eigen::VectorXd b = -1.0 * Eigen::VectorXd::Ones(obs_lists.size());
double cost = sdqp::sdqp<2>(Q, c, A, b, z);
```

3.2.1 实验结果

程序测试如下图所示。其中，红色表示障碍物的点，绿色表示机器人的中心位置，蓝色表示碰撞向量。**完整视频演示请看task_2.gif**

测试1（没碰撞）	测试2（没碰撞）	测试3（碰撞）
		

4 作业3

4.1 思路

汽车的非线性MPC算法可以建模成以下只有不等约束的非线性优化问题：

$$
\begin{aligned}
\min_{\mathbf{u}_{0:N-1}} & \sum_{k=0}^{N}(x_k - x_k^{ref})^2 + (y_k - y_k^{ref})^2 + w_\phi(\phi_k - \phi_k^{ref})^2 \\
s.t. & -0.1 \le v_k \le v_{max}, \forall k \in \{0,\dots,N\} \\
& -a_{max} \le a_k \le a_{max}, \forall k \in \{0,\dots,N-1\} \\
& -\delta_{max} \le \delta_k \le \delta_{max}, \forall k \in \{0,\dots,N-1\} \\
& -d\delta_{max} * dt \le d\delta_k - d\delta_{k-1} \le d\delta_{max} * dt, \forall k \in \{0,\dots,N-1\}
\end{aligned}
$$

为了输出更平滑连贯，可以增加输出平滑的惩罚项，故上述目标函数可以改为：

$$
\min_{\mathbf{u}_{0:N-1}} \sum_{k=0}^{N}(x_k - x_k^{ref})^2 + (y_k - y_k^{ref})^2 + w_\phi(\phi_k - \phi_k^{ref})^2 + w_a(a_k - a_{k-1})^2 + w_\delta(\delta_k - \delta_{k-1})^2
$$

其中，a_{-1}, δ_{-1}表示为上一次优化给执行器的输出。按照以下伪代码复现代码即可实现ALM算法：

Algorithm 1: PHR-ALM for NMPC

$\rho := 1, \boldsymbol{\mu} := \mathbf{0}, \gamma := 1, \beta := 1e^3, \xi := 0.1, found := false$;

while *not found* **do**

  Use L-BFGS solve $u := \arg\min_{\mathbf{u}} \mathcal{L}_\rho(\mathbf{u}, \boldsymbol{\mu})$;

  $\boldsymbol{\mu} := \max[\boldsymbol{\mu} + \rho g(\mathbf{u}), 0]$;

  $\rho := \min[(1+\gamma)\rho, \beta]$;

  $\xi := \min[\xi/10, 1e^{-5}]$;

  if $||\max[g(\mathbf{u}), -\boldsymbol{\mu}/\rho]||_\infty < \epsilon_{cons}$ *and* $||\nabla_{\mathbf{u}} \mathcal{L}_\rho(\mathbf{u}, \boldsymbol{\mu})||_\infty < \epsilon_{prec}$ **then**

    found := true ;

  else

    found := false ;

  end

end

4.2 代码实现

请查看mpc_car.hpp中**ReferenceCostGradient(), PHRObjeciveFunction(), SolvePHR()**函数

4.2.1 内环迭代优化

增广拉格朗日函数定义为：

$$
\mathcal{L}_\rho(\mathbf{u}, \mu) := f(\mathbf{u}) + \sum_{i=0}^{8N-2} \frac{\rho}{2} || \max[g(u_i) + \frac{\mu_i}{\rho}, 0]||^2
$$

当$J_p = \rho/2||\max[g(u_i) + \frac{\mu_i}{\rho}, 0]||^2 > 0$时，$u_i$对$J_p$的导数为：

$$\frac{\partial J_p}{\partial u_i} = \frac{\partial J_p}{\partial a}\frac{\partial a}{\partial u_i} = \rho a \frac{\partial g(u_i)}{\partial u_i}$$

其中，$a = g(\mu_i) + \mu_i/\rho$。这部分实现在**PHRObjeciveFunction()**函数中，部分代码为：

```
// 3. cost and gradient of inputs
for(int i=0; i<obj.N_; i++){
   // acc max
   double acc_max_bound = inputs.col(i)(0) - obj.a_max_ + obj.mu_(i,0)/obj.phr_rho_;
   if(std::max(acc_max_bound, 0.0) > 0.0){
    total_cost += 0.5 * obj.phr_rho_ * acc_max_bound * acc_max_bound;
    grad_inputs.col(i)(0) += obj.phr_rho_ * acc_max_bound;
   }
   // acc min
   int idx = obj.N_;
   double acc_min_bound = -inputs.col(i)(0) - obj.a_max_ + obj.mu_(i+idx,0)/obj.phr_rho_;
   if(std::max(acc_min_bound, 0.0) > 0.0){
    total_cost += 0.5 * obj.phr_rho_ * acc_min_bound * acc_min_bound;
    grad_inputs.col(i)(0) -= obj.phr_rho_ * acc_min_bound;
   }
   // delta max
   idx = 2*obj.N_;
   double delta_max_bound = inputs.col(i)(1) - obj.delta_max_ + obj.mu_(i+idx,0)/obj.phr_rho_;
   if(std::max(delta_max_bound, 0.0) > 0.0){
    total_cost += 0.5 * obj.phr_rho_ * delta_max_bound * delta_max_bound;
    grad_inputs.col(i)(1) += obj.phr_rho_ * delta_max_bound;
   }
   // delta min
   idx = 3*obj.N_;
   double delta_min_bound = -inputs.col(i)(1) - obj.delta_max_ + obj.mu_(i+idx,0)/obj.phr_rho_;
   if(std::max(delta_min_bound, 0.0) > 0.0){
    total_cost += 0.5 * obj.phr_rho_ * delta_min_bound * delta_min_bound;
    grad_inputs.col(i)(1) -= obj.phr_rho_ * delta_min_bound;
   }

   if(i < obj.N_ - 1){
    // ddelta max
    idx = 6*obj.N_;
    double ddelta_max_bound
     = inputs.col(i+1)(1) - inputs.col(i)(1) - obj.ddelta_max_*obj.dt_
     + obj.mu_(i+idx,0)/obj.phr_rho_;
    if(std::max(ddelta_max_bound, 0.0) > 0.0){
     total_cost += 0.5 * obj.phr_rho_ * ddelta_max_bound * ddelta_max_bound;
     grad_inputs.col(i)(1) -= obj.phr_rho_ * ddelta_max_bound;
     grad_inputs.col(i+1)(1) += obj.phr_rho_ * ddelta_max_bound;
    }
    // ddelta min
```

```
    idx = 7*obj.N_ - 1;
    double ddelta_min_bound
     = inputs.col(i)(1) - inputs.col(i+1)(1) - obj.ddelta_max_*obj.dt_
     + obj.mu_(i+idx,0)/obj.phr_rho_;
    if(std::max(ddelta_min_bound, 0.0) > 0.0){
     total_cost += 0.5 * obj.phr_rho_ * ddelta_min_bound * ddelta_min_bound;
     grad_inputs.col(i)(1) += obj.phr_rho_ * ddelta_min_bound;
     grad_inputs.col(i+1)(1) -= obj.phr_rho_ * ddelta_min_bound;
    }
   }
  }
```

4.2.2 参数更新

对应的公式为：

$$\mu := \max[\mu + \rho g(\mathbf{u}), 0] \; ;$$
$$\rho := \min[(1+\gamma)\rho, \beta] \; ;$$
$$\xi := \min[\xi/10, 1e^{-5}] \; ;$$

代码实现在**SolvePHR()**函数中，部分代码为：

```
// update mu
for(int i=0; i<N_; i++){
  // acc max
  double acc_max_bound = mu_(i, 0) + phr_rho_*(inputs.col(i)(0)-a_max_);
  mu_(i,0) = std::max(acc_max_bound, 0.0);
  // acc min
  int idx = N_;
  double acc_min_bound = mu_(i+idx, 0) + phr_rho_*(-inputs.col(i)(0)-a_max_);
  mu_(i+idx, 0) = std::max(acc_min_bound, 0.0);
  // delta max
  idx = 2*N_;
  double delta_max_bound = mu_(i+idx, 0) + phr_rho_*(inputs.col(i)(1)-delta_max_);
  mu_(i+idx, 0) = std::max(delta_max_bound, 0.0);
  // delta min
  idx = 3*N_;
  double delta_min_bound = mu_(i+idx, 0) + phr_rho_*(-inputs.col(i)(1)-delta_max_);
  mu_(i+idx, 0) = std::max(delta_min_bound, 0.0);
  // vel max
  idx = 4*N_;
  double vel_max_bound = mu_(i+idx, 0) + phr_rho_*(predictState_[i](3,0)-v_max_);
  mu_(i+idx, 0) = std::max(vel_max_bound, 0.0);
  // vel min
  idx = 5*N_;
```

```
    double vel_min_bound = mu_(i+idx, 0) + phr_rho_*(-predictState_[i](3,0)-0.1);
    mu_(i+idx, 0) = std::max(vel_min_bound, 0.0);

    if(i < N_-1){
      // ddelta max
      idx = 6*N_;
      double ddelta_max_bound
        = mu_(i+idx, 0) + phr_rho_*(inputs.col(i+1)(1)-inputs.col(i)(1)-ddelta_max_*dt_);
      mu_(i+idx, 0) = std::max(ddelta_max_bound, 0.0);

      // ddelta min
      idx = 7*N_ - 1;
      double ddelta_min_bound
        = mu_(i+idx, 0) + phr_rho_*(inputs.col(i)(1)-inputs.col(i+1)(1)-ddelta_max_*dt_);
      mu_(i+idx, 0) = std::max(ddelta_min_bound, 0.0);
    }
  }
  // update rho
  phr_rho_ = std::min((1+phr_gamma_)*phr_rho_, phr_beta_);
  // update xi
  phr_xi_ = phr_xi_ * 0.1;
  if(phr_xi_ < 1e-5){
    phr_xi_ = 1e-5;
  }
```

4.2.3 迭代收敛判断

对应公式为：

$$
\begin{aligned}
&\textbf{if } \|\max[g(\mathbf{u}), -\mu/\rho]\|_\infty < \epsilon_{cons} \textit{ and } \|\nabla_{\mathbf{u}} \mathcal{L}_\rho(\mathbf{u}, \mu)\|_\infty < \epsilon_{prec} \textbf{ then}\\
&\quad \text{found} := \text{true};\\
&\textbf{else}\\
&\quad \text{found} := \text{false};\\
&\textbf{end}
\end{aligned}
$$

代码实现在**SolvePHR()**函数中，部分代码为：

```
  // stop criterion
  kkt_1 = 0.0;
  for(int i=0; i<N_; i++){
    // acc max
    double acc_max = abs(std::max((inputs.col(i)(0)-a_max_), -mu_(i, 0)/phr_rho_));
    if(acc_max > kkt_1){
      kkt_1 = acc_max;
```

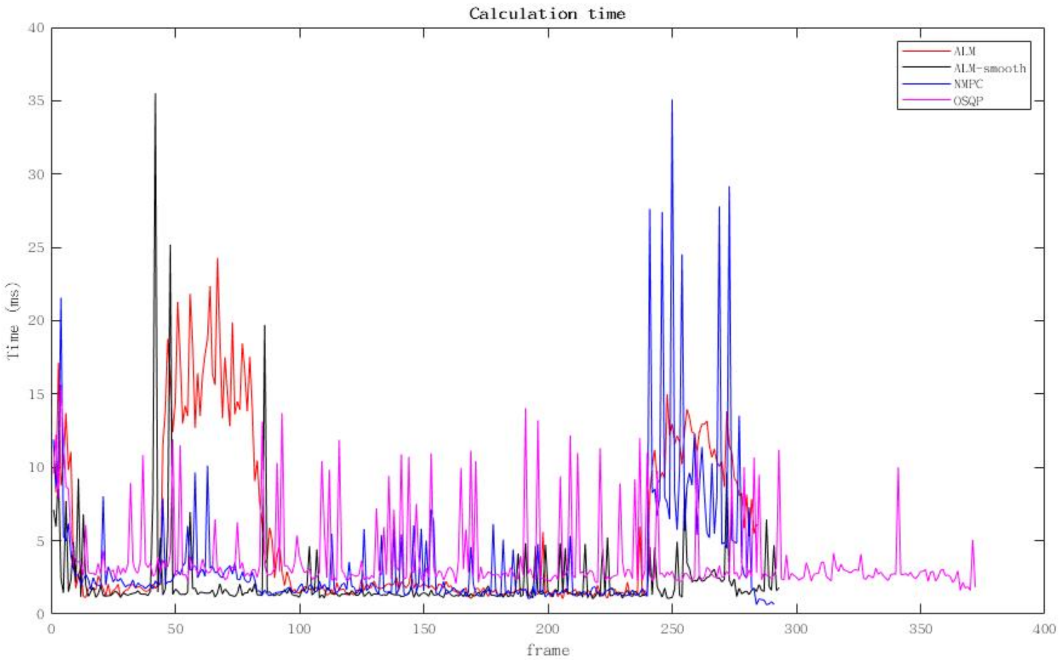
```
    }
    // acc min
    int idx = N_;
    double acc_min = abs(std::max((-inputs.col(i)(0)-a_max_), -mu_(i+idx, 0)/phr_rho_));
    if(acc_min > kkt_1){
      kkt_1 = acc_min;
    }
    // delta max
    idx = 2*N_;
    double delta_max = abs(std::max((inputs.col(i)(1)-delta_max_), -mu_(i+idx, 0)/phr_rho_));
    if(delta_max > kkt_1){
      kkt_1 = delta_max;
    }
    // delta min
    idx = 3*N_;
    double delta_min = abs(std::max((-inputs.col(i)(1)-delta_max_), -mu_(i+idx, 0)/phr_rho_));
    if(delta_min > kkt_1){
      kkt_1 = delta_min;
    }
    // vel max
    idx = 4*N_;
    double vel_max = abs(std::max((predictState_[i](3,0)-v_max_), -mu_(i+idx, 0)/phr_rho_));
    if(vel_max > kkt_1){
      kkt_1 = vel_max;
    }
    // vel min
    idx = 5*N_;
    double vel_min = abs(std::max((-predictState_[i](3,0)-0.1), -mu_(i+idx, 0)/phr_rho_));
    if(vel_min > kkt_1){
      kkt_1 = vel_min;
    }

    if(i<N_-1){
      idx = 6*N_;
      double ddelta_max = abs(
        std::max((inputs.col(i+1)(1)-inputs.col(i)(1)-ddelta_max_*dt_), -mu_(i+idx, 0)/phr_rho_));
      if(ddelta_max > kkt_1){
        kkt_1 = ddelta_max;
      }

      idx = 7*N_-1;
      double ddelta_min = abs(
        std::max((inputs.col(i)(1)-inputs.col(i+1)(1)-ddelta_max_*dt_), -mu_(i+idx, 0)/phr_rho_));
      if(ddelta_min > kkt_1){
        kkt_1 = ddelta_min;
      }
    }
  }
```

```
if(kkt_1 < 1e-5 && gradient_norm < 1e-4){
    found = true;
}
```

4.3 实验结果

4.3.1 耗时对比

详细代码请看task_3文件夹。不同算法耗时对比如下图所示。加入输出平滑惩罚项的ALM算法记为ALM-smooth。可以看到，加了输入平滑后能提升算法的速度

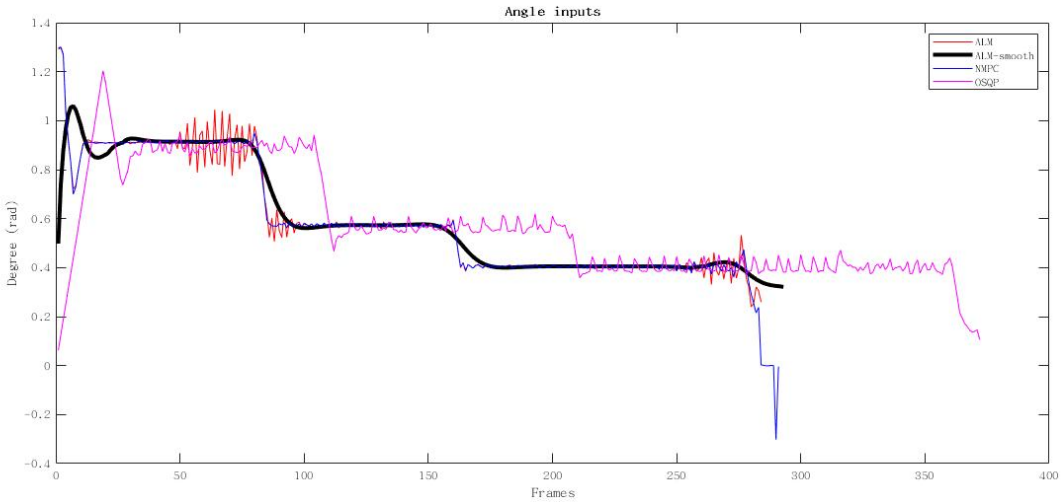


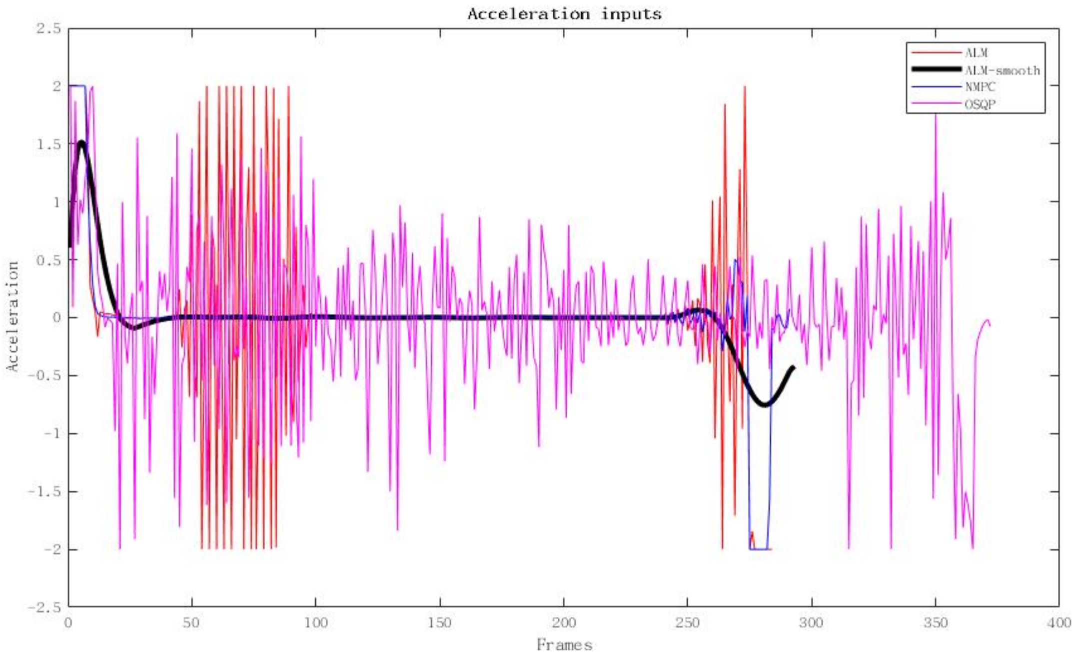
平均耗时如下表所示：

	ALM	ALM-smooth	NMPC	OSQP
Average time (ms)	5.405	2.194	3.521	3.8

4.3.2 输出曲线对比

不同算法产生的输出曲线如下图所示。从图中可以看出，**加入平滑惩罚项的ALM-smooth算法输出明显更平滑**。同时，ALM和OSQP算法的输出曲线都非常不稳定，有较大的波动


Angle inputs
ALM
ALM-smooth
NMPC
OSQP
Degree (rad)
Frames
1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
0
50
100
150
200
250
300
350
400


Acceleration inputs
ALM
ALM-smooth
NMPC
OSQP
Acceleration
Frames
2.5
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
-2.5
0
50
100
150
200
250
300
350
400