HW5

吴程锴 ckwu1201@163.com

一、MINIMUM SNAP

MINIMUM SNAP 是一种轨迹优化的方法，给定路标点和每段轨迹的时间，使用多项式曲线来表示轨迹，得到每段轨迹的多项式系数。从而得到光滑的轨迹提供给无人机执行。

该方法要求加加速度的平方和（推力的变化）最小，并且轨迹过路标点，且路标点处两段轨迹的位置、速度、加速度和加加速度连续。

二、使用 OOQP 求解器

OOQP 库主要是用来求解形如

$$\min \ \frac{1}{2}x^TQx+c^Tx$$

$$\begin{cases} Ax=b \\ d\leqslant Cx\leqslant f \\ l\ll x\ll u \end{cases}$$

的凸优化问题。

2.1 使用 OOQP 求解器的步骤

2.1.1 把 OOQP 头文件链接到可执行文件

```
target_link_libraries(trajectory_generator_node
    ${catkin_LIBRARIES}
    ooqpgensparse
    ooqpsparse
    ooqpgondzio
    ooqpbase blas ma27 gfortran
)
```

2.1.2 将Q、A等矩阵转化为 OOQP 的调用格式

这里我用的是稀疏矩阵的形式。

2.1.3 调用 OOQP 函数求得最优解

```
QpGenSparseMa27 * qp = new QpGenSparseMa27( nx, my, mz, nnzQ, nnzA, nnzC );
QpGenData      * prob = (QpGenData * ) qp->copyDataFromSparseTriple(
        c,      irowQ,  nnzQ,   jcolQ,  dQ,
        xlow,   ixlow,  xupp,   ixupp,
        irowA,  nnzA,   jcolA,  dA,     b,
        irowC,  nnzC,   jcolC,  dC,
        clow,   iclow,  cupp,   icupp );

QpGenVars      * vars = (QpGenVars *) qp->makeVariables( prob );
QpGenResiduals * resid = (QpGenResiduals *) qp->makeResiduals( prob );

GondzioSolver  * s     = new GondzioSolver( qp, prob );
int ierr = s->solve(prob,vars, resid);
```

2.1.4 将求得的解转化为可视化函数的调用格式

```
double result[nx] = {0};
vars->x->copyIntoArray(result);
// double a = vars->x->copyIntoArray
MatrixXd PolyCoeff_dim = MatrixXd::Zero(m, p_num1d);

for (int i = 0; i < m;i++){
    for (int j = 0; j < p_num1d;j++){
        PolyCoeff_dim(i, j) = result[i * p_num1d + j];
    }
}
PolyCoeff.block(0, dim * p_num1d, m, p_num1d) = PolyCoeff_dim;

```

2.2 Q 矩阵

最小化 jerk 的平方和，将其写成二次型的形式$J_j = p_j^T Q_j p_j$的形式，其中Q_j为第 j 段轨迹的 jerk 系数。使用 block 对 Q 矩阵按不同轨迹段进行分块赋值。

```
MatrixXd Q = MatrixXd::Zero(p_num1d * m, p_num1d * m);
MatrixXd Qi;
for (int k = 0; k < m; k++)
{
    Qi = MatrixXd::Zero(p_num1d, p_num1d);
    for (int i = 3; i < p_num1d; i++){
```

```
        for (int l = 3; l < p_num1d ; l++){
            Qi(i, l) = (i + 1) * i * (i - 1) * (i - 2) * (l + 1) * l * (l - 1) * (l - 2) / (i + l + 2 - 7) * pow(Time(k), i + l + 2 - 7);
        }
    }
    Q.block(k * p_num1d, k * p_num1d, p_num1d, p_num1d) = Qi;
}
```

2.3 首尾、路标点等式约束

$$A_j p_j = d_j$$

$$\Rightarrow \left[\cdots \frac{i!}{(i-k)} T_j^{i-k} \cdots \right] \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \end{bmatrix} = x_{T,j}^{(k)}$$

2.3.1 起始点 p、v、a、j

```
MatrixXd Aeq_start = MatrixXd::Zero(4, p_num1d * m);
MatrixXd beq_start = MatrixXd::Zero(4, 1);
Aeq_start(0, 0) = 1;//p
Aeq_start(1, 1) = 1;//v
Aeq_start(2, 2) = 2;//a
Aeq_start(3, 3) = 6;//j
beq_start(0) = Path(0, dim);
beq_start(1) = Vel(0, dim);
beq_start(2) = Acc(0, dim);
beq_start(3) = 0;
```

2.3.2 结束点 p、v、a、j

```
MatrixXd Aeq_end = MatrixXd::Zero(4, p_num1d * m);
MatrixXd beq_end = MatrixXd::Zero(4, 1);
//p
// cout << "***************"<< endl;
for (int i = 0; i < p_num1d;i++){
    Aeq_end(0, p_num1d * m - p_num1d + i) = pow(Time(m-1), i);
    //cout << "***************"<< endl;
}
//cout << "***************"<< endl;
//v
for (int i = 1; i < p_num1d;i++){
    Aeq_end(1, p_num1d * m - p_num1d + i) = i * pow(Time(m-1), i - 1);
}
```

```
//a
for (int i = 2; i < p_num1d;i++){
    Aeq_end(2, p_num1d * m - p_num1d + i) = i*(i-1)*pow(Time(m-1), i-2);
}
//j
for (int i = 3; i < p_num1d;i++){
    Aeq_end(3, p_num1d * m - p_num1d + i) = i * (i - 1) * (i - 1) * pow(Time(m-1), i - 3);
}
beq_end(0) = Path(Path.rows() - 1, dim);
beq_end(1) = Vel(1, dim);
beq_end(2) = Acc(1, dim);
beq_end(3) = 0;
```

2.3.3 路标点

```
//position constrain
MatrixXd Aeq_wp = MatrixXd::Zero(m-1, p_num1d * m);
MatrixXd beq_wp = MatrixXd::Zero(m-1, 1);
for (int i = 0; i < m - 1; i++){
    for (int j = 0; j < p_num1d; j++){
        Aeq_wp(i, i * p_num1d + j) = pow(Time(i), j);
    }
    beq_wp(i) = Path(i+1, dim);
}
```

2.4 连续性等式约束

$$[A_j - A_{j-1}]\begin{bmatrix} p_j \\ p_{j+1} \end{bmatrix} = 0$$

$$\Rightarrow \begin{bmatrix} \cdots & \frac{i!}{(i-k)}T_j^{i-k} & \cdots & -\frac{l!}{(l-k)}T_j^{l-k} & \cdots \end{bmatrix} \begin{bmatrix} \vdots \\ p_{j,i} \\ \vdots \\ p_{j+1,i} \\ \vdots \end{bmatrix} = 0$$

2.4.1 位置连续性

```
//position continuity constrain
MatrixXd Aeq_con_p = MatrixXd::Zero(m-1, p_num1d * m);
MatrixXd beq_con_p = MatrixXd::Zero(m-1, 1);
for (int i = 0; i < m - 1; i++){
    for (int j = 0; j < p_num1d; j++){
        Aeq_con_p(i, i * p_num1d + j) = pow(Time(i), j);
```

```
    }
    Aeq_con_p(i, (i + 1) * p_num1d) = -1;
}
```

2.4.2 速度连续性

```
//      << Aeq_con_p << endl;
MatrixXd Aeq_con_v = MatrixXd::Zero(m-1, p_num1d * m);
MatrixXd beq_con_v = MatrixXd::Zero(m-1, 1);
for (int i = 0; i < m - 1; i++){
    for (int j = 1; j < p_num1d; j++){
        Aeq_con_v(i, i * p_num1d + j) = j * pow(Time(i), j - 1);
    }
    //cout << "(i + 1) * p_num1d + 1:" << (i + 1) * p_num1d + 1 << endl;
    Aeq_con_v(i, (i + 1) * p_num1d + 1) = -1;
}
```

2.4.3 加速度连续性

```
//acceleration continuity constrain
MatrixXd Aeq_con_a = MatrixXd::Zero(m-1, p_num1d * m);
MatrixXd beq_con_a = MatrixXd::Zero(m-1, 1);
for (int i = 0; i < m-1;i++){
    for (int j = 2; j < p_num1d;j++){
        Aeq_con_a(i, i * p_num1d + j) =j*(j-1)*pow(Time(i), j-2);
    }
    Aeq_con_a(i, (i + 1) * p_num1d + 2) = -2;
}
```

2.4.4 Jerk 连续性

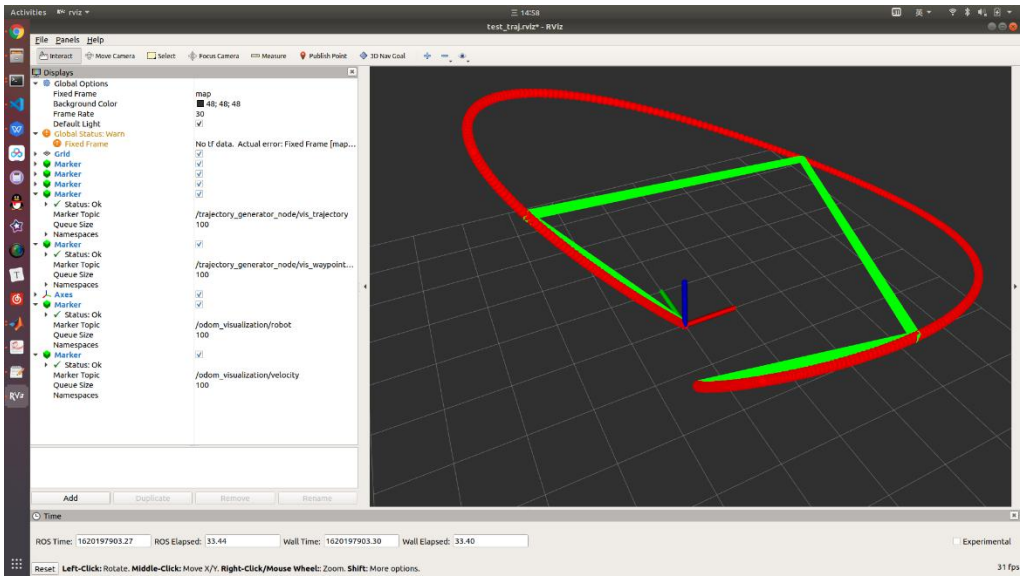
```
//jerk continuity constrain
MatrixXd Aeq_con_j = MatrixXd::Zero(m-1, p_num1d * m);
MatrixXd beq_con_j = MatrixXd::Zero(m-1, 1);
for (int i = 0; i < m-1;i++){
    for (int j = 3; j < p_num1d;j++){
        Aeq_con_j(i, i * p_num1d + j) =j*(j-1)*(j-2)*pow(Time(i), j-3);
    }
    Aeq_con_j(i, (i + 1) * p_num1d + 3) = -6;
}
```

2.4.5 最后将所有矩阵合并

```
MatrixXd Aeq = MatrixXd::Zero(5 * (m - 1) + 2 * 4, p_num1d * m);
MatrixXd beq = MatrixXd::Zero(5 * (m - 1) + 2 * 4, 1);
Aeq.block(0, 0, 4, p_num1d * m) = Aeq_start;
Aeq.block(4, 0, 4, p_num1d * m) = Aeq_end;
Aeq.block(8, 0, m - 1, p_num1d * m) = Aeq_wp;
Aeq.block(8 + 1 * (m - 1), 0, m - 1, p_num1d * m) = Aeq_con_p;
Aeq.block(8 + 2 * (m - 1), 0, m - 1, p_num1d * m) = Aeq_con_v;
Aeq.block(8 + 3 * (m - 1), 0, m - 1, p_num1d * m) = Aeq_con_a;
Aeq.block(8 + 4 * (m - 1), 0, m - 1, p_num1d * m) = Aeq_con_j;

beq.block(0, 0, 4, 1) = beq_start;
beq.block(4, 0, 4, 1) = beq_end;
beq.block(8, 0, m - 1, 1) = beq_wp;
beq.block(8 + 1 * (m - 1), 0, m - 1, 1) = beq_con_p;
beq.block(8 + 2 * (m - 1), 0, m - 1, 1) = beq_con_v;
beq.block(8 + 3 * (m - 1), 0, m - 1, 1) = beq_con_a;
beq.block(8 + 4 * (m - 1), 0, m - 1, 1) = beq_con_j;
```

2.5 结果



三、使用 Eigen 生成基于闭式解的最小加加加速度的轨迹

Q 矩阵与 2.1 中一致。把求解多项式系数转换为求解到达各路标处的速度和加速度来避免因为时间过大产生的奇异性。

3.1 映射矩阵 M

把系数映射为首末时刻的位置、速度、加速度和加加速度

$$M_j p_j = d_j$$

```
MatrixXd M = MatrixXd::Zero(p_num1d * m, p_num1d * m);
MatrixXd Mi;

for (int k = 0; k < m; k++)
{
    Mi = MatrixXd::Zero(p_num1d, p_num1d);
    //time=0
    Mi(0, 0) = 1;
    Mi(1, 1) = 1;
    Mi(2, 2) = 2;
    Mi(3, 3) = 6;
    //p
    for (int i = 0; i < p_num1d;i++){
        Mi(4, i) = pow(Time(k), i);
    }
    //v
    for (int i = 1; i < p_num1d;i++){
        Mi(5, i) = i * pow(Time(k), i - 1);
    }
    //a
    for (int i = 2; i < p_num1d;i++){
        Mi(6, i) = i * (i - 1) * pow(Time(k), i - 2);
    }
    //j
    for (int i = 3; i < p_num1d;i++){
        Mi(7, i) = i * (i - 1) * (i - 2) * pow(Time(k), i - 3);
    }
    M.block(k * p_num1d, k * p_num1d, p_num1d, p_num1d) = Mi;
}
```

3.2 选择矩阵 C

将已知的变量映射为d_F，未知的每段轨迹连接处的速度、加速度和加加速度映射为d_P。

$$C^T \begin{bmatrix} d_F \\ d_P \end{bmatrix} = \begin{bmatrix} d_1 \\ \vdots \\ d_M \end{bmatrix}$$

```
MatrixXd Ct = MatrixXd::Zero(2 * d_order * m, d_order * (m + 1));
Ct(0, 0) = 1;
Ct(1, 1) = 1;
Ct(2, 2) = 1;
Ct(3, 3) = 1;

Ct(2 * d_order * m - 4, d_order + m - 1) = 1;
Ct(2 * d_order * m - 3, d_order + m + 0) = 1;
Ct(2 * d_order * m - 2, d_order + m + 1) = 1;
Ct(2 * d_order * m - 1, d_order + m + 2) = 1;

for (int i = 0; i < m-1; i++){

    Ct(d_order+2*i*d_order,d_order+i) = 1;
    Ct(d_order+2*i*d_order+d_order,d_order+i) = 1;

    Ct(d_order + 2 * i * d_order + 1, 2 * d_order + m + i * (d_order - 1) - 1) = 1;// v_end
    Ct(d_order+2*i*d_order+2,2*d_order+m+i*(d_order-1)+0)=1;// a_end
    Ct(d_order+2*i*d_order+3,2*d_order+m+i*(d_order-1)+1)=1;// j_end

    Ct(d_order+2*i*d_order+1+d_order,2*d_order+m+i*(d_order-1)-1)=1;// v_start
    Ct(d_order+2*i*d_order+2+d_order,2*d_order+m+i*(d_order-1)+0)=1;// a_start
    Ct(d_order+2*i*d_order+3+d_order,2*d_order+m+i*(d_order-1)+1)=1;// j_start
}
```

3.3 得到多项式系数

最终将带有等式约束的 QP 问题转化为无约束的 QP 问题。接下来对代价函数

$$J = d_F^T R_{FF} d_F + d_F^T R_{FP} d_P + d_P^T R_{PF} d_F + d_P^T R_{PP} d_P$$

求导得到最优的$d_p^* = -R_{PP}^{-1} R_{FP}^T d_F$，再代入公式

$$Mp = d = C^T \begin{bmatrix} d_F \\ d_P \end{bmatrix} \Rightarrow p = M^{-1} C^T \begin{bmatrix} d_F \\ d_P \end{bmatrix}$$

得到每段轨迹对应的多项式系数。

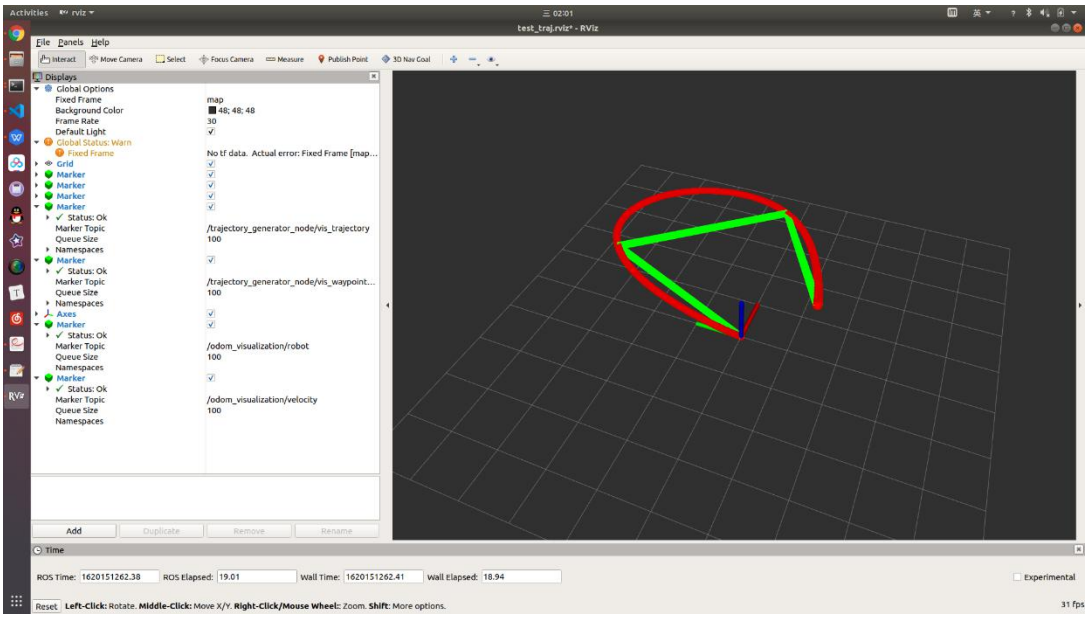
```
auto R = C * M.inverse().transpose() * Q * M.inverse() * Ct;

int Fsize = Path.size() / 3 - 2 + 2 * 4;
int Psize = R.rows() - Fsize;
auto R_pp = R.block(Fsize, Fsize, Psize, Psize);
auto R_fp = R.block(0, Fsize, Fsize, Psize);
cout << "Path:" << endl
```

```
        << Path << endl;
for (int i = 0; i < 3;i++){
    Vector4d start_cond(Path(0, i),Vel(0,i),Acc(1,i),0), end_cond(Path(Path.size() / 3-1, i),Vel(1,i),Acc(1,i),0);
    VectorXd dF(Fsize);
    dF << start_cond, Path.block(1,i,Path.size() / 3 - 2, 1), end_cond;
    auto dP = -R_pp.inverse() * R_fp.transpose() * dF;
    VectorXd d(Fsize + Psize);
    d << dF, dP;

    MatrixXd P1 = M.inverse() * Ct * d;
    P1 = P1.transpose();
    MatrixXd P(m, p_num1d);
    for (int j = 0; j < m; j++)
    {
        P.block(j,0,1,p_num1d) = P1.block(0, j * p_num1d, 1, p_num1d);
    }
    PolyCoeff.block(0, i * p_num1d, m, p_num1d) = P;
}
```

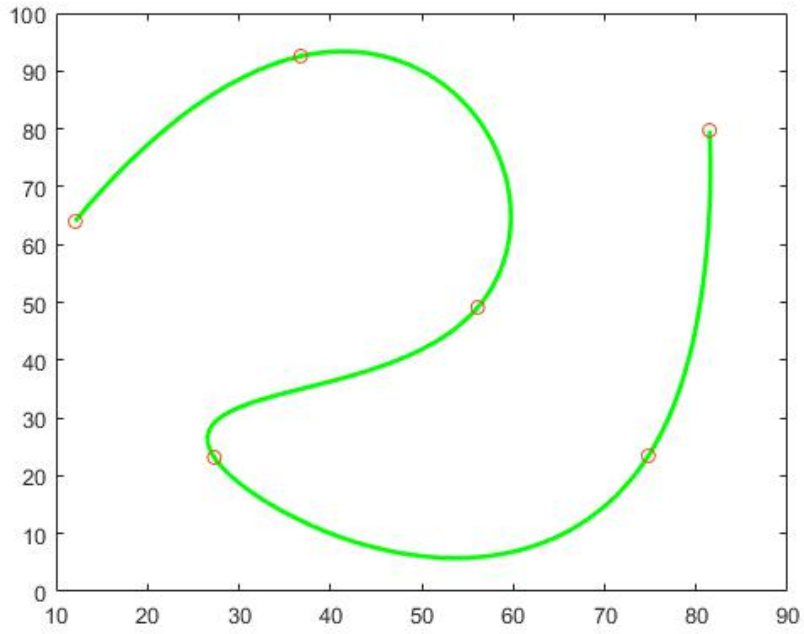
3.4 结果



四、Matlab 实现

MATLAB 的实现也是一样的原理，这里就直接放上结果

4.1 使用 QP 求解器的结果



4.2 闭式解结果