HW5

吴程锴 ckwu1201@163.com

# 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。

# 使用OOQP求解器

OOQP库主要是用来求解形如



的凸优化问题。

## 使用OOQP求解器的步骤

### 把OOQP头文件链接到可执行文件

1. target\_link\_libraries(trajectory\_generator\_node
2. ${catkin\_LIBRARIES}
3. ooqpgensparse
4. ooqpsparse
5. ooqpgondzio
6. ooqpbase blas ma27 gfortran
7. )

### 将、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。

### 调用OOQP函数求得最优解

1. QpGenSparseMa27 \* qp = **new** QpGenSparseMa27( nx, my, mz, nnzQ, nnzA, nnzC );
2. QpGenData      \* prob = (QpGenData \* ) qp->copyDataFromSparseTriple(
3. c,      irowQ,  nnzQ,   jcolQ,  dQ,
4. xlow,   ixlow,  xupp,   ixupp,
5. irowA,  nnzA,   jcolA,  dA,     b,
6. irowC,  nnzC,   jcolC,  dC,
7. clow,   iclow,  cupp,   icupp );
9. QpGenVars      \* vars = (QpGenVars \*) qp->makeVariables( prob );
10. QpGenResiduals \* resid = (QpGenResiduals \*) qp->makeResiduals( prob );
12. GondzioSolver  \* s     = **new** GondzioSolver( qp, prob );
13. **int** ierr = s->solve(prob,vars, resid);

### 将求得的解转化为可视化函数的调用格式

1. **double** result[nx] = {0};
2. vars->x->copyIntoArray(result);
3. // double a = vars->x->copyIntoArray
4. MatrixXd PolyCoeff\_dim = MatrixXd::Zero(m, p\_num1d);
6. **for** (**int** i = 0; i < m;i++){
7. **for** (**int** j = 0; j < p\_num1d;j++){
8. PolyCoeff\_dim(i, j) = result[i \* p\_num1d + j];
9. }
10. }
11. PolyCoeff.block(0, dim \* p\_num1d, m, p\_num1d) = PolyCoeff\_dim;

## Q矩阵

最小化jerk的平方和，将其写成二次型的形式的形式，其中为第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。

1. MatrixXd Q = MatrixXd::Zero(p\_num1d \* m, p\_num1d \* m);
2. MatrixXd Qi;
3. **for** (**int** k = 0; k < m; k++)
4. {
5. Qi = MatrixXd::Zero(p\_num1d, p\_num1d);
6. **for** (**int** i = 3; i < p\_num1d; i++){
7. **for** (**int** l = 3; l < p\_num1d ; l++){
8. 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);
9. }
10. }
11. Q.block(k \* p\_num1d, k \* p\_num1d, p\_num1d, p\_num1d) = Qi;
12. }

## 首尾、路标点等式约束



### 起始点p、v、a、j

1. MatrixXd Aeq\_start = MatrixXd::Zero(4, p\_num1d \* m);
2. MatrixXd beq\_start = MatrixXd::Zero(4, 1);
3. Aeq\_start(0, 0) = 1;//p
4. Aeq\_start(1, 1) = 1;//v
5. Aeq\_start(2, 2) = 2;//a
6. Aeq\_start(3, 3) = 6;//j
7. beq\_start(0) = Path(0, dim);
8. beq\_start(1) = Vel(0, dim);
9. beq\_start(2) = Acc(0, dim);
10. beq\_start(3) = 0;

### 结束点p、v、a、j

1. MatrixXd Aeq\_end = MatrixXd::Zero(4, p\_num1d \* m);
2. MatrixXd beq\_end = MatrixXd::Zero(4, 1);
3. //p
4. // cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;
5. **for** (**int** i = 0; i < p\_num1d;i++){
6. Aeq\_end(0, p\_num1d \* m - p\_num1d + i) = pow(Time(m-1), i);
7. //cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;
8. }
9. //cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"<< endl;
10. //v
11. **for** (**int** i = 1; i < p\_num1d;i++){
12. Aeq\_end(1, p\_num1d \* m - p\_num1d + i) = i \* pow(Time(m-1), i - 1);
13. }
14. //a
15. **for** (**int** i = 2; i < p\_num1d;i++){
16. Aeq\_end(2, p\_num1d \* m - p\_num1d + i) = i\*(i-1)\*pow(Time(m-1), i-2);
17. }
18. //j
19. **for** (**int** i = 3; i < p\_num1d;i++){
20. Aeq\_end(3, p\_num1d \* m - p\_num1d + i) = i \* (i - 1) \* (i - 1) \* pow(Time(m-1), i - 3);
21. }
22. beq\_end(0) = Path(Path.rows() - 1, dim);
23. beq\_end(1) = Vel(1, dim);
24. beq\_end(2) = Acc(1, dim);
25. beq\_end(3) = 0;

### 路标点

1. //position constrain
2. MatrixXd Aeq\_wp = MatrixXd::Zero(m-1, p\_num1d \* m);
3. MatrixXd beq\_wp = MatrixXd::Zero(m-1, 1);
4. **for** (**int** i = 0; i < m - 1; i++){
5. **for** (**int** j = 0; j < p\_num1d; j++){
6. Aeq\_wp(i, i \* p\_num1d + j) = pow(Time(i), j);
7. }
8. beq\_wp(i) = Path(i+1, dim);
9. }

## 连续性等式约束



### 位置连续性

1. //position continuity constrain
2. MatrixXd Aeq\_con\_p = MatrixXd::Zero(m-1, p\_num1d \* m);
3. MatrixXd beq\_con\_p = MatrixXd::Zero(m-1, 1);
4. **for** (**int** i = 0; i < m - 1; i++){
5. **for** (**int** j = 0; j < p\_num1d; j++){
6. Aeq\_con\_p(i, i \* p\_num1d + j) = pow(Time(i), j);
7. }
8. Aeq\_con\_p(i, (i + 1) \* p\_num1d) = -1;
9. }

### 速度连续性

1. //      << Aeq\_con\_p << endl;
2. MatrixXd Aeq\_con\_v = MatrixXd::Zero(m-1, p\_num1d \* m);
3. MatrixXd beq\_con\_v = MatrixXd::Zero(m-1, 1);
4. **for** (**int** i = 0; i < m - 1; i++){
5. **for** (**int** j = 1; j < p\_num1d; j++){
6. Aeq\_con\_v(i, i \* p\_num1d + j) = j \* pow(Time(i), j - 1);
7. }
8. //cout << "(i + 1) \* p\_num1d + 1:" << (i + 1) \* p\_num1d + 1 << endl;
9. Aeq\_con\_v(i, (i + 1) \* p\_num1d + 1) = -1;
10. }

### 加速度连续性

1. //acceleration continuity constrain
2. MatrixXd Aeq\_con\_a = MatrixXd::Zero(m-1, p\_num1d \* m);
3. MatrixXd beq\_con\_a = MatrixXd::Zero(m-1, 1);
4. **for** (**int** i = 0; i < m-1;i++){
5. **for** (**int** j = 2; j < p\_num1d;j++){
6. Aeq\_con\_a(i, i \* p\_num1d + j) =j\*(j-1)\*pow(Time(i), j-2);
7. }
8. Aeq\_con\_a(i, (i + 1) \* p\_num1d + 2) = -2;
9. }

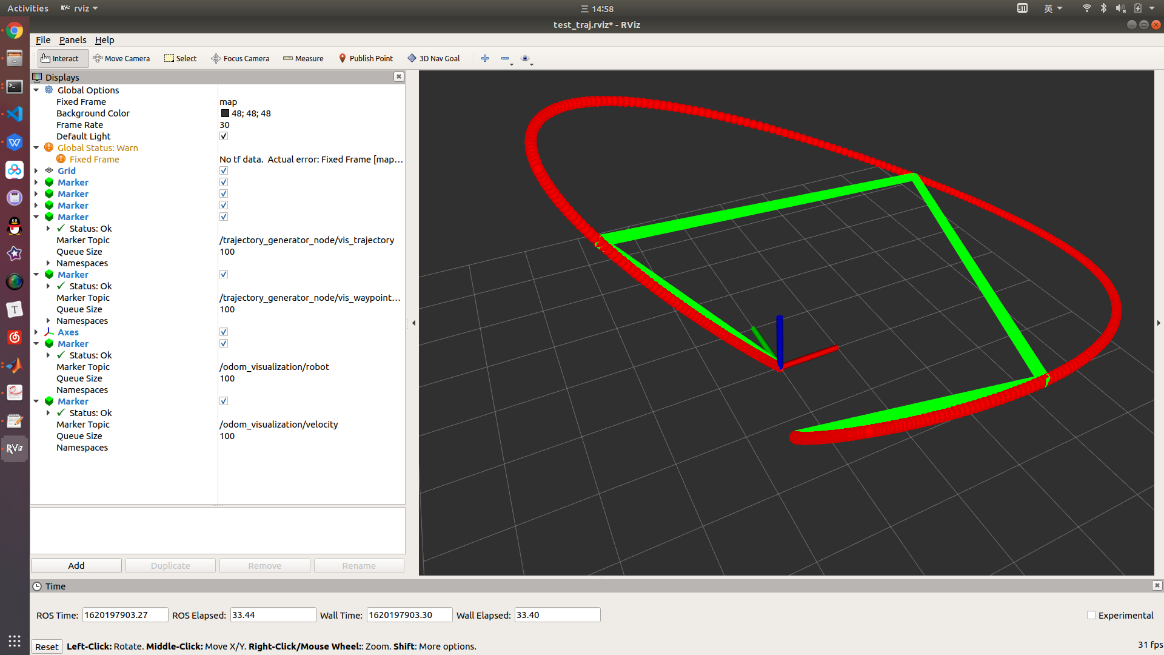
### Jerk连续性

1. //jerk continuity constrain
2. MatrixXd Aeq\_con\_j = MatrixXd::Zero(m-1, p\_num1d \* m);
3. MatrixXd beq\_con\_j = MatrixXd::Zero(m-1, 1);
4. **for** (**int** i = 0; i < m-1;i++){
5. **for** (**int** j = 3; j < p\_num1d;j++){
6. Aeq\_con\_j(i, i \* p\_num1d + j) =j\*(j-1)\*(j-2)\*pow(Time(i), j-3);
7. }
8. Aeq\_con\_j(i, (i + 1) \* p\_num1d + 3) = -6;
9. }

### 最后将所有矩阵合并

1. MatrixXd Aeq = MatrixXd::Zero(5 \* (m - 1) + 2 \* 4, p\_num1d \* m);
2. MatrixXd beq = MatrixXd::Zero(5 \* (m - 1) + 2 \* 4, 1);
3. Aeq.block(0, 0, 4, p\_num1d \* m) = Aeq\_start;
4. Aeq.block(4, 0, 4, p\_num1d \* m) = Aeq\_end;
5. Aeq.block(8, 0, m - 1, p\_num1d \* m) = Aeq\_wp;
6. Aeq.block(8 + 1 \* (m - 1), 0, m - 1, p\_num1d \* m) = Aeq\_con\_p;
7. Aeq.block(8 + 2 \* (m - 1), 0, m - 1, p\_num1d \* m) = Aeq\_con\_v;
8. Aeq.block(8 + 3 \* (m - 1), 0, m - 1, p\_num1d \* m) = Aeq\_con\_a;
9. Aeq.block(8 + 4 \* (m - 1), 0, m - 1, p\_num1d \* m) = Aeq\_con\_j;
11. beq.block(0, 0, 4, 1) = beq\_start;
12. beq.block(4, 0, 4, 1) = beq\_end;
13. beq.block(8, 0, m - 1, 1) = beq\_wp;
14. beq.block(8 + 1 \* (m - 1), 0, m - 1, 1) = beq\_con\_p;
15. beq.block(8 + 2 \* (m - 1), 0, m - 1, 1) = beq\_con\_v;
16. beq.block(8 + 3 \* (m - 1), 0, m - 1, 1) = beq\_con\_a;
17. beq.block(8 + 4 \* (m - 1), 0, m - 1, 1) = beq\_con\_j;

## 结果



# 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。

## 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



1. MatrixXd M = MatrixXd::Zero(p\_num1d \* m, p\_num1d \* m);
2. MatrixXd Mi;
4. **for** (**int** k = 0; k < m; k++)
5. {
6. Mi = MatrixXd::Zero(p\_num1d, p\_num1d);
7. //time=0
8. Mi(0, 0) = 1;
9. Mi(1, 1) = 1;
10. Mi(2, 2) = 2;
11. Mi(3, 3) = 6;
12. //p
13. **for** (**int** i = 0; i < p\_num1d;i++){
14. Mi(4, i) = pow(Time(k), i);
15. }
16. //v
17. **for** (**int** i = 1; i < p\_num1d;i++){
18. Mi(5, i) = i \* pow(Time(k), i - 1);
19. }
20. //a
21. **for** (**int** i = 2; i < p\_num1d;i++){
22. Mi(6, i) = i \* (i - 1) \* pow(Time(k), i - 2);
23. }
24. //j
25. **for** (**int** i = 3; i < p\_num1d;i++){
26. Mi(7, i) = i \* (i - 1) \* (i - 2) \* pow(Time(k), i - 3);
27. }
28. M.block(k \* p\_num1d, k \* p\_num1d, p\_num1d, p\_num1d) = Mi;
29. }

## 选择矩阵C

将已知的变量映射为，未知的每段轨迹连接处的速度、加速度和加加速度映射为。



1. MatrixXd Ct = MatrixXd::Zero(2 \* d\_order \* m, d\_order \* (m + 1));
2. Ct(0, 0) = 1;
3. Ct(1, 1) = 1;
4. Ct(2, 2) = 1;
5. Ct(3, 3) = 1;
7. Ct(2 \* d\_order \* m - 4, d\_order + m - 1) = 1;
8. Ct(2 \* d\_order \* m - 3, d\_order + m + 0) = 1;
9. Ct(2 \* d\_order \* m - 2, d\_order + m + 1) = 1;
10. Ct(2 \* d\_order \* m - 1, d\_order + m + 2) = 1;
12. **for** (**int** i = 0; i < m-1; i++){
14. Ct(d\_order+2\*i\*d\_order,d\_order+i) = 1;
15. Ct(d\_order+2\*i\*d\_order+d\_order,d\_order+i) = 1;
17. Ct(d\_order + 2 \* i \* d\_order + 1, 2 \* d\_order + m + i \* (d\_order - 1) - 1) = 1; // v\_end
18. Ct(d\_order+2\*i\*d\_order+2,2\*d\_order+m+i\*(d\_order-1)+0)=1;// a\_end
19. Ct(d\_order+2\*i\*d\_order+3,2\*d\_order+m+i\*(d\_order-1)+1)=1;// j\_end
21. Ct(d\_order+2\*i\*d\_order+1+d\_order,2\*d\_order+m+i\*(d\_order-1)-1)=1;// v\_start
22. Ct(d\_order+2\*i\*d\_order+2+d\_order,2\*d\_order+m+i\*(d\_order-1)+0)=1;// a\_start
23. Ct(d\_order+2\*i\*d\_order+3+d\_order,2\*d\_order+m+i\*(d\_order-1)+1)=1;// j\_start
24. }

## 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



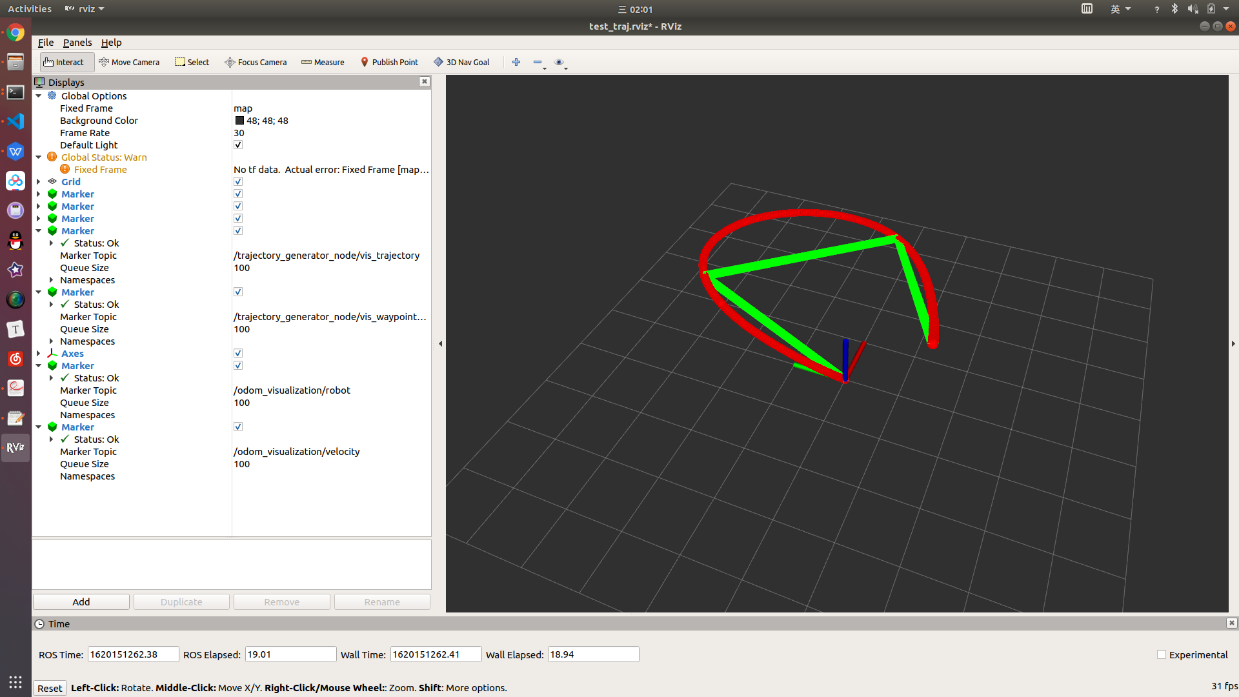
求导得到最优的，再代入公式



得到每段轨迹对应的多项式系数。

1. auto R = C \* M.inverse().transpose() \* Q \* M.inverse() \* Ct;
3. **int** Fsize = Path.size() / 3 - 2 + 2 \* 4;
4. **int** Psize = R.rows() - Fsize;
5. auto R\_pp = R.block(Fsize, Fsize, Psize, Psize);
6. auto R\_fp = R.block(0, Fsize, Fsize, Psize);
7. cout << "Path:" << endl
8. << Path << endl;
9. **for** (**int** i = 0; i < 3;i++){
10. 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);
11. VectorXd dF(Fsize);
12. dF << start\_cond, Path.block(1,i,Path.size() / 3 - 2, 1), end\_cond;
13. auto dP = -R\_pp.inverse() \* R\_fp.transpose() \* dF;
14. VectorXd d(Fsize + Psize);
15. d << dF, dP;
17. MatrixXd P1 = M.inverse() \* Ct \* d;
18. P1 = P1.transpose();
19. MatrixXd P(m, p\_num1d);
20. **for** (**int** j = 0; j < m; j++)
21. {
22. P.block(j,0,1,p\_num1d) = P1.block(0, j \* p\_num1d, 1, p\_num1d);
23. }
24. PolyCoeff.block(0, i \* p\_num1d, m, p\_num1d) = P;
25. }

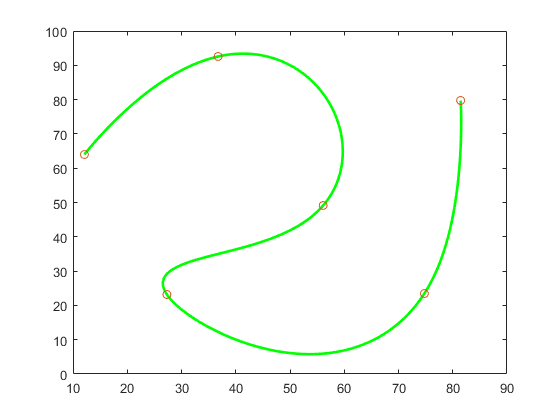
## 结果



# Matlab实现

MATLAB的实现也是一样的原理，这里就直接放上结果

## 使用QP求解器的结果



## 闭式解结果

