6 机器臂轨迹规划实验

6.1 机械臂轨迹规划算法简介

现在机器人比较热，特别是工业机器人领域，中国是装机量排名靠前的市场，也是发展最快的市场，各大机器人厂家纷纷抢占市场，有四大机器人本体厂家之说：ABB、库卡(Kuka)、安川(Yasukawa)、发那科(Fanuc)。笔者有幸在十二五期间参与到国家 863 重点项目研究，那时我们就提出要攻克机器人核心技术，看样子这条路到现在来说绝对是正确的，特别是在中美贸易冲突的背景下，被人卡脖子的事情太多了，我国提出的智能制造，其中工业机器人的自主知识产权至关重要。

当前国家、地方出台机器人相关优惠政策很多，国产机器人蜂拥而起，竞争相当激烈，这个事情有利也有弊，有利的是可以加快我国工业机器人的发展，但最大的弊病就是低端的竞争带来了低廉的价格战，对我国的机器人发展非常不利，很多投机分子做了一个机械的本体，就开始忽悠政府、忽悠投资人，相当一部分人连基本的机器人原理都没有掌握，运动学、动力学一知半解，更谈不上控制优化、工艺改进以及品质了。实际上机器人在国内发展了很多年，有一些高校的机器人所都非常著名，然而不少的高校还是不太明白机器人，即使给学生开了课，但除了操作，基本没有实际开发实验，除了专业研究机器人的研究生，其他的开发人员相当紧缺，这应该是我们的机器人发展落后的深层次原因之一。

扯远了，我们还是回到我们的主题，今天的主题是讲解机械臂的轨迹规划实验。机械臂有了正解、逆解，这是最基本的工作，如果我们要让我们的机器人能够较好地进行不同的作业，必须配备轨迹规划算法。

机器人有多种形态，总的来说都需要运动规划(motion planning)，但工业机器人我们一般不需避障，所以一般叫轨迹规划(trajectory planning)，移动机器人最早都是基于水平面的运动，所以一般叫路径规划(path planning)，但现在机器人已经需要在非结构化 3D 环境运动，还需要避障，所以 motion planning 包含的内容更广一些。本章目前为止只介绍 trajectory planning 的基本算法。

一般教科书上都会介绍基于**笛卡尔空间的轨迹规划**算法以及基于**关节空间的轨迹规划**算法。基于笛卡尔空间的轨迹规划算法，在直角坐标系下进行两点之间的轨迹插补，然后求位置点的逆解，再到关节空间进行位置、速度、加速度控制。基于关节空间的轨迹规划，直接在关节空间进行关节变化量的求解，利用一阶导得到关节速度、二阶导得到关节加速度，进而实现作业轨迹控制。有些情况下，不如特殊的精细作业、高速作业，有可能还需要考察关节角变化量的三阶导(jerk)，考察轴的运动冲击问题。

本章实验给大家介绍教科书上几个基本的轨迹规划算法：Ctraj、Jtraj（三次多项式、五次多项式）以及三次样条平滑（Smooth）与插补(Interpolation)算法实

验。

6.2 Ctraj 算法

已知初始和终止作业点的末端工具位姿，利用匀加速、匀减速运动来规划轨迹的算法，在 peter coker 的 matlab robotics toolbox 里面有这个 ctraj.m 文件。这个算法最简单，只是在笛卡尔空间对末端工具坐标值进行轨迹插补。

我把它翻译成 c++代码，在 ros 下运行，代码如下：

```
//关节空间直线插补函数
Eigen::Matrix<double,MOVLNUM,6> ctrajMat(Eigen::Matrix<double,4,4> startPos, Eigen::Matrix<double,4,4> endPos){
    Eigen::MatrixXd radVector= Eigen::Matrix<double,1,6>::Identity();

    int step;
    Eigen::MatrixXd temPos(4,4);

    step=MOVLNUM;
    Eigen::MatrixXd rtnMatPoints(step,6);
    double qf[6] = {0, M_PI/2, 0, M_PI/2 , 0 ,0};//start point joint angle


    double lx;
    double ly;
    double lz;
    lx=endPos(0,3)-startPos(0,3);
    ly=endPos(1,3)-startPos(1,3);
    lz=endPos(2,3)-startPos(2,3);

    float deltax,deltay,deltaz;

    deltax=(double)lx/step;
    deltay=(double)ly/step;
    deltaz=(double)lz/step;

    temPos=startPos;
```

```
    for(int i=0;i<step;++i)
    {
        temPos(0,3)=startPos(0,3)+(i+1)*(deltax);
        temPos(1,3)=startPos(1,3)+(i+1)*(deltay);
        temPos(2,3)=startPos(2,3)+(i+1)*(deltaz);

        radVector = getOptimalIK(temPos,qf);

        for(int j=0;j<6;++j)
        {
            rtnMatPoints(i,j)= radVector(0,j);
        }

    }
    std::cout<<"planning points matrix:\n"<<rtnMatPoints<<std::endl;

    return rtnMatPoints;
}
```

我们再简单写个测试例程：

```
void testCTraj(){
    Eigen::Matrix<double,4,4> startPos, endPos;
    int step = MOVLNUM;
    Eigen::MatrixXd PlanPoints(step,6);

    points.header.stamp = ros::Time::now();
    points.action = visualization_msgs::Marker::ADD;

    points.scale.x = 0.01f;
    points.scale.y = 0.01f;
    points.scale.z = 0.01f;
    points.color.r = 1.0f;
    points.color.g = 0.0f;
    points.color.b = 0.0f;
    points.color.a = 1.0;
```

```
    startPos = ForwardKinematics(0,-M_PI/2.0,-0,-M_PI/2.0,0,0);//from start position real pos= 0 0 0 0 0 0
    endPos = ForwardKinematics(1.570796,-0.785398,0,-1.570796,0.174533,0.174533);//to an end position

    DisplayPoint(startPos(0,3),startPos(1,3),startPos(2,3));
    DisplayPoint(endPos(0,3),endPos(1,3),endPos(2,3));

    points.color.r = 0.0f;
    points.color.g = 1.0f;

    double q[step][6];
    Eigen::Matrix<double,4,4> TcpCenter;
    double pointX,pointY,pointZ;

    moveJ(0,0,0,0,0,0);//move to start pos

    //getchar();

    PlanPoints = ctrajMat(startPos,endPos);

    //getchar();

    for(int i=0;i<step;i++){

        for(int j=0;j<6;j++){
            q[i][j]=PlanPoints(i,j);
        }

        moveJ(q[i][0],q[i][1]+M_PI/2.0,-q[i][2],q[i][3]+M_PI/2.0,q[i][4],q[i][5]);


        TcpCenter=ForwardKinematics(q[i][0],q[i][1],q[i][2],q[i][3],q[i][4],q[i][5]);
```

```
        pointX=TcpCenter(0,3);
        pointY=TcpCenter(1,3);
        pointZ=TcpCenter(2,3);



        DisplayPoint(pointX,pointY,pointZ);
        //getchar();
    }
}
```

先还是给出起始点关节角及终止点关节角，然后用正解解出起始点、终止点的姿态矩阵，在笛卡尔空间做匀速等距离轨迹插补，反求各插补点的关节角，然后控制关节转动。图 6.1 为插补效果图。

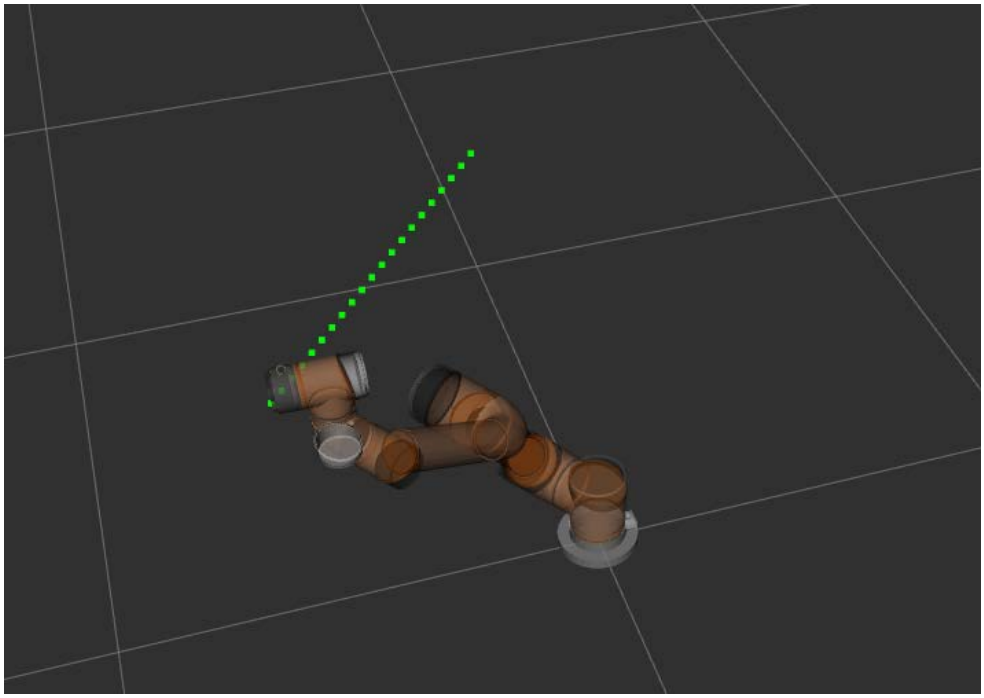


图 6.1 Ctraj 直线插补运动

通过 kazam 屏幕录制也可以看到动画效果。

https://github.com/mhuasong/AUBO-Robot-on-ROS/blob/master/video/ctraj.mp4

大家可以通过和 matlab 的程序运行对比一下。

6.3 Jtraj 算法

已知初始和终止的关节角度，利用多项式在关节空间来进行规划轨迹。在 peter coker 的 matlab robotics toolbox 里面有这个 jtraj.m 文件。这个算法也很简

单，和 Ctraj 不同的是，jtraj 在关节空间直接对所有关节角进行多项式求解，可以无需再进行逆解，将求出的关节角、关节速度、加速度直接拿来去控制机械臂。

下面我们还是把它翻译成 C++代码，为了比较我还扩展了三次多项式的关节空间插补，再加上五次多项式的关节空间插补，代码如下：

```
//三次多项式关节空间插补
Eigen::Matrix<double,MOVPNUM,6> jtrajCubic(Eigen::MatrixXd qStart, Eigen::MatrixXd qEnd){
    int n;
    double m;
    n=MOVPNUM;
    m=n-1;
    Eigen::MatrixXd t(n,1);

    Eigen::MatrixXd q0(1,6), q1(1,6);
    q0=qStart;
    q1=qEnd;

    int i=0;
    double tscal=n;
    //normalized time from 0->1
    for(;i<n;i++)
    {
        t(i,0)=i/m;
    }

    Eigen::MatrixXd qd0(1,6); //not using velocities boundary [qd0 qd1]
    qd0 = Eigen::Matrix<double,1,6>::Zero();
    Eigen::MatrixXd qd1(1,6);
    qd1 = Eigen::Matrix<double,1,6>::Zero();

    Eigen::MatrixXd A(1,6);
    Eigen::MatrixXd B(1,6);
    Eigen::MatrixXd C(1,6);
```

```
//compute the polynomal coefficients
A = -2*(q1-q0);
B = 3*(q1 - q0);
C = q0;


Eigen::MatrixXd tt(n,4);
for(i=0;i<n;i++){
	tt(i,0) = pow(t(i,0),3);
	tt(i,1) = pow(t(i,0),2);
	tt(i,2) = pow(t(i,0),1);
	tt(i,3) = 1;
}

Eigen::MatrixXd c(4,6);
Eigen::MatrixXd temp(1,6);
temp<<Eigen::MatrixXd::Zero(1,6);
for(int i=0;i<6;i++)
{
	c(0,i)=A(0,i);
	c(1,i)=B(0,i);
	c(2,i)=temp(0,i);
	c(3,i)=C(0,i);
}

Eigen::MatrixXd qt(n,6);

qt=tt*c;

for(int i=0;i<n;++i)
{
	for(int j=0;j<6;++j)
	{
		if(fabs(qt(i,j))<ZERO_THRESH)
			qt(i,j)=0;
```

```
        }
    }

    //calculate velocity
    Eigen::MatrixXd qdt(n,6);
    for(int i=0;i<6;i++)
    {
        c(0,i)=temp(0,i);
        c(1,i)=3*A(0,i);
        c(2,i)=2*B(0,i);
        c(3,i)=temp(0,i);
    }
    qdt = tt*c/tscal;

    //calculate acceleration
    Eigen::MatrixXd qddt(n,6);
    for(int i=0;i<6;i++)
    {
        c(0,i)=temp(0,i);
        c(1,i)=6*A(0,i);
        c(2,i)=2*B(0,i);
        c(3,i)=temp(0,i);
    }
    qddt = tt*c/pow(tscal,2);


    return qt;

}
    //五次多项式的关节空间插补
Eigen::Matrix<double,MOVPNUM,6>  jtrajQuintic(Eigen::MatrixXd  qStart,
Eigen::MatrixXd qEnd){
    int n;
    double m;
    n=MOVPNUM;
```

```
m=n-1;
Eigen::MatrixXd t(n,1);

Eigen::MatrixXd q0(1,6), q1(1,6);
q0=qStart;
q1=qEnd;

int i=0;
double tscal=n;
//normalized time from 0->1
for(;i<n;i++)
{
    t(i,0)=i/m;
}

Eigen::MatrixXd qd0(1,6); //not using velocities boundary [qd0 qd1]
qd0 = Eigen::Matrix<double,1,6>::Zero();
Eigen::MatrixXd qd1(1,6);
qd1 = Eigen::Matrix<double,1,6>::Zero();

Eigen::MatrixXd A(1,6);
Eigen::MatrixXd B(1,6);
Eigen::MatrixXd C(1,6);
Eigen::MatrixXd E(1,6);
Eigen::MatrixXd F(1,6);

//compute the polynomal coefficients
A = 6*(q1-q0)-3*(qd1+qd0)*tscal;
B = -15*(q1 - q0) + (8*qd0 + 7*qd1)*tscal;
C = 10*(q1 - q0) - (6*qd0 + 4*qd1)*tscal;
E = qd0*tscal;
F = q0;



Eigen::MatrixXd tt(n,6);
```

```
for(i=0;i<n;i++){
    tt(i,0) = pow(t(i,0),5);
    tt(i,1) = pow(t(i,0),4);
    tt(i,2) = pow(t(i,0),3);
    tt(i,3) = pow(t(i,0),2);
    tt(i,4) = pow(t(i,0),1);
    tt(i,5) = 1;
}

Eigen::MatrixXd c(6,6);
Eigen::MatrixXd temp(1,6);
temp<<Eigen::MatrixXd::Zero(1,6);
for(int i=0;i<6;i++)
{
    c(0,i)=A(0,i);
    c(1,i)=B(0,i);
    c(2,i)=C(0,i);
    c(3,i)=temp(0,i);
    c(4,i)=E(0,i);
    c(5,i)=F(0,i);
}

Eigen::MatrixXd qt(n,6);
qt=tt*c;

for(int i=0;i<n;++i)
{
    for(int j=0;j<6;++j)
    {
        if(fabs(qt(i,j))<ZERO_THRESH)
            qt(i,j)=0;
    }
}

//calculate velocity
```

```
    Eigen::MatrixXd qdt(n,6);
    for(int i=0;i<6;i++)
    {
        c(0,i)=temp(0,i);
        c(1,i)=5*A(0,i);
        c(2,i)=4*B(0,i);
        c(3,i)=3*C(0,i);
        c(4,i)=temp(0,i);
        c(5,i)=E(0,i);
    }
    qdt = tt*c/tscal;

    //calculate acceleration
    Eigen::MatrixXd qddt(n,6);
    for(int i=0;i<6;i++)
    {
        c(0,i)=temp(0,i);
        c(1,i)=temp(0,i);
        c(2,i)=20*A(0,i);
        c(3,i)=12*B(0,i);
        c(4,i)=6*C(0,i);
        c(5,i)=temp(0,i);
    }
    qddt = tt*c/pow(tscal,2);


    return qt;

}
```

三次多项式和五次多项式插补，在坐标轨迹上差别不大，但在关节速度、加速度方面上，五次多项式更优，这个教科书和讲义里都有讲，我写的函数，速度、加速度都有求，但没有传入 moveJ 函数去进行实际控制，大家可以在后面的实验中，自行加入速度、加速度控制。

测试例程：

```
void testJTraj(){
```

```
    Eigen::Matrix<double,4,4> startPos, endPos;
    int step = MOVPNUM;
    Eigen::MatrixXd PlanPoints(step,6);

    points.header.stamp = ros::Time::now();
    points.action = visualization_msgs::Marker::ADD;

    points.scale.x = 0.01f;
    points.scale.y = 0.01f;
    points.scale.z = 0.01f;
    points.color.r = 1.0f;
    points.color.g = 0.0f;
    points.color.b = 0.0f;
    points.color.a = 1.0;

    Eigen::MatrixXd qStart(1,6),qEnd(1,6);
    qStart<<0,-M_PI/2.0,-0,-M_PI/2.0,0,0;
    qEnd<<1.570796,-0.785398,0,-0.785398,0.174533,0.174533;

    std::cout<<"starting draw start point and end point"<<std::endl;

    startPos = ForwardKinematics(0,-M_PI/2.0,-0,-M_PI/2.0,0,0);//from start position real pos= 0 0 0 0 0 0
    endPos = ForwardKinematics(1.570796,-0.785398,0,-0.785398,0.174533,0.174533);//to an end position

    DisplayPoint(startPos(0,3),startPos(1,3),startPos(2,3));
    DisplayPoint(endPos(0,3),endPos(1,3),endPos(2,3));

    std::cout<<startPos<<std::endl<<endPos<<std::endl;
    points.color.r = 0.0f;
    points.color.g = 1.0f;
```

```
double q[step][6];
Eigen::Matrix<double,4,4> TcpCenter;
double pointX,pointY,pointZ;

moveJ(0,0,0,0,0,0);//move to start pos

//getchar();
printf("\nPress 1 for Quintic polynomial. \
\nPress any key for Cubic polynomial.\n");
int command;
scanf("%d",&command);
if(command==1){
    std::cout<<"Quintic polynomial."<<std::endl;
    PlanPoints = jtrajQuintic(qStart,qEnd);
}
else{
    std::cout<<"Cubic polynomial."<<std::endl;
    PlanPoints = jtrajCubic(qStart,qEnd);
}
//getchar();

for(int i=0;i<step;i++){

    for(int j=0;j<6;j++){
        q[i][j]=PlanPoints(i,j);
    }

    moveJ(q[i][0],q[i][1]+M_PI/2.0,-q[i][2],q[i][3]+M_PI/2.0,q[i][4],q[i][5]);

    TcpCenter=ForwardKinematics(q[i][0],q[i][1],q[i][2],q[i][3],q[i][4],q[i][5]);

    pointX=TcpCenter(0,3);
    pointY=TcpCenter(1,3);
    pointZ=TcpCenter(2,3);
```

```
            DisplayPoint(pointX,pointY,pointZ);
            //getchar();
        }
}
```

测试效果如图 6.2 所示，可以看到关节空间的多项式插补算法，末端轨迹是从起点到终点的一段圆弧。

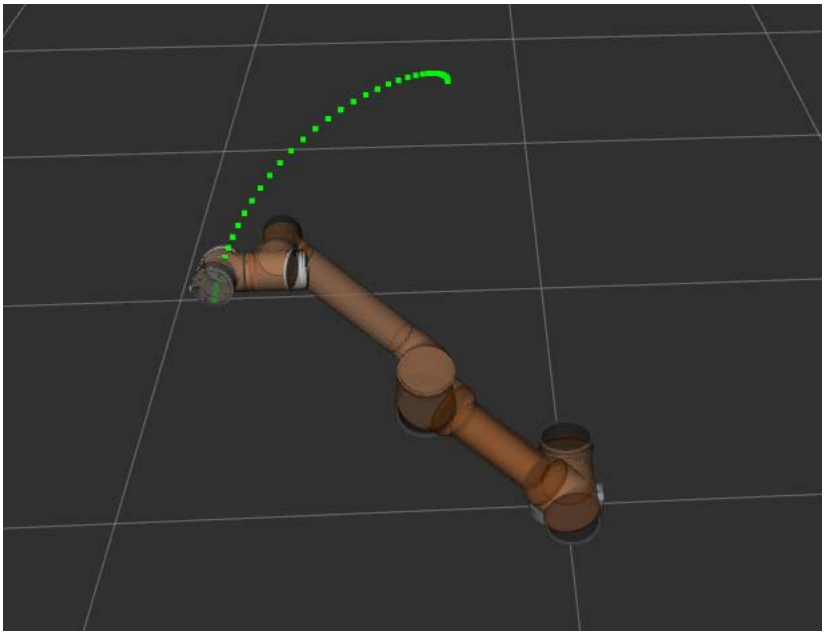


图 6.2 (Quintic)五次多项式关节空间插补效果

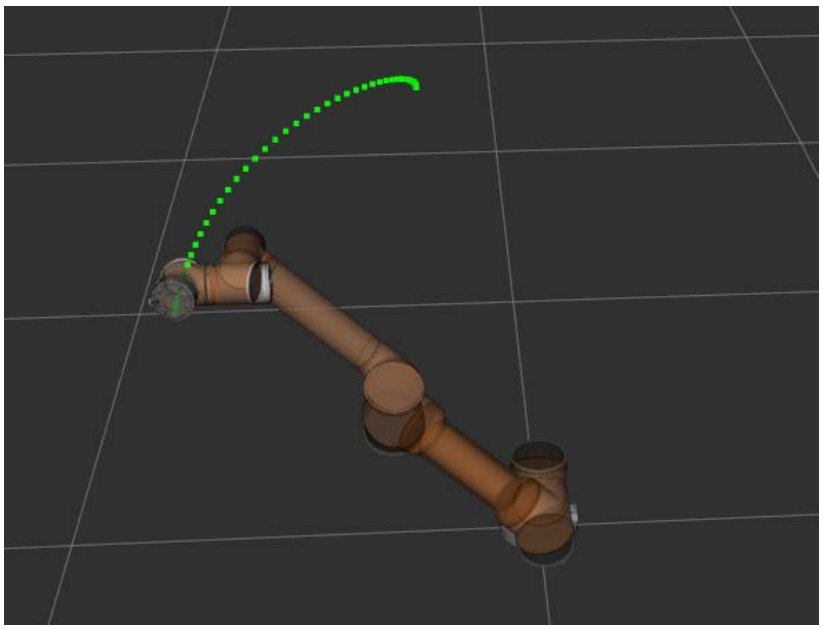


图 6.3 (Cubic)三次多项式关节空间插补效果

通过 kazam 屏幕录制也可以看到动画效果。

https://github.com/mhuasong/AUBO-Robot-on-ROS/blob/master/video/jtraj.mp4

一般教科书书还有讲抛物线、过路点(way point)的抛物线插补，但这个我觉得在实际的工程应用中用处不多，所以本实验教程不讲这个。给大家布置个作业，用关节空间插补算法画一个完整的圆。

在实际的工程应用中，比如：需要走轨迹的缝焊、空间曲面的喷涂、打磨、涂胶等作业，我们可以采用图像传感器采集图像，进行轨迹、轮廓识别，然后再规划机器人的作业路径，控制执行自动作业，不能简单地用示教器提供的插补轨迹函数去示教编程，这个时候我们需要进行复杂一点的轨迹规划。

空间曲线的插补算法，除了前面我们介绍的圆弧插补之外，往往是一些非规则的作业路径，这个时候我们无法简单用规则的圆弧或直线去逼近，这个时候我们需要了解曲线插补算法。讲义和教材上有样条函数的插补原理，只要有曲线公式，我们即可进行复杂的曲线逼近，这里我只介绍两个简单的多段式曲线规划算法，一种是采用多段样条函数进行光滑计算，另一种是经过所有路点的平滑曲线，后面一种可能用的会更多。

6.4 Cubic Bspline 平滑规划算法（拟合）

三次 B 样条函数是一种性能优良的曲线光滑算法，一般需要最少 4 个控制点。这里我举一个 7 个控制点的曲线参数化平滑算法，其原理见讲义，这里节约篇幅，只贴出示范代码，供大家参考。

```
void testBspline(){
    //draw    marker point
    points.action = visualization_msgs::Marker::ADD;
    points.lifetime=ros::Duration();
    points.scale.x = 0.005f;
    points.scale.y = 0.005f;
    points.scale.z = 0.01f;
    points.color.r = 1.0f;
    points.color.g = 0.0f;
    points.color.b = 0.0f;
    points.color.a = 1.0;

    // the control points for the curve
    double Points[7][3] = {
```

```
        {-0.5,0.4,0.3},
        {-0.32,0.6,0.3},
        {0.2,0.45,0.3},
        {0.3,0.63,0.25},
        {0.34,0.56,0.3},
        {0.36,0.45,0.3},
        {0.40,0.51,0.27}
};

for(int j=0;j<NUM_POINTS;j++){
        std::cout<<Points[j][0]<<" "<<Points[j][1]<<" "<<Points[j][2]<<std::endl;
        DisplayPoint( Points[j][0],Points[j][1],Points[j][2] );

}
//getchar();
double qf[6] = {0, M_PI/2, 0, M_PI/2 , 0 ,0};//start point joint angle
Eigen::Matrix<double,4,4> temPos;
Eigen::Matrix3d poseT;
Eigen::MatrixXd q= Eigen::Matrix<double,1,6>::Identity();

// the level of detail of the curve
unsigned int LOD=40;
poseT = RPY2R(M_PI/4.0,M_PI,0);

// start position
temPos.block<3,3>(0,0) = poseT.block<3,3>(0,0);
temPos(0,3) = Points[0][0];
temPos(1,3) = Points[0][1];
temPos(2,3) = Points[0][2];
temPos(3,3) = 1;

// use the parametric time value 0 to 1
for(int start_cv=-3,m=0;m!=NUM_SEGMENTS;++m,++start_cv){
        for(int i=0;i!=LOD;++i) {
```

```
double t = (double)i/(LOD-1);

// the t value inverted
double it = 1.0-t;

// calculate blending functions
double b0 = it*it*it/6.0;
double b1 = (3*t*t*t - 6*t*t +4)/6.0;
double b2 = (-3*t*t*t +3*t*t + 3*t + 1)/6.0;
double b3 =    t*t*t/6.0;

// sum the control points mulitplied by their respective blending functions
double x = b0*GetPoint(start_cv+0,Points)[0] +
        b1*GetPoint(start_cv+1,Points)[0] +
        b2*GetPoint(start_cv+2,Points)[0] +
        b3*GetPoint(start_cv+3,Points)[0] ;

double y = b0*GetPoint(start_cv+0,Points)[1] +
        b1*GetPoint(start_cv+1,Points)[1] +
        b2*GetPoint(start_cv+2,Points)[1] +
        b3*GetPoint(start_cv+3,Points)[1] ;

double z = b0*GetPoint(start_cv+0,Points)[2] +
        b1*GetPoint(start_cv+1,Points)[2] +
        b2*GetPoint(start_cv+2,Points)[2] +
        b3*GetPoint(start_cv+3,Points)[2] ;

temPos(0,3) = x;
temPos(1,3) = y;
temPos(2,3) = z;

q = getFirstIK(temPos,qf);
moveJ(q(0),q(1)+M_PI/2.0,-q(2),q(3)+M_PI/2.0,q(4),q(5));
// Display the point
```

```
                DisplayPoint( x,y,z );
            }
        }

        //specify the last point on the curve
        temPos(0,3) = Points[NUM_POINTS-1][0];
        temPos(1,3) = Points[NUM_POINTS-1][1];
        temPos(2,3) = Points[NUM_POINTS-1][2];

        q = getFirstIK(temPos,qf);
        moveJ(q(0),q(1)+M_PI/2.0,-q(2),q(3)+M_PI/2.0,q(4),q(5));
        // Display the last point
        DisplayPoint( temPos(0,3) ,temPos(1,3),temPos(2,3) );
}
```

在前面的历程中，我们都是先给出关节角，然后求逆解得到姿态矩阵，但在应用中，我们往往只知道坐标值，而不知道姿态矩阵，前面逆解章节我们讲过姿态矩阵的表达方法，这里我们简单编写了一个用 RPY 角生成姿态矩阵的函数，代码如下：

```
Eigen::Matrix3d RPY2R(double roll, double pitch, double yaw){
    Eigen::AngleAxisd rollAngle(roll, Eigen::Vector3d::UnitX());
    Eigen::AngleAxisd pitchAngle(pitch, Eigen::Vector3d::UnitY());
    Eigen::AngleAxisd yawAngle(yaw, Eigen::Vector3d::UnitZ());
    Eigen::Quaterniond    q = rollAngle*pitchAngle*yawAngle;

    Eigen::Matrix3d rotationMatrix = q.matrix();
    return rotationMatrix;
}
```

在测试过程中，姿态矩阵我直接指定了 roll、pitch、yaw 角度：

```
poseT = RPY2R(M_PI/4.0,M_PI,0);
```

测试效果如图 6.4 所示。

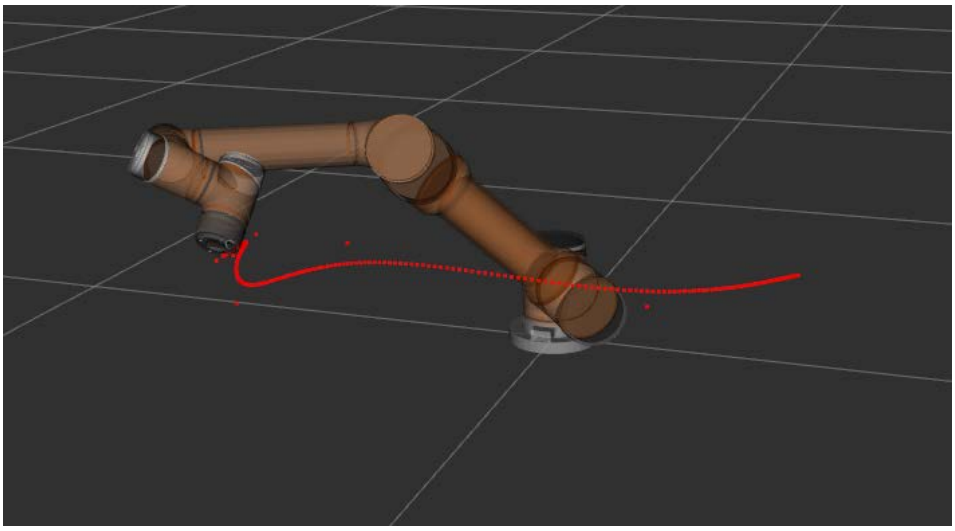


图 6.4 Cubic B-Spline 圆滑轨迹规划

通过 kazam 屏幕录制也可以看到动画效果。

https://github.com/mhuasong/AUBO-Robot-on-ROS/blob/master/video/cubic_bspline.mp4

6.5 过所有路点的插补算法（参数化插补）

在很多作业中，我们可以采用传感器得到需要作业的曲线轨迹或曲面加工轨迹，这个时候不能只是靠控制点来圆滑轨迹，很多的加工工艺都要求机器人末端工具经过给出的所有路点。这个算法有很多种，比如贝塞尔曲线、三次样条、五次样条、非均匀有理样条函数等，算法非常多，这里我介绍一种简单的算法，在游戏程序里经常用到，那就是 Catmull-Rom 样条函数算法，这个算法根据 p0,p1,p2,p3 共 4 个连续点的坐标，在 p1 和 p2 之间，根据参数 t (取值范围[0.0f,1.0f])插值，使得曲线更平滑。Catmull-Rom 算法本来只进行 p1 到 p2 之间的参数化平滑，但我们可以认为给它首尾加两段构成三段{2p0-p1,p0,p1,p2}、{p0,p1,p2,p3}、{p1,p2,p3,2p3-p2}，让它能够经过 4 个路点进行平滑。

矩阵表达式：

(参见：https://www.cs.cmu.edu/~462/projects/assn2/assn2/catmullRom.pdf)

$$\mathbf{p}(s) = \begin{bmatrix} 1 & u & u^2 & u^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau-3 & 3-2\tau & -\tau \\ -\tau & 2-\tau & \tau-2 & \tau \end{bmatrix} \begin{bmatrix} \mathbf{p}_{i-2} \\ \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \end{bmatrix} \quad (6\text{-}1)$$

公式很简单，我略去说明，看代码即可明白。

```
void testCatmullRom(){
    //draw    marker point
    points.action = visualization_msgs::Marker::ADD;
```

```
points.lifetime=ros::Duration();
points.scale.x = 0.005f;
points.scale.y = 0.005f;
points.scale.z = 0.01f;
points.color.r = 1.0f;
points.color.g = 0.0f;
points.color.b = 0.0f;
points.color.a = 1.0;

// the control points for the curve
double Points[4][3] = {
    {-0.5,0.4,0.3},
    {-0.32,0.6,0.3},
    {0.2,0.45,0.3},
    {0.3,0.63,0.25},
};

for(int j=0;j<4;j++){
    std::cout<<Points[j][0]<<" "<<Points[j][1]<<" "<<Points[j][2]<<std::endl;
    DisplayPoint( Points[j][0],Points[j][1],Points[j][2] );

}

getchar();
getchar();
double qf[6] = {0, M_PI/2, 0, M_PI/2 , 0 ,0};//reference point joint angle
Eigen::Matrix<double,4,4> temPos;
Eigen::Matrix3d poseT;
Eigen::MatrixXd q= Eigen::Matrix<double,1,6>::Identity();

// the level of detail of the curve
unsigned int LOD=40;
poseT = RPY2R(M_PI/4.0,M_PI,0);

// start position
```

```
    temPos.block<3,3>(0,0) = poseT.block<3,3>(0,0);
    temPos(0,3) = Points[0][0];
    temPos(1,3) = Points[0][1];
    temPos(2,3) = Points[0][2];
    temPos(3,3) = 1;
    Eigen::Vector3d P0,P1,P2,P3;

    //the first segment curve
    P0<<2*Points[0][0]-Points[1][0],2*Points[0][1]-Points[0][1],2*Points[0][2]-Points
[1][2];
    P1<<Points[0][0],Points[0][1],Points[0][2];
    P2<<Points[1][0],Points[1][1],Points[1][2];
    P3<<Points[2][0],Points[2][1],Points[2][2];

    double factor = 0.5;
    Eigen::Vector3d c0 = P1;
    Eigen::Vector3d c1 = (P2-P0)*factor;
    Eigen::Vector3d c2 = (P2-P1)*3.0-(P3-P1)*factor-(P2-P0)*2.0*factor;
    Eigen::Vector3d c3 = (P2-P1)*-2.0+(P3-P1)*factor+(P2-P0)*factor;

    Eigen::Vector3d curvePoint;

    for(int i=0;i!=LOD;++i) {
        double t = (double)i/LOD;
        //double it = 1.0-t;
        curvePoint = c3*t*t*t + c2*t*t + c1*t + c0;

        temPos(0,3) = curvePoint(0);
        temPos(1,3) = curvePoint(1);
        temPos(2,3) = curvePoint(2);

        q = getFirstIK(temPos,qf);
        moveJ(q(0),q(1)+M_PI/2.0,-q(2),q(3)+M_PI/2.0,q(4),q(5));
        DisplayPoint( curvePoint(0),curvePoint(1),curvePoint(2));
```

```
}

//the second segment curve
P0<<Points[0][0],Points[0][1],Points[0][2];
P1<<Points[1][0],Points[1][1],Points[1][2];
P2<<Points[2][0],Points[2][1],Points[2][2];
P3<<Points[3][0],Points[3][1],Points[3][2];

factor = 0.5;
c0 = P1;
c1 = (P2-P0)*factor;
c2 = (P2-P1)*3.0-(P3-P1)*factor-(P2-P0)*2.0*factor;
c3 = (P2-P1)*-2.0+(P3-P1)*factor+(P2-P0)*factor;

//Eigen::Vector3d curvePoint;

for(int i=0;i!=LOD;++i) {
    double t = (double)i/LOD;
    //double it = 1.0-t;
    curvePoint = c3*t*t*t + c2*t*t + c1*t + c0;

    temPos(0,3) = curvePoint(0);
    temPos(1,3) = curvePoint(1);
    temPos(2,3) = curvePoint(2);

    q = getFirstIK(temPos,qf);
    moveJ(q(0),q(1)+M_PI/2.0,-q(2),q(3)+M_PI/2.0,q(4),q(5));
    DisplayPoint( curvePoint(0),curvePoint(1),curvePoint(2));

}

//the third segment curve

P0<<Points[1][0],Points[1][1],Points[1][2];
P1<<Points[2][0],Points[2][1],Points[2][2];
```

```
    P2<<Points[3][0],Points[3][1],Points[3][2];
    P3<<2*Points[3][0]-Points[2][0],2*Points[3][1]-Points[2][1],2*Points[3][2]-Points
[2][2];

    factor = 0.3;
    c0 = P1;
    c1 = (P2-P0)*factor;
    c2 = (P2-P1)*3.0-(P3-P1)*factor-(P2-P0)*2.0*factor;
    c3 = (P2-P1)*-2.0+(P3-P1)*factor+(P2-P0)*factor;



    for(int i=0;i!=LOD;++i) {
        double t = (double)i/LOD;
        //double it = 1.0-t;
        curvePoint = c3*t*t*t + c2*t*t + c1*t + c0;
        temPos(0,3) = curvePoint(0);
        temPos(1,3) = curvePoint(1);
        temPos(2,3) = curvePoint(2);

        q = getFirstIK(temPos,qf);
        moveJ(q(0),q(1)+M_PI/2.0,-q(2),q(3)+M_PI/2.0,q(4),q(5));
        DisplayPoint( curvePoint(0),curvePoint(1),curvePoint(2));

    }
}
```

测试效果图如图 6.5 所示。

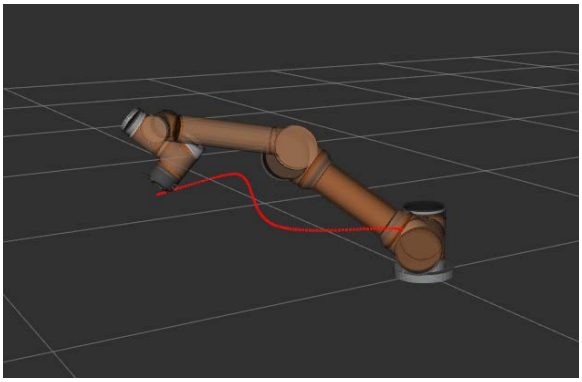


图 6.5 Catmull-Rom 参数化插补算法

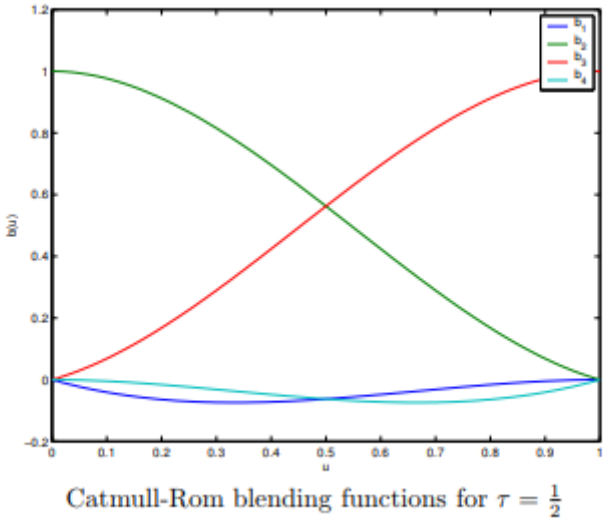
通过 kazam 屏幕录制也可以看到动画效果。

https://github.com/mhuasong/AUBO-Robot-on-ROS/blob/master/video/catmull_rom.mp4

上面代码 Catmull-Rom 算法除了参数化时间轴 t 以及插值点数 LOD 之外，还提供了 factor 系数，这个系数我直接赋值了 0.5，关于这几个参数是如何影响 blending 的，可以从它的系数表达式看到：

$$\begin{aligned}\mathbf{c}_0 &= \mathbf{p}_{i-1}\\ \mathbf{c}_1 &= \tau(\mathbf{p}_i - \mathbf{p}_{i-2})\\ \mathbf{c}_2 &= 3(\mathbf{p}_i - \mathbf{p}_{i-1}) - \tau(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}) - 2\tau(\mathbf{p}_i - \mathbf{p}_{i-2})\\ \mathbf{c}_3 &= -2(\mathbf{p}_i - \mathbf{p}_{i-1}) + \tau(\mathbf{p}_{i+1} - \mathbf{p}_{i-1}) + \tau(\mathbf{p}_i - \mathbf{p}_{i-2})\end{aligned} \tag{6-2}$$

τ 又称有 blending function 作用，blending 这个词在 UR 以及遨博的翻译中都把它称为交融半径，总感觉不是很妥，我也一直没什么好的词去翻译。



Catmull-Rom blending functions for $\tau = \frac{1}{2}$

图 6.6 τ=0.5(程序中用 factor 表示)对曲率系数的影响

有关机械臂的轨迹规划(trajectory planning)的基本实验介绍到这里，目前我们虽然有求速度、加速度，甚至加加速（jerk），但并没有实现控制，后面两种算法甚至在函数里求都没求，原因在于这几个算法，求解速度、加速度、加加速也比较难控制，一般我们可以另外再规划速度、加速度控制，这个留给同学们在实际的应用中去扩展实现。

6.6 本章版权及声明

站内容，但是必须注明来源。

本章内容由闵华松(mhuasong@wust.edu.cn)编写。