



## 实验 5：世界坐标系下的轨迹规划

机器人技术与实践实验报告

课程名称: 机器人技术与实践

小组成员: 金加康 吴必兴 沈学文 钱满亮 赵钰泓

组号: 第三组

指导教师: 周春琳

报告日期: 2025 年 11 月 23 日

## 目录

1 实验任务与分工 .....	4
1.1 实验目的 .....	4
1.2 任务要求 .....	4
1.3 组员分工 .....	4
2 ZJUI 型机械臂 DH 参数与坐标系 .....	4
3 基于速度控制的轨迹规划 .....	5
3.1 结果与分析 .....	5
3.1.1 正方形轨迹 .....	5
3.1.2 圆形轨迹 .....	5
3.1.3 圆锥轨迹 .....	6
3.2 CoppeliaSim 仿真参数设置 .....	7
3.3 理论分析与计算 .....	8
3.3.1 Jacobian 矩阵的定义 .....	8
3.3.2 ZJU-I 型机械臂几何雅可比矩阵表达式 .....	8
3.3.2.1 线速度雅可比矩阵 $J_v$ .....	9
3.3.2.2 角速度雅可比矩阵 $J_\omega$ .....	10
3.3.3 速度层逆解问题 .....	10
3.3.4 三种轨迹的速度定义与计算 .....	10
3.3.4.1 正方形轨迹速度 .....	10
3.3.4.2 圆形轨迹速度 .....	11
3.3.4.3 圆锥轨迹速度 .....	11
3.4 核心代码实现 .....	12
3.4.1 Jacobian 矩阵计算 .....	12
3.4.2 PyBind11 封装模块 .....	15
3.4.3 编译配置 (setup.py) .....	16
3.4.4 运动轨迹控制代码实现 .....	16
4 基于位置控制的轨迹规划 .....	20
4.1 结果与分析 .....	20
4.1.1 正方形轨迹先 .....	20
4.1.2 圆形轨迹 .....	20
4.1.3 圆锥轨迹 .....	21
4.2 理论分析与计算 .....	22
4.2.1 坐标系与运动学 .....	22
4.2.2 五次多项式轨迹规划 .....	23
4.2.3 正方形轨迹规划 .....	23
4.2.4 圆形轨迹规划 .....	23
4.2.5 圆锥轨迹规划 .....	23
4.3 代码实现思路 .....	24

---

4.3.1 总体架构 .....	24
4.3.2 正方形轨迹实现 .....	24
4.3.3 圆形轨迹实现 .....	24
4.3.4 圆锥轨迹实现 .....	25
4.3.5 关键技术细节 .....	25
4.4 核心代码实现 .....	26
4.4.1 逆运动学求解器（自行编写的） .....	26
4.4.2 五次多项式轨迹规划 .....	27
4.4.3 圆锥轨迹的姿态处理 .....	28
4.4.4 最优自旋角搜索 .....	29
5 总结 .....	31

# 实验 5：世界坐标系下的轨迹规划

第 3 组 金加康 吴必兴 沈学文 钱满亮 赵钰泓

## § 1 实验任务与分工

### 1.1 实验目的

- 掌握在笛卡尔空间进行轨迹规划的方法
- 学习控制机械臂末端执行器在世界坐标系下沿指定路径运动
- 理解关节空间与笛卡尔空间之间的转换关系
- 熟练运用逆运动学求解器实现复杂轨迹规划

### 1.2 任务要求

- 正方形轨迹：**控制 ZJU-I 型机械臂末端执行器端点沿着边长为 10cm 的正方形连续移动，移动过程中末端执行器空间姿态保持不变，自定义正方形在笛卡尔空间的位置、末端点的移动速度。
- 圆形轨迹：**控制 ZJU-I 型机械臂末端执行器端点沿着直径为 10cm 的圆连续移动，移动过程中末端执行器空间姿态保持不变，自定义圆在笛卡尔空间的位置、末端点的移动速度。
- 圆锥轨迹：**控制 ZJU-I 型机械臂末端执行器绕着执行器的端点转动，即末端点位于空间圆锥体的顶点上，末端执行器绕点运动时其轴线（机械臂第 6 轴）始终与圆锥体的素线重合，圆锥体的锥角为 60 度，自定义圆锥体在笛卡尔空间中的位置、旋转角速度。
- 提交计算过程报告、仿真工程文件以及仿真结果视频。

### 1.3 组员分工

- 三种轨迹的速度控制实现：吴必兴、沈学文
- 三种轨迹的位置控制实现：金加康、赵钰泓、钱满亮
- 报告撰写：小组成员撰写各自实现部分

## § 2 ZJUI 型机械臂 DH 参数与坐标系

表 1 DH 参数表

关节 $i$	$\alpha_{i-1}$ (rad)	$a_{i-1}$ (mm)	$d_i$ (mm)	$\theta_i$ (rad)	初始偏 置	关节类 型
1	0	0	230	$\theta_1$	0	旋转
2	$-\frac{\pi}{2}$	0	0	$\theta_2$	$-\frac{\pi}{2}$	旋转
3	0	185	0	$\theta_3$	0	旋转
4	0	170	23	$\theta_4$	$\frac{\pi}{2}$	旋转
5	$\frac{\pi}{2}$	0	77	$\theta_5$	$\frac{\pi}{2}$	旋转
6	$\frac{\pi}{2}$	0	85.5	$\theta_6$	0	旋转

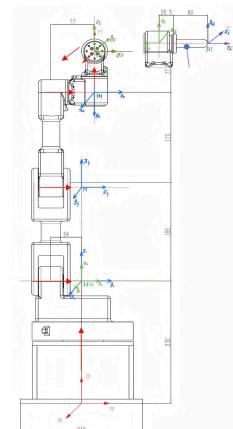


图 1 机械臂坐标系示意图

## § 3 基于速度控制的轨迹规划

### 3.1 结果与分析

#### 3.1.1 正方形轨迹

下图展示了机械臂末端执行器沿正方形轨迹运动的结果。使用速度控制方法，末端执行器准确地沿着边长为10cm的正方形路径运动，红色轨迹线清晰地显示了运动路径：

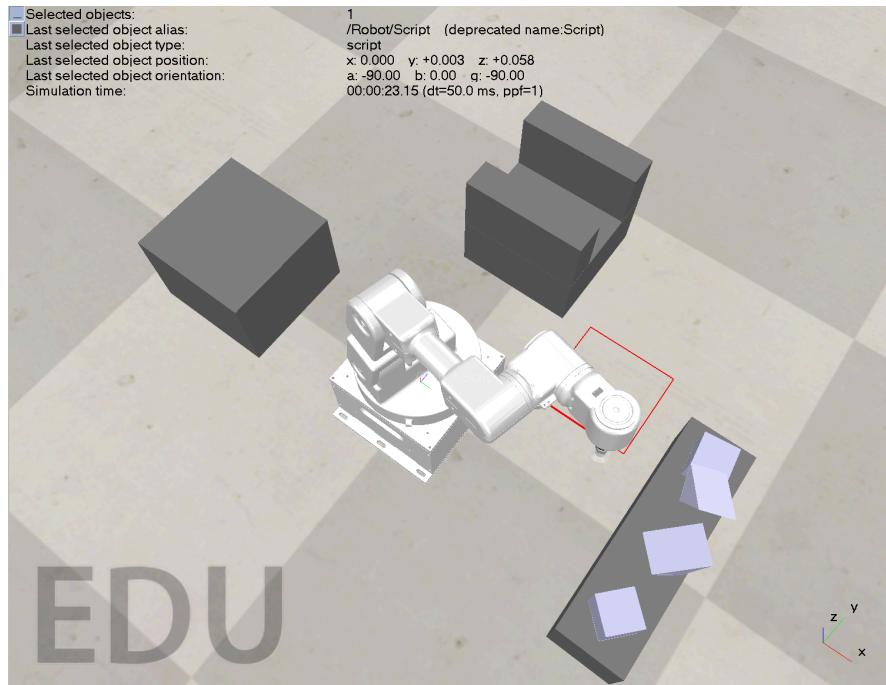


图 2 正方形轨迹运动结果 | 速度控制

下图为正方形轨迹运动时的关节运动曲线：

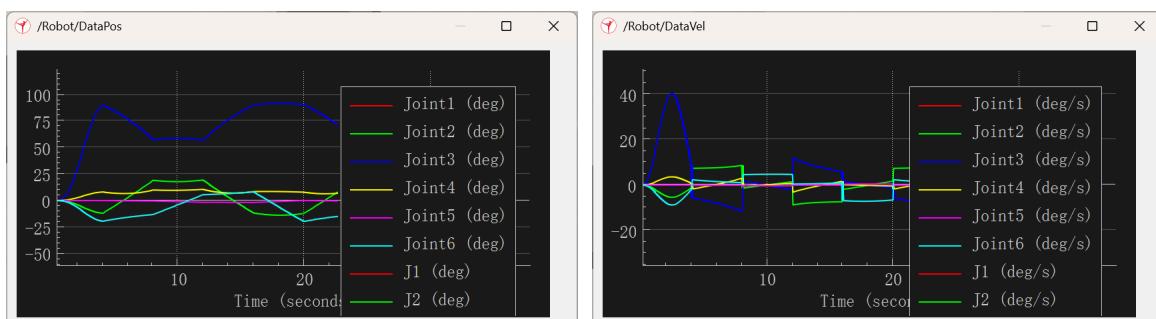


图 3 正方形轨迹的关节运动曲线 | 速度控制

正方形轨迹的速度控制分析：

- 速度直接控制：**与位置控制不同，关节速度是直接计算并控制的，通过 Jacobian 矩阵从末端速度实时映射得到。
- 较好的实时性：**速度控制不需要预先规划整条轨迹，而是每个控制周期实时计算，因此具有更好的实时性。

#### 3.1.2 圆形轨迹

下图展示了机械臂末端执行器沿圆形轨迹运动的结果。红色圆圈清晰地显示了末端执行器的运动轨迹，直径约为10cm：

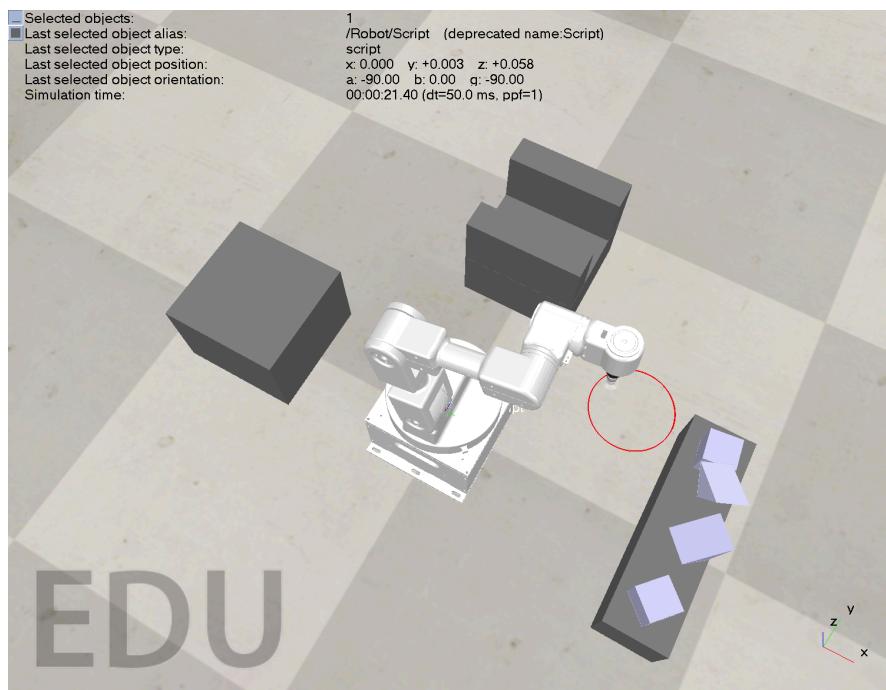


图 4 圆形轨迹运动结果 | 速度控制

下图为圆形轨迹运动时的关节运动曲线：

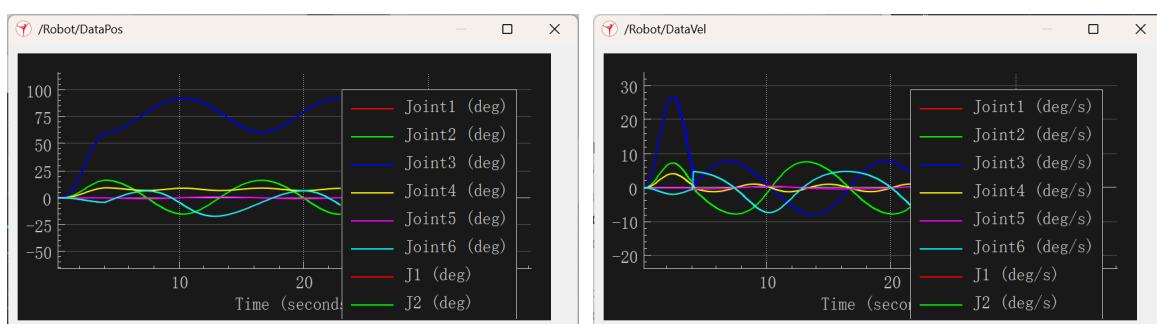


图 5 圆形轨迹的关节运动曲线 | 速度控制

圆形轨迹的速度控制分析：

- 控制方法直观：**圆形轨迹的速度是连续变化的，适合速度控制方法。与位置控制相比，直接对圆周运动的参数方程求导即可得到速度。
- 平滑周期性：**各关节的位置和速度曲线都呈现周期性，说明速度控制能够很好地跟踪连续变化的轨迹。
- 速度恒定：**末端执行器的线速度大小保持恒定（0.025 m/s），仅方向沿切线方向变化。

### 3.1.3 圆锥轨迹

下图展示了机械臂末端执行器绕固定点作圆锥运动的结果。从图中可以看出，末端执行器的轴线与圆锥素线重合，末端点保持固定：

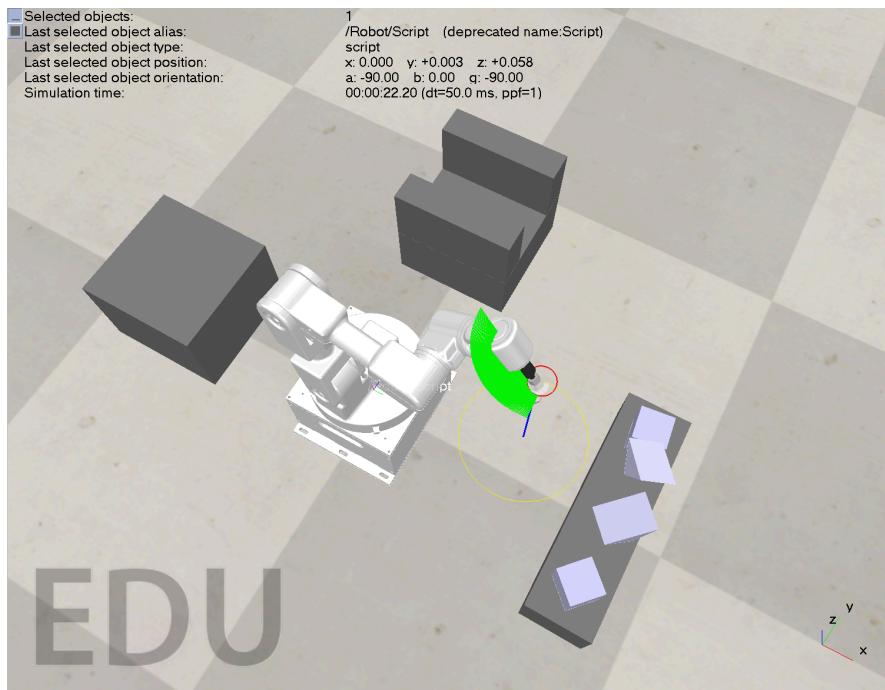


图 6 圆锥轨迹运动结果 | 速度控制

下图为圆锥轨迹运动时的关节运动曲线：

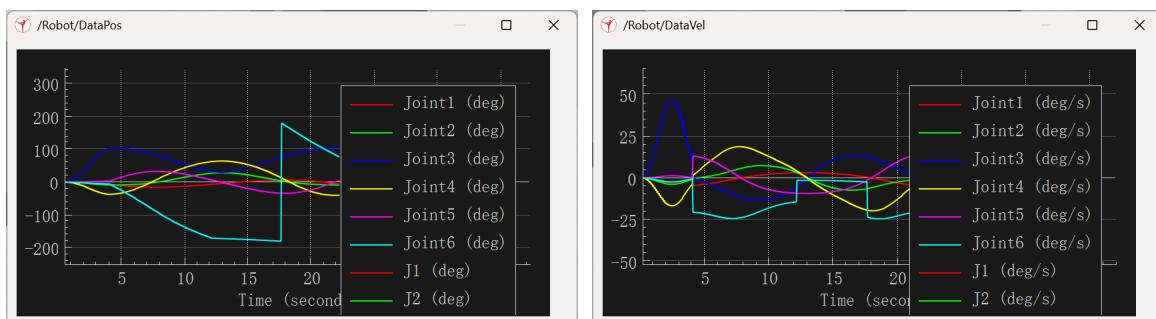


图 7 圆锥轨迹的关节运动曲线 | 速度控制

圆锥轨迹的速度控制分析：

1. 纯姿态控制：圆锥运动仅涉及姿态变化（末端点位置固定），因此线速度为零，仅有角速度。
2. 实现简洁：速度控制下，圆锥运动的实现只需给定一个恒定的角速度向量  $\omega = [0, 0, 20^\circ/s]^T$ ，无需复杂的姿态规划。
3. 周期性变化：各关节速度呈现周期性变化，周期约为 18 秒 ( $360^\circ \div 20^\circ/s$ )，与给定的旋转角速度一致。

### 3.2 CoppeliaSim 仿真参数设置

1. 首先通过 **Simulation->Simulation settings** 进入仿真设置界面，通过修改 **Gravity vector** 为全 0 取消重力
2. 再依次双击各个 **Joint** 关节图标，进入 **Scene Object Properties** 设置界面，选择 **Dynamic mode**，再点击 **Dynamic properties dialog** 进入 **Joint Dynamic Properties** 设置界面，将 **Control mode** 设置为 **Velocity** 速度控制模式

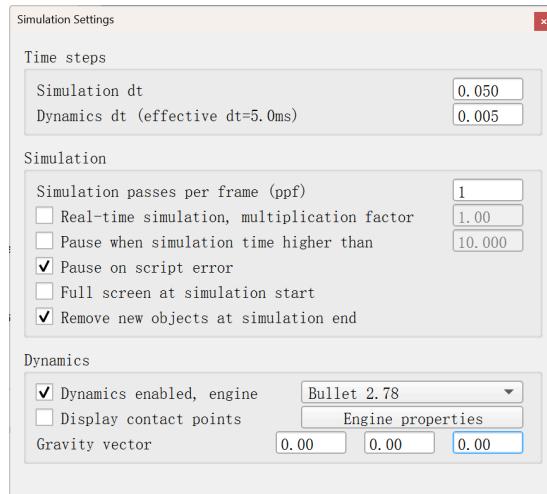


图 8 修改 Gravity vector 取消重力

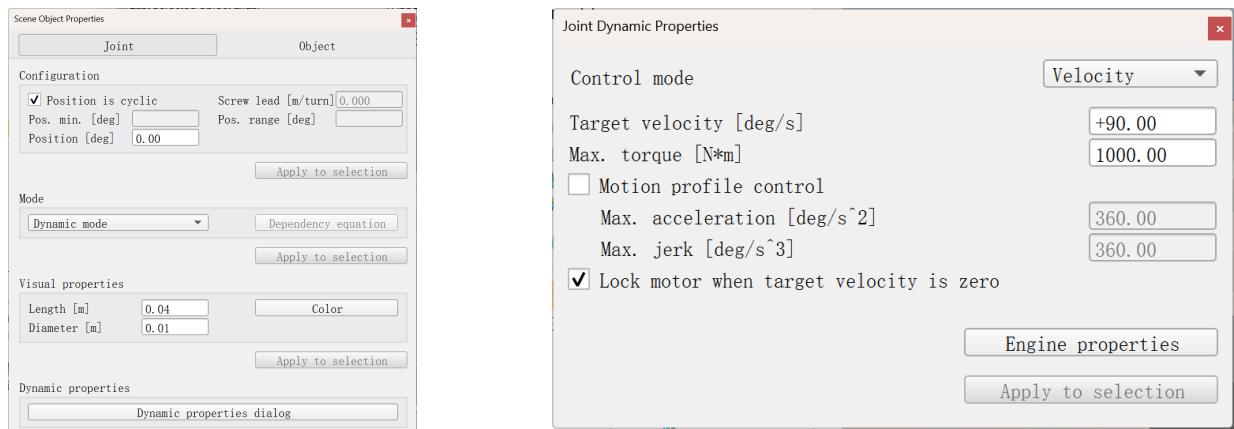


图 9 设置为速度控制模式

### 3.3 理论分析与计算

#### 3.3.1 Jacobian 矩阵的定义

Jacobian 矩阵建立了机械臂关节速度与末端速度末端速度之间的线性映射关系：

$$\dot{x} = J(q)\dot{q} \quad (1)$$

Jacobian 矩阵的结构为：

$$J(q) = \begin{pmatrix} J_v \\ J_\omega \end{pmatrix} = \begin{pmatrix} J_{v_x1} & J_{v_x2} & J_{v_x3} & J_{v_x4} & J_{v_x5} & J_{v_x6} \\ J_{v_y1} & J_{v_y2} & J_{v_y3} & J_{v_y4} & J_{v_y5} & J_{v_y6} \\ J_{v_z1} & J_{v_z2} & J_{v_z3} & J_{v_z4} & J_{v_z5} & J_{v_z6} \\ J_{\omega_x1} & J_{\omega_x2} & J_{\omega_x3} & J_{\omega_x4} & J_{\omega_x5} & J_{\omega_x6} \\ J_{\omega_y1} & J_{\omega_y2} & J_{\omega_y3} & J_{\omega_y4} & J_{\omega_y5} & J_{\omega_y6} \\ J_{\omega_z1} & J_{\omega_z2} & J_{\omega_z3} & J_{\omega_z4} & J_{\omega_z5} & J_{\omega_z6} \end{pmatrix} \quad (2)$$

上半部分  $J_v$  描述线速度映射，下半部分  $J_\omega$  描述角速度映射。

#### 3.3.2 ZJU-I 型机械臂几何雅可比矩阵表达式

结合上一次实验报告的推导结果，ZJU-I 型机械臂的几何雅可比矩阵可以表示为：

$$J(q) = \begin{pmatrix} J_v \\ J_\omega \end{pmatrix} = \begin{pmatrix} J_v^1 & J_v^2 & J_v^3 & J_v^4 & J_v^5 & J_v^6 \\ J_\omega^1 & J_\omega^2 & J_\omega^3 & J_\omega^4 & J_\omega^5 & J_\omega^6 \end{pmatrix} \quad (3)$$

其中， $J_v \in \mathbb{R}^{3 \times 6}$  为线速度雅可比， $J_\omega \in \mathbb{R}^{3 \times 6}$  为角速度雅可比。为简化表达，记  $\Sigma = q_2 + q_3 + q_4$ 。

### 3.3.2.1 线速度雅可比矩阵 $J_v$

$$J_v = \begin{pmatrix} -y_e & \cos(q_1)J_v^{(2)\#} & \cos(q_1)J_v^{(3)\#} & \cos(q_1)J_v^{(4)\#} & J_v^{(5)_x} & 0 \\ x_e & \sin(q_1)J_v^{(2)\#} & \sin(q_1)J_v^{(3)\#} & \sin(q_1)J_v^{(4)\#} & J_v^{(5)_y} & 0 \\ 0 & J_v^{(2)_z} & J_v^{(3)_z} & J_v^{(4)_z} & J_v^{(5)_z} & 0 \end{pmatrix} \quad (4)$$

其中各项具体表达式为：

第 1 列：

$$J_v^1 = [-y_e, x_e, 0]^T \quad (5)$$

$$\begin{aligned} x_e = & -\frac{171}{2} \sin(q_1) \sin(q_5) - 23 \sin(q_1) + 185 \sin(q_2) \cos(q_1) \\ & + 170 \sin(q_2 + q_3) \cos(q_1) + 77 \sin(\Sigma) \cos(q_1) \\ & + \frac{171}{2} \cos(q_1) \cos(q_5) \cos(\Sigma) \end{aligned} \quad (6)$$

$$\begin{aligned} y_e = & 185 \sin(q_1) \sin(q_2) + 170 \sin(q_1) \sin(q_2 + q_3) + 77 \sin(q_1) \sin(\Sigma) \\ & + \frac{171}{2} \sin(q_1) \cos(q_5) \cos(\Sigma) + \frac{171}{2} \sin(q_5) \cos(q_1) + 23 \cos(q_1) \end{aligned} \quad (7)$$

第 2 列：

$$\begin{aligned} J_v^{(2)\#} = & -\frac{171}{2} \sin(\Sigma) \cos(q_5) + 185 \cos(q_2) \\ & + 170 \cos(q_2 + q_3) + 77 \cos(\Sigma) \end{aligned} \quad (8)$$

$$\begin{aligned} J_v^{(2)_z} = & -185 \sin(q_2) - 170 \sin(q_2 + q_3) - 77 \sin(\Sigma) \\ & - \frac{171}{2} \cos(q_5) \cos(\Sigma) \end{aligned} \quad (9)$$

第 3 列：

$$J_v^{(3)\#} = -\frac{171}{2} \sin(\Sigma) \cos(q_5) + 170 \cos(q_2 + q_3) + 77 \cos(\Sigma) \quad (10)$$

$$J_v^{(3)_z} = -170 \sin(q_2 + q_3) - 77 \sin(\Sigma) - \frac{171}{2} \cos(q_5) \cos(\Sigma) \quad (11)$$

第 4 列：

$$J_v^{(4)\#} = -\frac{171}{2} \sin(\Sigma) \cos(q_5) + 77 \cos(\Sigma) \quad (12)$$

$$J_v^{(4)_z} = -77 \sin(\Sigma) - \frac{171}{2} \cos(q_5) \cos(\Sigma) \quad (13)$$

第 5 列：

$$J_v^{(5)x} = -\frac{171}{2} \sin(q_1) \cos(q_5) - \frac{171}{2} \sin(q_5) \cos(q_1) \cos(\Sigma) \quad (14)$$

$$J_v^{(5)y} = -\frac{171}{2} \sin(q_1) \sin(q_5) \cos(\Sigma) + \frac{171}{2} \cos(q_1) \cos(q_5) \quad (15)$$

$$J_v^{(5)z} = \frac{171}{2} \sin(q_5) \sin(\Sigma) \quad (16)$$

第 6 列：末端关节轴通过末端点，线速度贡献为零。

### 3.3.2.2 角速度雅可比矩阵 $J_\omega$

$$J_\omega = \begin{pmatrix} 0 & -\sin(q_1) & -\sin(q_1) & -\sin(q_1) & \sin(\Sigma) \cos(q_1) & J_\omega^{(6)x} \\ 0 & \cos(q_1) & \cos(q_1) & \cos(q_1) & \sin(q_1) \sin(\Sigma) & J_\omega^{(6)y} \\ 1 & 0 & 0 & 0 & \cos(\Sigma) & J_\omega^{(6)z} \end{pmatrix} \quad (17)$$

其中第 6 列各分量为：

$$J_\omega^{(6)x} = -\sin(q_1) \sin(q_5) + \cos(q_1) \cos(q_5) \cos(\Sigma) \quad (18)$$

$$J_\omega^{(6)y} = \sin(q_1) \cos(q_5) \cos(\Sigma) + \sin(q_5) \cos(q_1) \quad (19)$$

$$J_\omega^{(6)z} = -\sin(\Sigma) \cos(q_5) \quad (20)$$

### 3.3.3 速度层逆解问题

**问题描述：**已知期望末端速度  $\dot{x}_{des}$ ，求解关节速度  $\dot{q}$ 。

最直接的想法是对方程 式 1 两边求逆：

$$\dot{q} = \mathbf{J}^{-1}(\mathbf{q}) \dot{x} \quad (21)$$

然而，这个方法存在以下问题：

1. **奇异性问题：**当机械臂处于奇异构型时， $\mathbf{J}$  的行列式接近零， $\mathbf{J}^{-1}$  趋于无穷大，导致关节速度剧烈增大，实际不可行。
2. **数值不稳定：**即使不完全奇异， $\mathbf{J}$  接近奇异时，直接求逆的数值误差也会被放大。

为了解决奇异性问题，采用**阻尼最小二乘法**（也称为 Levenberg-Marquardt 方法）：

$$\dot{q} = \mathbf{J}^T (\mathbf{J} \mathbf{J}^T + \lambda^2 \mathbf{I}_6)^{-1} \dot{x} \quad (22)$$

其中  $\lambda > 0$  是阻尼系数，通常取较小值（如 0.01-0.1）。

### 3.3.4 三种轨迹的速度定义与计算

#### 3.3.4.1 正方形轨迹速度

正方形轨迹由四条边组成，每条边是一条直线段。在速度控制下，需要实时计算末端在当前边上的运动速度。

**速度计算：**

1. 确定当前时刻所在的边（第  $i$  条边）：

$$i = \left\lfloor t \bmod \frac{T_{\text{total}}}{T_{\text{edge}}} \right\rfloor \in \{0, 1, 2, 3\} \quad (23)$$

其中  $T_{\text{total}}$  是完成一圈的总时间， $T_{\text{edge}}$  是每条边的运动时间。

2. 计算边的方向单位向量：

$$\mathbf{u}_i = \frac{\mathbf{p}_{i+1} - \mathbf{p}_i}{\|\mathbf{p}_{i+1} - \mathbf{p}_i\|} \quad (24)$$

其中  $\mathbf{p}_i$  和  $\mathbf{p}_{i+1}$  是第  $i$  条边的起点和终点。

3. 线速度为恒定速度沿边方向：

$$\mathbf{v}(t) = v_{\text{lin}} \mathbf{u}_i \quad (25)$$

其中  $v_{\text{lin}} = 0.025 \text{ m/s}$  是给定的线速度大小。

4. 角速度为零（姿态保持不变）：

$$\boldsymbol{\omega}(t) = \mathbf{0} \quad (26)$$

### 3.3.4.2 圆形轨迹速度

圆形轨迹是最适合速度控制的路径之一，因为其速度是连续变化的。

轨迹参数方程：

圆心为  $\mathbf{p}_c = [x_c, y_c, z_c]^T$ ，半径为  $R$ ，在水平面内运动：

$$\mathbf{p}(t) = \begin{pmatrix} x_c + R \cos(\omega t) \\ y_c + R \sin(\omega t) \\ z_c \end{pmatrix} \quad (27)$$

其中  $\omega = \frac{v_{\text{lin}}}{R}$  是角频率。

速度推导：

对位置方程对时间求导：

$$\mathbf{v}(t) = \frac{d}{dt} \mathbf{p}(t) = \begin{pmatrix} -R\omega \sin(\omega t) \\ R\omega \cos(\omega t) \\ 0 \end{pmatrix} = v_{\text{lin}} \begin{pmatrix} -\sin(\omega t) \\ \cos(\omega t) \\ 0 \end{pmatrix} \quad (28)$$

可以验证速度大小恒定：

$$\|\mathbf{v}(t)\| = v_{\text{lin}} \sqrt{\sin^2(\omega t) + \cos^2(\omega t)} = v_{\text{lin}} \quad (29)$$

速度方向始终与位置向量  $\mathbf{p}(t) - \mathbf{p}_c$  垂直，即沿圆的切线方向。

角速度仍为零：

$$\boldsymbol{\omega}(t) = \mathbf{0} \quad (30)$$

### 3.3.4.3 圆锥轨迹速度

圆锥运动是纯姿态变化的任务，末端点位置固定，仅姿态绕圆锥轴旋转。

运动描述：

1. 末端点固定在圆锥顶点：

$$\mathbf{p}(t) = \mathbf{p}_{\text{apex}} = \text{常量} \quad (31)$$

2. 末端执行器轴线在圆锥面上旋转，与圆锥轴夹角为半锥角  $\alpha_{\text{cone}} = 30^\circ$

3. 轴线绕圆锥轴（取为 z 轴）以角速度  $\omega_z$  旋转

速度定义：

1. 线速度为零：

$$\mathbf{v}(t) = \mathbf{0} \quad (32)$$

2. 角速度为绕圆锥轴的旋转：

$$\boldsymbol{\omega}(t) = \omega_z \mathbf{e}_z = \begin{pmatrix} 0 \\ 0 \\ \omega_z \end{pmatrix} \quad (33)$$

其中  $\omega_z = 20^\circ/\text{s} = 0.349 \text{ rad/s}$ ,  $\mathbf{e}_z$  是圆锥轴方向（通常取为  $[0, 0, -1]^T$  或  $[0, 0, 1]^T$ ）。

由于线速度为零，速度映射方程简化为：

$$\begin{pmatrix} \mathbf{0} \\ \boldsymbol{\omega} \end{pmatrix} = \begin{pmatrix} \mathbf{J}_v \\ \mathbf{J}_\omega \end{pmatrix} \dot{\mathbf{q}} \Rightarrow \boldsymbol{\omega} = \mathbf{J}_\omega \dot{\mathbf{q}} \quad (34)$$

## 3.4 核心代码实现

### 3.4.1 Jacobian 矩阵计算

基于 DH 参数推导的解析公式计算  $6 \times 6$  几何 Jacobian 矩阵，使用 C++ 实现。

```

1 #include "jacobi_core.h"                                     C++
2 #include <cmath>
3
4 // Length parameters (185, 170, 77, 23, 171/2, 230) in mm units
5
6 void jacobi_zjui_core(const double q[6], double J[36]) {
7     const double q1 = q[0];
8     const double q2 = q[1];
9     const double q3 = q[2];
10    const double q4 = q[3];
11    const double q5 = q[4];
12    // q6 not used in geometric Jacobian
13
14    const double Sigma = q2 + q3 + q4;
15
16    const double s1 = std::sin(q1);
17    const double c1 = std::cos(q1);
18    const double s2 = std::sin(q2);
19    const double c2 = std::cos(q2);
20    const double s23 = std::sin(q2 + q3);
21    const double c23 = std::cos(q2 + q3);
22    const double sSig = std::sin(Sigma);
23    const double cSig = std::cos(Sigma);
24    const double s5 = std::sin(q5);
25    const double c5 = std::cos(q5);
26
27    const double k = 171.0 / 2.0;
28
29    // Column 1

```

```
30     const double J1vx = -185.0 * s1 * s2 - 170.0 * s1 * s23 - 77.0 * s1 * sSig -
31                     k * s1 * c5 * cSig - k * s5 * c1 - 23.0 * c1;
32
33     const double J1vy = -k * s1 * s5 - 23.0 * s1 + 185.0 * s2 * c1 +
34                     170.0 * s23 * c1 + 77.0 * sSig * c1 + k * c1 * c5 * cSig;
35
36     // Linear velocity
37     J[0 * 6 + 0] = J1vx;
38     J[1 * 6 + 0] = J1vy;
39     J[2 * 6 + 0] = 0.0;
40     // Angular velocity: z1 = [0,0,1]^T
41     J[3 * 6 + 0] = 0.0;
42     J[4 * 6 + 0] = 0.0;
43     J[5 * 6 + 0] = 1.0;
44
45     // Column 2
46     const double common2 =
47         -k * sSig * c5 + 185.0 * c2 + 170.0 * c23 + 77.0 * cSig;
48
49     const double J2vx = c1 * common2;
50     const double J2vy = s1 * common2;
51     const double J2vz = -185.0 * s2 - 170.0 * s23 - 77.0 * sSig - k * c5 * cSig;
52
53     J[0 * 6 + 1] = J2vx;
54     J[1 * 6 + 1] = J2vy;
55     J[2 * 6 + 1] = J2vz;
56     // Angular velocity: z2 = [-sin(q1), cos(q1), 0]^T
57     J[3 * 6 + 1] = -s1;
58     J[4 * 6 + 1] = c1;
59     J[5 * 6 + 1] = 0.0;
60
61     // Column 3
62     const double common3 = -k * sSig * c5 + 170.0 * c23 + 77.0 * cSig;
63
64     const double J3vx = c1 * common3;
65     const double J3vy = s1 * common3;
66     const double J3vz = -170.0 * s23 - 77.0 * sSig - k * c5 * cSig;
67
68     J[0 * 6 + 2] = J3vx;
69     J[1 * 6 + 2] = J3vy;
70     J[2 * 6 + 2] = J3vz;
71     // Angular velocity: z3 = z2
72     J[3 * 6 + 2] = -s1;
73     J[4 * 6 + 2] = c1;
74     J[5 * 6 + 2] = 0.0;
75
76     // Column 4
77     const double common4 = -k * sSig * c5 + 77.0 * cSig;
```

```
78
79     const double J4vx = c1 * common4;
80     const double J4vy = s1 * common4;
81     const double J4vz = -77.0 * sSig - k * c5 * cSig;
82
83     J[0 * 6 + 3] = J4vx;
84     J[1 * 6 + 3] = J4vy;
85     J[2 * 6 + 3] = J4vz;
86     // Angular velocity: z4 = z2
87     J[3 * 6 + 3] = -s1;
88     J[4 * 6 + 3] = c1;
89     J[5 * 6 + 3] = 0.0;
90
91     // Column 5
92     const double J5vx = -k * s1 * c5 - k * s5 * c1 * cSig;
93
94     const double J5vy = -k * s1 * s5 * cSig + k * c1 * c5;
95
96     const double J5vz = k * s5 * sSig;
97
98     J[0 * 6 + 4] = J5vx;
99     J[1 * 6 + 4] = J5vy;
100    J[2 * 6 + 4] = J5vz;
101    // Angular velocity: z5
102    J[3 * 6 + 4] = sSig * c1;
103    J[4 * 6 + 4] = s1 * sSig;
104    J[5 * 6 + 4] = cSig;
105
106    // Column 6
107    // Linear velocity is zero
108    J[0 * 6 + 5] = 0.0;
109    J[1 * 6 + 5] = 0.0;
110    J[2 * 6 + 5] = 0.0;
111
112    // Angular velocity: z6
113    const double J6wx = -s1 * s5 + c1 * c5 * cSig;
114    const double J6wy = s1 * c5 * cSig + s5 * c1;
115    const double J6wz = -sSig * c5;
116
117    J[3 * 6 + 5] = J6wx;
118    J[4 * 6 + 5] = J6wy;
119    J[5 * 6 + 5] = J6wz;
120
121    // Convert linear velocity from mm to m
122    const double scale = 1e-3;
123    for (int col = 0; col < 6; ++col) {
124        for (int row = 0; row < 3; ++row) {
125            J[row * 6 + col] *= scale;
```

```
126     }
127 }
128 }
```

代码解析：

1. **DH 参数**: 185, 170, 77, 23 等参数可参见第二章节中的 DH 参数表。
2. **累加角度 $\Sigma$** : 机械臂的关节 2、3、4 是串联的旋转关节，它们的累加效果体现在  $\Sigma = q_2 + q_3 + q_4$ 。
3. **预计算三角函数**:  $s1 = \sin(q_1)$ ,  $c1 = \cos(q_1)$  等，避免重复调用  $\sin/\cos$  函数，提高效率。
4. **分列计算**: 每一列对应一个关节的贡献。上半部分（前 3 行）是线速度  $\mathbf{J}_v$ ，下半部分（后 3 行）是角速度  $\mathbf{J}_\omega$ 。
5.  **$z$  轴方向**:
  - (1) 关节 1 绕世界坐标系的  $z$  轴，故  $\mathbf{z}_1 = [0, 0, 1]^T$
  - (2) 关节 2、3、4 绕经过关节 1 旋转后的  $y$  轴（在世界坐标系中是  $[-\sin(q_1), \cos(q_1), 0]^T$ ）
  - (3) 关节 5、6 的轴线方向更复杂，需要通过完整运动学推导
6. **单位转换**: 推导时使用 mm 作为长度单位（与 DH 参数一致），最后转换为 m，以匹配 SI 单位制。

### 3.4.2 PyBind11 封装模块

使用 PyBind11 对 C++ 代码进行封装

```
1 #include "jacobi_core.h"                                     C++
2 #include <pybind11/numpy.h>
3 #include <pybind11/pybind11.h>
4
5 namespace py = pybind11;
6
7 // Python wrapper for Jacobian computation
8 py::array_t<double>
9 jacobi_py(py::array_t<double, py::array::c_style | py::array::forcecast> q_in) {
10    if (q_in.size() != 6)
11        throw std::runtime_error("q must be length 6");
12
13    // Extract joint angles from numpy array
14    double q[6];
15    auto buf = q_in.request();
16    double *qptr = static_cast<double *>(buf.ptr);
17    for (int i = 0; i < 6; ++i)
18        q[i] = qptr[i];
19
20    // Compute Jacobian
21    double J[36];
22    jacobi_zjui_core(q, J);
23
24    // Return 6x6 numpy array
25    auto result = py::array_t<double>({6, 6});
```

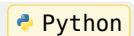
```

26     auto rbuf = result.request();
27     double *rptr = static_cast<double *>(rbuf.ptr);
28     for (int i = 0; i < 36; ++i)
29         rptr[i] = J[i];
30
31     return result;
32 }
33
34 PYBIND11_MODULE(jacobi_zjui, m) {
35     m.doc() = "Geometric Jacobian for ZJU-I arm";
36     m.def("jacobi", &jacobi_py, "Compute 6x6 Jacobian J(q)");
37 }
```

### 3.4.3 编译配置 (setup.py)

```

1  from setuptools import setup
2  from pybind11.setup_helpers import Pybind11Extension, build_ext
3
4  # Extension module configuration
5  ext_modules = [
6      Pybind11Extension(
7          "jacobi_zjui",
8          ["jacobi_zjui_module.cpp", "jacobi_core.cpp"],
9      ),
10 ]
11
12 # Package setup
13 setup(
14     name="jacobi_zjui",
15     version="0.1",
16     ext_modules=ext_modules,
17     cmdclass={"build_ext": build_ext},
18 )
```



编译命令：

```

1 # 安装pybind11
2 pip install pybind11
3
4 # 编译生成jacobi_zjui.pyd
5 python setup.py build_ext --inplace
```



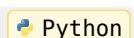
编译成功后，会在当前目录生成 `jacobi_zjui.cp310-win_amd64.pyd` 可以直接在 Python 中导入。

### 3.4.4 运动轨迹控制代码实现

这是速度控制的主要逻辑，实现了从期望速度到关节速度的转换。

```

1  def sysCall_actuation():
2
3      t = sim.getSimulationTime()
```



```
4
5      # Initial positioning
6      if t < INIT_DURATION:
7          q_current = np.array([sim.getJointPosition(h) for h in self.jointHandles])
8
9          # Quintic polynomial interpolation for smooth motion
10         tau = t / INIT_DURATION
11         s = 10*tau**3 - 15*tau**4 + 6*tau**5
12         q_desired = self.q0 + s * (self.q_start - self.q0)
13
14         # Proportional control with velocity limiting
15         position_error = q_desired - q_current
16         qdot = POSITION_GAIN * position_error
17
18         # Apply velocity limits
19         for i in range(6):
20             if abs(qdot[i]) > self.Vel_limits[i]:
21                 qdot[i] = np.sign(qdot[i]) * self.Vel_limits[i] * 0.5
22
23         # Send velocity commands
24         for i in range(6):
25             sim.setJointTargetVelocity(self.jointHandles[i], float(qdot[i]))
26
27     return
28
29     # Initialize trajectory control
30     if not self.velocityModeEnabled:
31         self.velocityModeEnabled = True
32         q_before_switch = np.array([sim.getJointPosition(h) for h in self.jointHandles])
33         self.last_q = q_before_switch.copy()
34         self.sim_dt = sim.getSimulationTimeStep()
35
36     # Trajectory control
37     if not JACOBI_OK:
38         sim.pauseSimulation()
39         return
40
41     t2 = t - INIT_DURATION
42     q = np.array([sim.getJointPosition(h) for h in self.jointHandles])
43
44     # Compute desired end-effector velocity
45     if TASK_MODE == 1:
46         # Square
47         tau = t2 % self.squarePeriod
48         idx = min(int(tau // self.squareEdgeTime), 3)
49         p1 = self.squareCorners[idx]
50         p2 = self.squareCorners[(idx+1) % 4]
51         v = SQUARE_VELOCITY * (p2 - p1) / np.linalg.norm(p2 - p1)
```

```
52     omega = np.zeros(3)
53
54     elif TASK_MODE == 2:
55         # Circle
56         phi = self.circle0omega * t2
57         v = np.array([
58             -CIRCLE_RADIUS * self.circle0omega * np.sin(phi),
59             CIRCLE_RADIUS * self.circle0omega * np.cos(phi),
60             0
61         ])
62         omega = np.zeros(3)
63
64     else: # TASK_MODE == 3
65         # Cone orientation
66         v = np.zeros(3)
67         omega = np.array([0.0, 0.0, CONE_ANGULAR_VEL])
68
69         # Visualize end effector axis during cone motion
70         if self.cone_vis_enabled:
71             try:
72                 tip_pos_current = np.array(sim.getObjectPosition(self.tipHandle, -1))
73                 tip_matrix = sim.getObjectMatrix(self.tipHandle, sim.handle_world)
74                 axis_z = np.array([tip_matrix[2], tip_matrix[6], tip_matrix[10]])
75                 axis_length = 0.12
76                 line_start = tip_pos_current
77                 line_end = tip_pos_current + axis_length * axis_z
78                 sim.addDrawingObjectItem(self.endAxisDrawing, list(line_start) +
79                                         list(line_end))
80             except:
81                 pass
82
83     # Combine linear and angular velocity
84     xdot = np.concatenate([v, omega])
85
86     # Jacobian-based inverse kinematics
87     J = np.array(jacobi_zjui.jacobi(q), dtype=float)
88     JT = J.T
89     JJT = J @ JT
90     qdot = JT @ np.linalg.solve(JJT + (DAMPING_FACTOR**2) * np.eye(6), xdot)
91
92     # Apply joint velocity limits
93     for i in range(6):
94         if abs(qdot[i]) > self.Vel_limits[i]:
95             qdot[i] = np.sign(qdot[i]) * self.Vel_limits[i] * 0.9
96
97     # Soft joint limit protection
98     limit_margin = 10.0 * np.pi / 180
99     for i in range(6):
```

```
99         if q[i] < self.Joint_limits[i,0] + limit_margin and qdot[i] < 0:
100             qdot[i] *= 0.1
101         elif q[i] > self.Joint_limits[i,1] - limit_margin and qdot[i] > 0:
102             qdot[i] *= 0.1
103
104     # Send joint velocity commands
105     for i in range(6):
106         sim.setJointTargetVelocity(self.jointHandles[i], float(qdot[i]))
107
108     # Trajectory visualization
109     if self.trajectory_enabled and self.tipHandle is not None:
110         tip_pos = sim.getObjectPosition(self.tipHandle, -1)
111         sim.addDrawingObjectItem(self.drawingObject, tip_pos)
```

代码解析：

### 1. 初始定位轨迹规划

- (1) 仿真开始时关节角度全为零，直接启动速度控制可能偏离轨迹，所以需要在开始阶段加入一个初始定位，以避免轨迹偏离。
- (2) 使用五次多项式进行位置规划，插值函数为 $s(\tau) = 10\tau^3 - 15\tau^4 + 6\tau^5$ ，其中 $\tau \in [0, 1]$ 是归一化时间，这保证了 $s(0) = 0, s(1) = 1, s'(0) = s'(1) = 0$ ，即起点和终点速度为零
- (3) **比例控制：**加入比例控制，产生跟踪速度消除位置误差。公式为 $\dot{q} = K_p(q_{des} - q)$ ，增益 $K_p = 2.0$ 是经验值

### 2. 期望速度计算

- (1) **正方形：**根据时间计算当前所在边，沿边方向给定恒定速度
- (2) **圆形：**直接使用圆周运动的参数方程导数
- (3) **圆锥：**仅给定角速度，线速度为零

### 3. 阻尼最小二乘求解

- (1) **实现原理：**  $\dot{q} = J^T(JJ^T + \lambda^2 I)^{-1}\dot{x}$ ，使用此公式来求逆从而进行逆运动学求解
- (2) **实现方式：** 使用`np.linalg.solve`求解线性方程组 $(JJ^T + \lambda^2 I)z = \dot{x}$ ，然后计算 $\dot{q} = J^Tz$ ，比直接求逆更稳定
- (3) **阻尼系数：** 阻尼项 $\lambda^2 I$ 防止了雅可比矩阵接近奇异时的数值不稳定

## § 4 基于位置控制的轨迹规划

### 4.1 结果与分析

#### 4.1.1 正方形轨迹先

下图展示了机械臂末端执行器沿正方形轨迹运动的结果。从仿真截图可以看出，末端执行器准确地沿着边长为 10cm 的正方形路径运动，绿色轨迹线清晰地显示了运动路径：

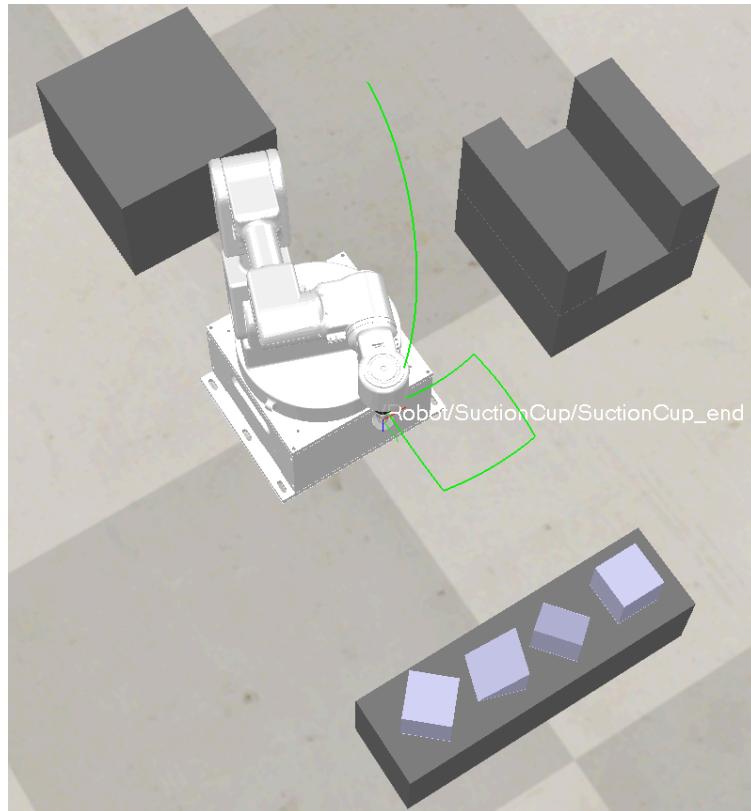


图 10 正方形轨迹运动结果 | 位置控制

下图为机械臂运动时各关节的位置、速度和加速度变化曲线：

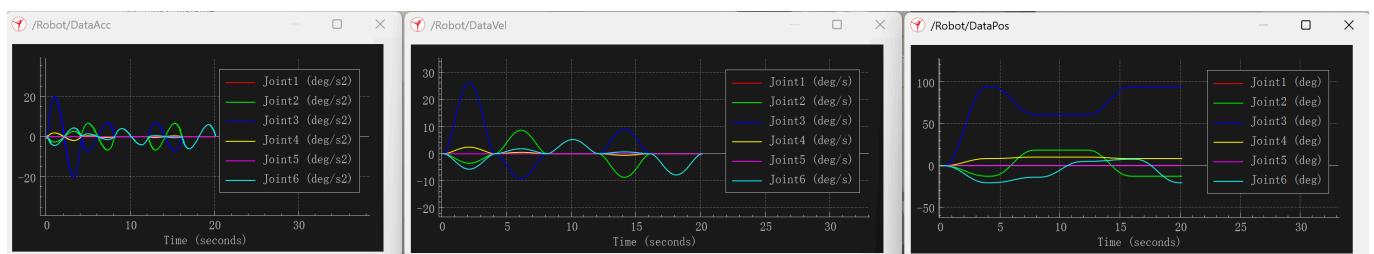


图 11 正方形轨迹的关节运动曲线 | 位置控制

从曲线可以观察到：

- 位置连续性：**各关节位置曲线平滑连续，在正方形四个顶点处有明显的转折，这是由于路径方向改变所致。
- 速度平滑性：**速度曲线呈现周期性变化，在直线段保持相对稳定，在转角处有适当的过渡。
- 加速度控制：**加速度峰值控制在限制范围内，整体变化平稳，无剧烈跳变。

#### 4.1.2 圆形轨迹

下图展示了机械臂末端执行器沿圆形轨迹运动的结果。绿色圆圈清晰地显示了末端执行器的运动轨迹，直径约为 10cm：

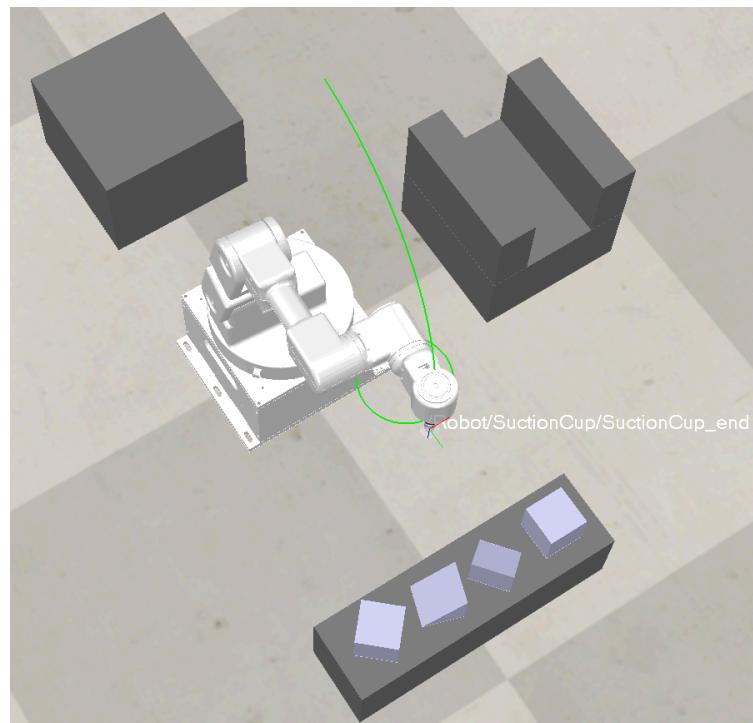


图 12 圆形轨迹运动结果 | 位置控制

下图为圆形轨迹运动时的关节运动曲线：

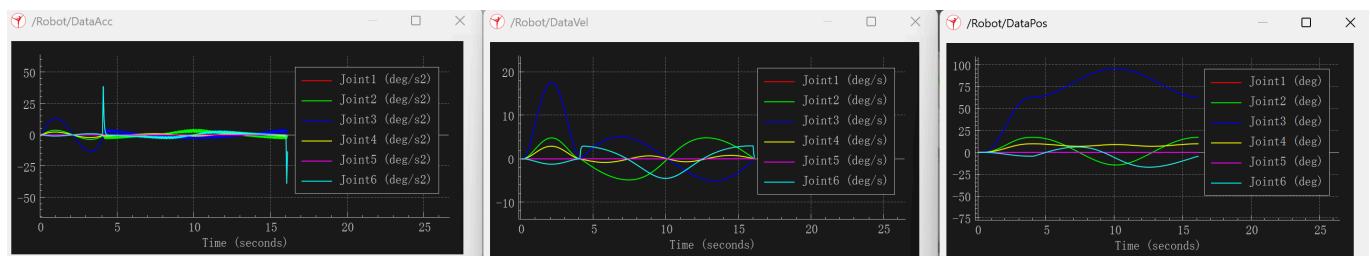


图 13 圆形轨迹的关节运动曲线 | 位置控制

圆形轨迹的运动特点：

- 周期性：**由于圆形轨迹的周期性特性，各关节的位置、速度曲线均呈现明显的周期性变化。
- 平滑性：**相比正方形轨迹，圆形轨迹的速度和加速度变化更加平滑，这是因为圆形路径没有尖锐的转角。
- 关节 3 波动：**关节 3（蓝色曲线）的速度变化幅度最大，达到约  $\pm 20$  deg/s，这是由于该关节在维持圆形轨迹中承担了主要的运动任务。

#### 4.1.3 圆锥轨迹

下图展示了机械臂末端执行器绕固定点作圆锥运动的结果。从图中可以看出，末端执行器的轴线与圆锥素线重合，末端点保持固定：

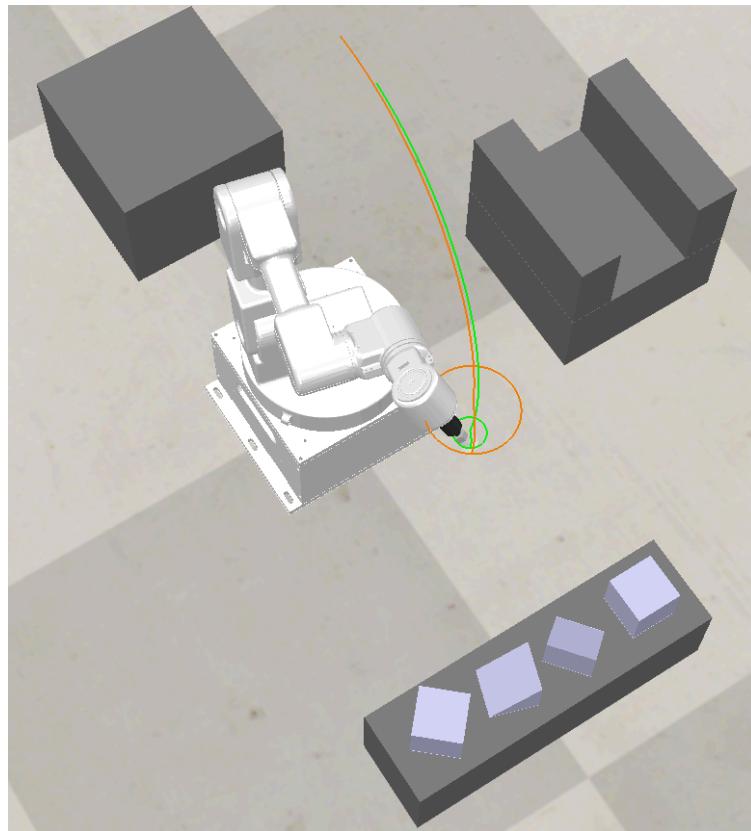


图 14 圆锥轨迹运动结果 | 位置控制

下图为圆锥轨迹运动时的关节运动曲线：

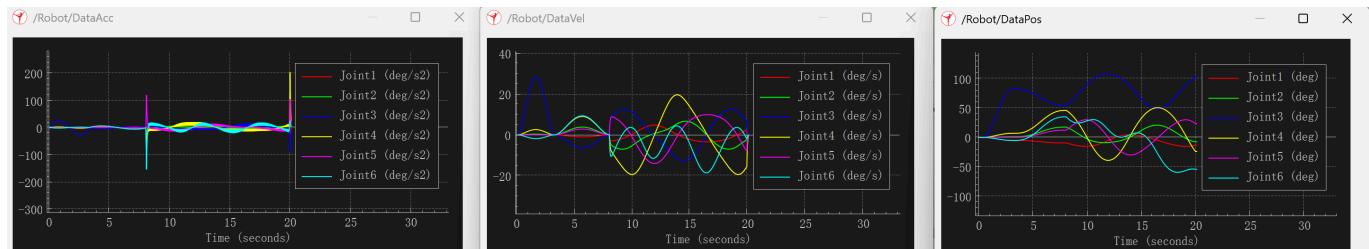


图 15 圆锥轨迹的关节运动曲线 | 位置控制

圆锥轨迹的特点：

- 姿态变化：**与前两个任务不同，圆锥运动主要体现在末端执行器的姿态变化上，而非位置变化。
- 速度分布：**各关节速度呈现周期性变化，其中关节 1 和关节 3 的速度变化最为显著。
- 加速度峰值：**在约 10 秒时刻，关节 5 出现了较大的加速度峰值（超过  $100 \text{ deg/s}^2$ ），但仍在安全限制 ( $500 \text{ deg/s}^2$ ) 范围内。

## 4.2 理论分析与计算

### 4.2.1 坐标系与运动学

- 在机械臂轨迹规划中，涉及两种主要的坐标系：
  - 关节空间：**由各关节角度  $\mathbf{q} = [\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]^T$  组成的空间。
  - 笛卡尔空间：**由末端执行器的位置和姿态  $\mathbf{p} = [x, y, z, \alpha, \beta, \gamma]^T$  组成的空间。
- 正运动学建立了从关节空间到笛卡尔空间的映射：

$$\mathbf{p} = f(\mathbf{q}) \quad (35)$$

3. 逆运动学则是其逆过程：

$$\mathbf{q} = f^{-1}(\mathbf{p}) \quad (36)$$

由于逆运动学通常存在多解，需要通过优化算法选择最合理的解。

#### 4.2.2 五次多项式轨迹规划

为了保证关节运动的平滑性，采用五次多项式进行轨迹插值。五次多项式的形式为：

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \quad (37)$$

对应的速度和加速度为：

$$\begin{cases} \dot{\theta}(t) = a_1 + 2a_2 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \\ \ddot{\theta}(t) = 2a_2 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \end{cases} \quad (38)$$

给定初始和终止条件：

$$\begin{cases} \theta(0) = \theta_0, \dot{\theta}(0) = v_0, \ddot{\theta}(0) = a_0 \\ \theta(T) = \theta_T, \dot{\theta}(T) = v_T, \ddot{\theta}(T) = a_T \end{cases} \quad (39)$$

可以求解出六个系数  $a_0, a_1, a_2, a_3, a_4, a_5$ ，从而得到平滑的轨迹。

#### 4.2.3 正方形轨迹规划

1. 正方形轨迹由四条边组成，每条边的规划方法相同。对于每条边：

- (1) 确定起点和终点的笛卡尔坐标
- (2) 使用逆运动学求解器计算对应的关节角
- (3) 在关节空间使用五次多项式插值
- (4) 保持末端姿态不变

2. 正方形的四个顶点坐标设定为（以中心为原点）：

$$\begin{cases} \mathbf{p}_1 = [x_c - L/2, y_c - L/2, z_c, \alpha, \beta, \gamma]^T \\ \mathbf{p}_2 = [x_c + L/2, y_c - L/2, z_c, \alpha, \beta, \gamma]^T \\ \mathbf{p}_3 = [x_c + L/2, y_c + L/2, z_c, \alpha, \beta, \gamma]^T \\ \mathbf{p}_4 = [x_c - L/2, y_c + L/2, z_c, \alpha, \beta, \gamma]^T \end{cases} \quad (40)$$

其中  $L = 0.1$  m 为边长。

#### 4.2.4 圆形轨迹规划

圆形轨迹通过参数方程描述。将圆周分为  $N$  个离散点，每个点的位置为：

$$\begin{cases} x(t) = x_c + R \cos(2\pi t/T) \\ y(t) = y_c + R \sin(2\pi t/T) \\ z(t) = z_c \end{cases} \quad (41)$$

其中  $R = 0.05$  m 为半径， $T$  为运动周期。对于每个时间段，使用五次多项式在关节空间插值。

#### 4.2.5 圆锥轨迹规划

1. 圆锥运动的特点是末端点位置固定，但末端执行器的姿态按圆锥面旋转。设圆锥顶点为  $\mathbf{p}_{\text{apex}}$ ，半锥角为  $\alpha_{\text{cone}} = 30^\circ$ 。
2. 末端执行器的轴线（approach vector）在圆锥面上运动，可以用球坐标描述：

$$\mathbf{a}(\varphi) = \begin{pmatrix} \sin(\alpha_{\text{cone}}) \cos(\varphi) \\ \sin(\alpha_{\text{cone}}) \sin(\varphi) \\ \cos(\alpha_{\text{cone}}) \end{pmatrix} \quad (42)$$

其中  $\varphi \in [0, 2\pi]$  为方位角。对于每个姿态，需要构建完整的旋转矩阵  $\mathbf{R} = [\mathbf{n}, \mathbf{o}, \mathbf{a}]$ ，然后转换为欧拉角。

3. 为了保证姿态的连续性，引入自旋角（spin angle）参数，通过优化算法选择使关节角变化最小的解。

## 4.3 代码实现思路

### 4.3.1 总体架构

代码分为以下几个主要部分：

1. 逆运动学求解器（myIKSolver 类）：根据末端位姿计算关节角，返回所有可能的解。
2. 轨迹规划函数（quintic\_trajectory\_coefficients 等）：计算多项式系数和轨迹点。
3. 姿态处理函数（build\_rotation\_matrix\_from\_a\_axis 等）：处理旋转矩阵和欧拉角转换。
4. 初始化函数（sysCall\_init）：为每个任务生成完整的轨迹参数。
5. 执行函数（sysCall\_actuation）：在每个仿真步实时计算并执行关节角。

### 4.3.2 正方形轨迹实现

1. 正方形轨迹的关键步骤：

(1) 定义正方形中心位置和边长。

```
1 square_center = [0.05, 0.4, 0.25]
2 square_size = 0.1
3 square_pose = [np.pi, 0, -np.pi/2]
```

Python

(2) 计算四个顶点的笛卡尔坐标。

(3) 使用逆运动学求解器计算每个顶点的关节角。

(4) 从多个解中选择与前一点距离最小的解，保证连续性。

(5) 对每条边使用五次多项式规划，设置合适的边界条件。

2. 关键代码片段：

```
1 # 计算四条边的五次多项式系数
2 for i in range(4):
3     start_angles = square_angles[i]
4     end_angles = square_angles[(i+1) % 4]
5     edge_coeffs = []
6     for j in range(6):
7         coeffs = quintic_trajectory_coefficients(
8             start_angles[j], 0, 0,
9             end_angles[j], 0, 0,
10            time_per_edge
11        )
12         edge_coeffs.append(coeffs)
13 self.square_edge_coeffs.append(edge_coeffs)
```

Python

### 4.3.3 圆形轨迹实现

圆形轨迹将圆周均匀分为多个点：

1. 设定圆心、半径和姿态。

```
1 circle_center = [0.05, 0.4, 0.25]
2 circle_radius = 0.05
3 circle_pose = [np.pi, 0, -np.pi/2]
4 circle_points = 60
```

Python

2. 使用参数方程计算圆周上的点。

```
1 for i in range(circle_points):
2     angle = 2 * np.pi * i / circle_points
3     x = circle_center[0] + circle_radius * np.cos(angle)
4     y = circle_center[1] + circle_radius * np.sin(angle)
5     z = circle_center[2]
6     pose = [x, y, z] + circle_pose
```

Python

3. 对每个相邻点对进行五次多项式插值。  
 4. 特别处理最后一点到第一点的连接，形成闭环。

#### 4.3.4 圆锥轨迹实现

圆锥轨迹的难点在于姿态的连续性控制：

1. 固定末端点位置。

```
1 cone_apex = [0.05, 0.4, 0.25]
2 cone_half_angle_deg = 30
```

Python

2. 计算圆锥面上的方向向量。

```
1 for i in range(num_points):
2     phi = 2 * np.pi * i / num_points
3     a_axis = np.array([
4         np.sin(half_angle_rad) * np.cos(phi),
5         np.sin(half_angle_rad) * np.sin(phi),
6         np.cos(half_angle_rad)
7     ])
```

Python

3. 使用 `find_best_spin_for_pose` 函数搜索最优自旋角：该函数在一定范围内搜索，找到使关节角变化最小的姿态。

```
1 best_angles, best_spin = find_best_spin_for_pose(
2     iks, position, a_axis, prev_angles, prev_spin,
3     joint_limits, num_samples=SPIN_SEARCH_SAMPLES
4 )
```

Python

#### 4.3.5 关键技术细节

1. 多解选择策略：

- (1) 对于每个笛卡尔位姿，逆运动学可能返回多个解（最多 8 个）
- (2) 选择策略：优先选择与前一时刻关节角距离最小的解
- (3) 同时考虑关节限位，过滤掉超出限制的解

```
1 best_sol = None
2 min_dist = float('inf')
3 for sol_idx in range(angles.shape[1]):
4     candidate = angles[:, sol_idx]
5     # 检查关节限位
```

Python

```

6     if not all(joint_limits[j,0] <= candidate[j] <= joint_limits[j,1]
7         for j in range(6)):
8     continue
9     # 计算距离
10    dist = np.sum((candidate - prev_angles)**2)
11    if dist < min_dist:
12        min_dist = dist
13    best_sol = candidate

```

## 2. 过渡轨迹：

- (1) 在开始主轨迹前，需要从初始位置（关节角全为 0）过渡到轨迹起点
- (2) 使用五次多项式规划过渡段，设置合适的过渡时间
- (3) 对于圆锥任务，使用两段过渡以避免关节角剧烈变化

## 3. 轨迹可视化：

- (1) 使用 CoppeliaSim 的 drawing object 功能绘制轨迹
- (2) 绿色轨迹显示末端执行器路径
- (3) 对于圆锥任务，额外用橙色显示关节 6 的轨迹

## 4.4 核心代码实现

### 4.4.1 逆运动学求解器（自行编写的）

```

1 class myIKSolver:
2     def solve(self, p):
3         d_x, d_y, d_z, alpha, beta, gamma = p
4         d_1, a_2, a_3, d_4, d_5, d_6 = 0.230, 0.185, 0.170, 0.023, 0.077, 0.0855
5         n_x, n_y, n_z = np.cos(gamma)* np.cos(beta), np.cos(alpha) * np.sin(gamma) +
5             np.sin(alpha) * np.sin(beta) * np.cos(gamma), np.sin(alpha) * np.sin(gamma) -
5                 np.cos(alpha) * np.sin(beta) * np.cos(gamma)
6         o_x, o_y, o_z = -np.sin(gamma) * np.cos(beta), np.cos(alpha) * np.cos(gamma) -
6             np.sin(alpha) * np.sin(beta) * np.sin(gamma), np.sin(alpha) * np.cos(gamma) +
6                 np.cos(alpha) * np.sin(beta) * np.sin(gamma)
7         a_x, a_y, a_z = np.sin(beta), - np.sin(alpha) * np.cos(beta), np.cos(alpha) *
7             np.cos(beta)
8
9         A = d_y - d_6 * a_y
10        B = d_x - d_6 * a_x
11        pm = np.array([1, -1])
12        theta_1 = np.arctan2(A, B) - np.arctan2(d_4, pm * np.sqrt(A**2 + B**2 - d_4**2))
13
14        theta_5 = np.arcsin(a_y * np.cos(theta_1) - a_x * np.sin(theta_1))
15
16        theta_5 = np.array([[x, np.pi - x if x > 0 else -np.pi - x] for x in
16            theta_5]).flatten()
17        theta_1 = np.array([[x, x] for x in theta_1]).flatten()
18
19        C = n_y * np.cos(theta_1) - n_x * np.sin(theta_1)
20        D = o_y * np.cos(theta_1) - o_x * np.sin(theta_1)

```

```

21     theta_6 = np.arctan2(C, D) - np.arctan2(np.cos(theta_5), 0)
22
23     E = -d_5 * (np.sin(theta_6) * (n_x * np.cos(theta_1) + n_y * np.sin(theta_1)) +
24         np.cos(theta_6) * (o_x * np.cos(theta_1) + o_y * np.sin(theta_1))) - d_6 * (a_x *
25         np.cos(theta_1) + a_y * np.sin(theta_1)) + d_x * np.cos(theta_1) + d_y *
26         np.sin(theta_1)
27     F = -d_1 - a_z * d_6 - d_5 * (o_z * np.cos(theta_6) + n_z * np.sin(theta_6)) + d_z
28     theta_3 = np.arccos((E**2 + F**2 - a_2**2 - a_3**2) / (2 * a_2 * a_3))
29
30     theta_3 = np.array([[x, -x] for x in theta_3]).flatten()
31     theta_1 = np.array([[x, x] for x in theta_1]).flatten()
32     theta_5 = np.array([[x, x] for x in theta_5]).flatten()
33     theta_6 = np.array([[x, x] for x in theta_6]).flatten()
34     E = np.array([[x, x] for x in E]).flatten()
35     F = np.array([[x, x] for x in F]).flatten()
36
37     G = a_2 + a_3 * np.cos(theta_3)
38     H = a_3 * np.sin(theta_3)
39     theta_2 = np.arctan2(G * E - H * F, G * F + H * E)
40
41     I = (n_x * np.cos(theta_1) + n_y * np.sin(theta_1)) * np.sin(theta_6) + (o_x *
42         np.cos(theta_1) + o_y * np.sin(theta_1)) * np.cos(theta_6)
43     J = n_z * np.sin(theta_6) + o_z * np.cos(theta_6)
44     theta_4 = np.arctan2(I, J) - theta_2 - theta_3
45
46     ans = np.array([theta_1, theta_2, theta_3, theta_4, theta_5, theta_6])
47     cols_with_nan = np.isnan(ans).any(axis=0)
48     ans = ans[:, ~cols_with_nan]
49
50     return ans

```

这个逆运动学求解器是基于机械臂的 DH 参数推导出来的解析解。整个求解过程按照 1→5→6→3→2→4 的顺序来算，这个顺序是经过推导确定的最优顺序。核心思路是利用末端位姿反推各关节角度，因为是 6 自由度机械臂，理论上最多可能有 8 组解（每个关节的正负解组合）。

代码里 `pm = np.array([1, -1])` 这一步会同时计算两种可能的解，然后通过数组操作不断扩展解空间。比如求 `theta_3` 时用了 `[[x, -x] for x in theta_3]` 这种方式来同时考虑肘部向上和向下两种姿态。最后用 `cols_with_nan` 来过滤掉那些数学上不成立的解（比如 `arccos` 超出定义域导致的 `NaN`）。

有个坑是最后那句 `ans[:, [0, 1, 2, 3]] = ans[:, [2, 3, 0, 1]]`，这是因为我推导时的关节顺序和实际仿真环境中的编号不一样，需要重新排列一下。返回的 `ans` 是个  $6 \times n$  的矩阵，每一列代表一组可行解。

#### 4.4.2 五次多项式轨迹规划

```

1 def quintic_trajectory_coefficients(p0, v0, a0, p1, v1, a1, total_time):
2     T = total_time

```

```

3
4     coeff_a0 = p0
5     coeff_a1 = v0
6     coeff_a2 = a0 / 2.0
7     coeff_a3 = (20*p1 - 20*p0 - (8*v1 + 12*v0)*T - (3*a0 - a1)*T**2) / (2*T**3)
8     coeff_a4 = (30*p0 - 30*p1 + (14*v1 + 16*v0)*T + (3*a0 - 2*a1)*T**2) / (2*T**4)
9     coeff_a5 = (12*p1 - 12*p0 - (6*v1 + 6*v0)*T - (a0 - a1)*T**2) / (2*T**5)
10
11    return [coeff_a0, coeff_a1, coeff_a2, coeff_a3, coeff_a4, coeff_a5]
12
13
14 def quintic_trajectory_eval(coeffs, t):
15     a0, a1, a2, a3, a4, a5 = coeffs
16     return a0 + a1*t + a2*t**2 + a3*t**3 + a4*t**4 + a5*t**5

```

五次多项式是轨迹规划里最常用的方法之一，好处是可以同时约束位置、速度和加速度。系数的计算公式看着挺吓人，其实就是把六个边界条件（起点的位置/速度/加速度，终点的位置/速度/加速度）代入多项式及其导数，解一个六元一次方程组得到的。

这里直接把解出来的公式硬编码了，比用矩阵求逆要快一些。计算时注意 coeff\_a0 到 coeff\_a5 对应的是多项式从低次到高次的系数。quintic\_trajectory\_eval 就是个简单的霍纳法则求值，给定时间 t 就能算出对应的关节角度。

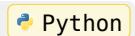
实际使用时，一般会把起点和终点的速度、加速度都设为 0，这样可以保证在路径切换的时候不会有突变。但如果想让运动更流畅，也可以在中间点保持一定的速度，让机械臂不用完全停下来再启动。

#### 4.4.3 圆锥轨迹的姿态处理

```

1  def build_rotation_matrix_from_a_axis(a_vec, spin_angle=0):
2      # build rotation matrix [n, o, a] where a is the approach vector
3      # spin_angle controls rotation around a axis
4      a = a_vec / np.linalg.norm(a_vec)
5
6      # choose reference perpendicular to a
7      if np.abs(a[2]) < 0.9:
8          ref = np.array([0, 0, 1])
9      else:
10         ref = np.array([1, 0, 0])
11
12     # base n and o vectors
13     n_base = np.cross(a, ref)
14     n_base = n_base / np.linalg.norm(n_base)
15     o_base = np.cross(a, n_base)
16
17     # apply spin rotation around a axis
18     cos_s = np.cos(spin_angle)
19     sin_s = np.sin(spin_angle)
20     n = cos_s * n_base + sin_s * o_base
21     o = -sin_s * n_base + cos_s * o_base
22

```



```

23     R = np.column_stack([n, o, a])
24     return R
25
26
27 def rotation_matrix_to_euler_zyx(R):
28     beta = np.arctan2(-R[2, 0], np.sqrt(R[0, 0]**2 + R[1, 0]**2))
29
30     if np.abs(np.cos(beta)) > 1e-6:
31         alpha = np.arctan2(R[2, 1] / np.cos(beta), R[2, 2] / np.cos(beta))
32         gamma = np.arctan2(R[1, 0] / np.cos(beta), R[0, 0] / np.cos(beta))
33     else:
34         alpha = 0
35         gamma = np.arctan2(-R[0, 1], R[1, 1])
36
37     return alpha, beta, gamma

```

圆锥轨迹最麻烦的就是姿态控制。机械臂末端需要绕着一个固定点旋转，同时保持轴线方向始终在圆锥面上。这就需要从 approach 向量（也就是末端轴线方向）构建完整的旋转矩阵。

`build_rotation_matrix_from_a_axis` 函数解决的就是这个问题。给定 a 向量后，需要再找两个垂直的向量 n 和 o 来构成完整的坐标系。这里用了个小技巧：先选一个参考向量（如果 a 不接近 z 轴就用 z 轴，否则用 x 轴），然后用叉乘来构造正交基。

关键是 `spin_angle` 这个参数。即使 a 向量确定了，绕 a 轴旋转还有无穷多种可能的姿态，这就是自旋自由度。通过调整 `spin_angle`，可以在保持末端方向不变的情况下旋转整个末端执行器。这个自由度后面会用来优化轨迹的连续性。

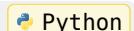
`rotation_matrix_to_euler_zyx` 就是标准的旋转矩阵转欧拉角公式了，需要注意万向锁的情况（当 `beta` 接近  $\pm 90^\circ$  时会出现奇异）。这里用了一个阈值判断来避免除零错误。

#### 4.4.4 最优自旋角搜索

```

1  def find_best_spin_for_pose(iks, position, a_axis, prev_angles, prev_spin,
                                joint_limits, num_samples=20):
2      # find best spin angle for smooth continuation from previous pose
3      # searches wider range and returns both angles and spin
4
5
6      # narrow search around previous spin for continuity
7      narrow_range = np.pi / 2
8      test_spins_narrow = np.linspace(prev_spin - narrow_range,
9                                      prev_spin + narrow_range, num_samples)
10
11     best_angles = None
12     best_spin = prev_spin
13     best_score = float('inf')
14
15     # narrow search first
16     for spin in test_spins_narrow:
17         R = build_rotation_matrix_from_a_axis(a_axis, spin)
18         alpha, beta, gamma = rotation_matrix_to_euler_zyx(R)

```



```
19     pose = np.concatenate([position, [alpha, beta, gamma]])
20
21     try:
22         angles = iks.solve(pose)
23         if angles.shape[1] == 0:
24             continue
25
26         for sol_idx in range(angles.shape[1]):
27             candidate = angles[:, sol_idx]
28
29             # check limits
30             if not all(joint_limits[j, 0] <= candidate[j] <= joint_limits[j, 1]
31                         for j in range(6)):
32                 continue
33
34             # score: heavily prioritize continuity
35             dist_score = np.sum((candidate - prev_angles)**2)
36             max_score = np.max(np.abs(candidate)) * 0.05
37             score = dist_score + max_score
38
39             if score < best_score:
40                 best_score = score
41                 best_angles = candidate
42                 best_spin = spin
43
44     except:
45         continue
46
47     # if no good solution or big jump, try wider search
48     if best_angles is None or np.max(np.abs(best_angles - prev_angles)) > 0.5:
49         test_spins_wide = np.linspace(prev_spin - np.pi,
50                                       prev_spin + np.pi, num_samples * 2)
51
52         for spin in test_spins_wide:
53             R = build_rotation_matrix_from_a_axis(a_axis, spin)
54             alpha, beta, gamma = rotation_matrix_to_euler_zyx(R)
55             pose = np.concatenate([position, [alpha, beta, gamma]])
56
57             try:
58                 angles = iks.solve(pose)
59                 if angles.shape[1] == 0:
60                     continue
61
62                 for sol_idx in range(angles.shape[1]):
63                     candidate = angles[:, sol_idx]
64
65                     if not all(joint_limits[j, 0] <= candidate[j] <= joint_limits[j, 1]
66                         for j in range(6)):
67                         continue
```

```
67
68         dist_score = np.sum((candidate - prev_angles)**2)
69         max_score = np.max(np.abs(candidate)) * 0.05
70         score = dist_score + max_score
71
72         if score < best_score:
73             best_score = score
74             best_angles = candidate
75             best_spin = spin
76     except:
77         continue
78
79     return best_angles, best_spin
```

这个函数是整个圆锥轨迹能够平滑运动的核心。问题在于，对于圆锥面上的每个点，虽然 approach 向量是固定的，但绕这个向量旋转（spin）还有一个自由度。如果随便选一个 spin 角度，虽然末端方向对了，但关节角可能会突然跳变一大截，导致运动不连续。

搜索策略是两阶段的。先在上一次 spin 角附近搜索（ $\pm 90^\circ$  范围），这是基于连续性假设——相邻两个点的最优 spin 角应该比较接近。如果这个窄范围搜索失败了，或者找到的解跳变太大（超过  $0.5$  弧度），就扩大到  $\pm 180^\circ$  全范围搜索。

评分函数设计也有讲究：`dist_score` 是关节角变化的平方和，这个权重最大，保证连续性优先；`max_score` 是关节角绝对值的惩罚，避免选择那种关节角特别极端的解。两个分数加起来，分数越低越好。

每个 spin 角度都会生成一个位姿，然后调用逆运动学求解器得到多组解，再逐个检查关节限位，最后选出综合得分最好的那组。整个过程是个暴力搜索，但因为只在一维空间搜索（spin 角），计算量还能接受。实测下来，24 个采样点基本够用，既能保证找到好的解，又不会太慢。

## § 5 总结

本实验通过速度控制和位置控制两种方法实现了 ZJU-I 型机械臂在笛卡尔空间的轨迹规划任务。实验结果表明，两种控制方式各有特点，均能成功完成正方形、圆形和圆锥三种轨迹的跟踪。

速度控制方法通过 Jacobian 矩阵建立关节速度与末端速度的直接映射关系，具有较好的实时性和直观性。该方法在圆形和正方形轨迹上表现良好，因为这两类任务的末端速度可以直接定义。然而，速度控制在圆锥任务中的实现相对简单：由于末端点位置固定，线速度为零，仅需给定恒定的角速度向量即可完成轨迹跟踪，无需复杂的姿态规划。

相比之下，位置控制在圆锥任务中面临更大挑战。虽然末端点位置固定，但末端执行器的姿态需要在圆锥面上连续变化，这涉及复杂的旋转矩阵构建和欧拉角转换。更为关键的是，对于圆锥面上的每个姿态点，绕末端轴线旋转存在一个自旋自由度，导致逆运动学存在无穷多解。若自旋角选择不当，相邻路径点的关节角可能产生剧烈跳变，严重影响运动连续性。为解决这一问题，本实验设计了两阶段自旋角优化搜索算法，在保证姿态正确的前提下，通过最小化关节角变化来选择最优解，这使得位置控制的圆锥轨迹规划在算法复杂度和计算量上都显著高于速度控制方法。

通过本次实验，我们深入理解了笛卡尔空间轨迹规划的原理，掌握了 Jacobian 矩阵的应用、五次多项式插值以及逆运动学多解选择等关键技术，为后续复杂机器人任务的开发奠定了坚实基础。