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

2025 年 11 月 23 日

§ 1 实验目的和要求

1.1 实验目的

1. 掌握在笛卡尔空间进行轨迹规划的方法
2. 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
3. 理解关节空间与笛卡尔空间之间的转换关系
4. 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

1. **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。
2. **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。
3. **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。
4. 提交计算过程报告、仿真工程文件以及仿真结果视频。

§ 2 结果与分析

2.1 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。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：

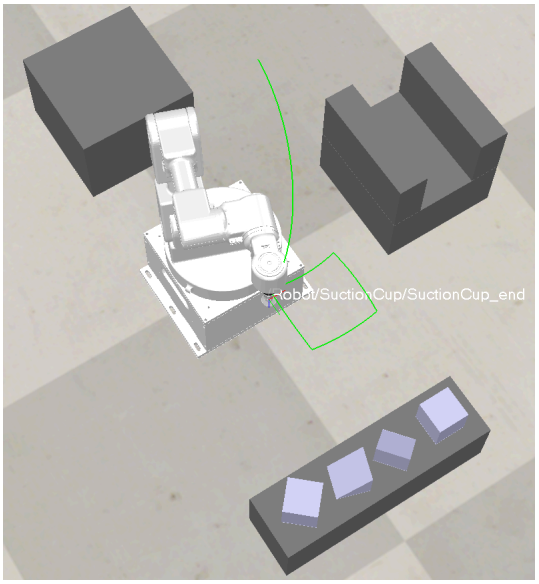

图 1 正方形轨迹运动结果

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：

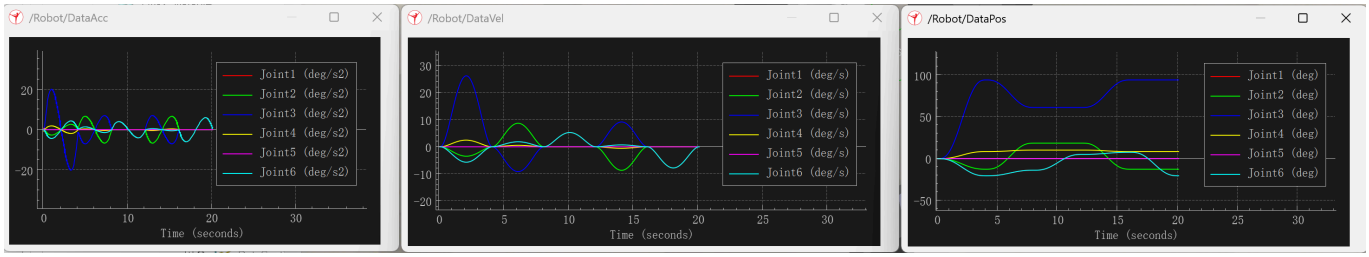

图 2　正方形轨迹的关节运动曲线

从曲线可以观察到：

1. 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。
2. 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。
3. 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。

2.2　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：

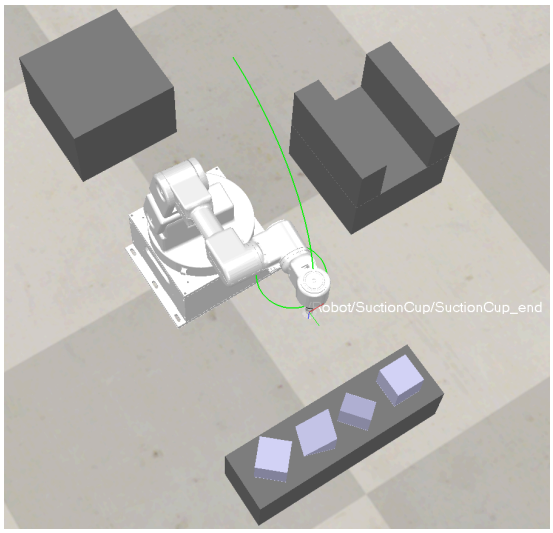

图 3　圆形轨迹运动结果

下图为圆形轨迹运动时的关节运动曲线：

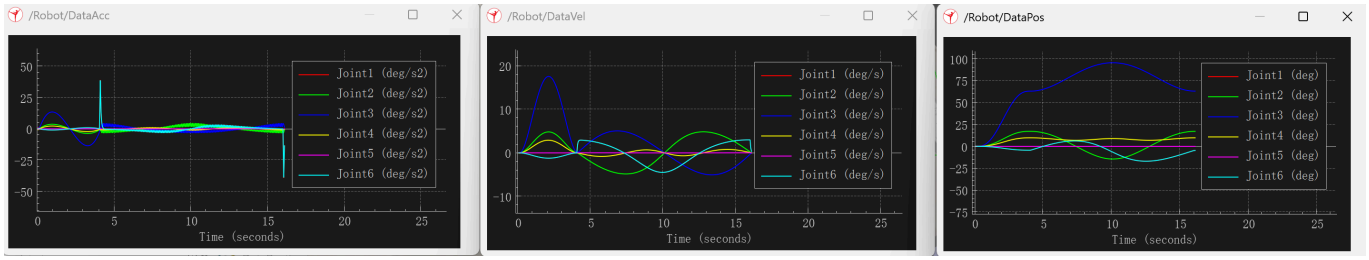

图 4　圆形轨迹的关节运动曲线

圆形轨迹的运动特点：

1. 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。
2. 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。

3. 关节 3 波动：关节 3（蓝色曲线）的速度变化幅度最大，达到约±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。

2.3 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：

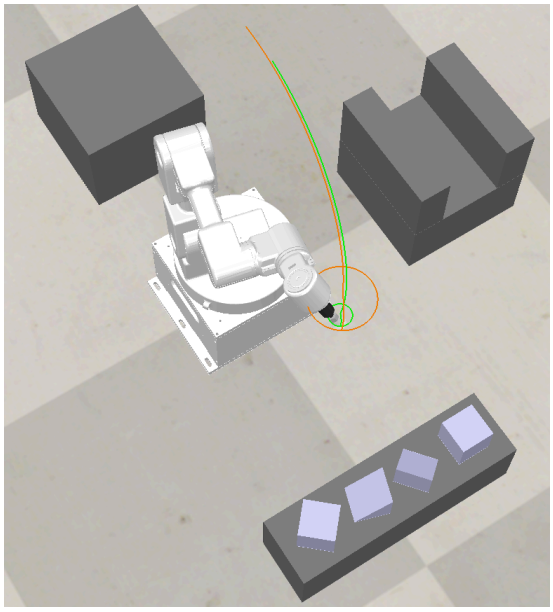

图 5 圆锥轨迹运动结果

下图为圆锥轨迹运动时的关节运动曲线：

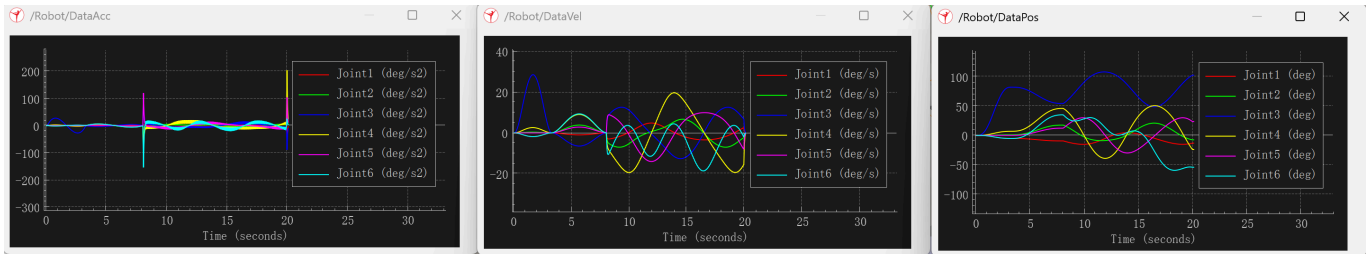

图 6 圆锥轨迹的关节运动曲线

圆锥轨迹的特点：

1. 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。
2. 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。
3. 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 deg/s²），但仍在安全限制（500 deg/s²）范围内。

§ 3 实验内容与原理

3.1 坐标系与运动学

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

$$\boldsymbol{p} = f(\boldsymbol{q}) \tag{1}$$

3. 逆运动学则是其逆过程：

$$\boldsymbol{q} = f^{-1}(\boldsymbol{p}) \tag{2}$$

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。

3.2 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：

$$\theta(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \tag{3}$$

对应的速度和加速度为：

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

给定初始和终止条件：

$$\begin{cases} \theta(0) = \theta_0, \dot{\theta}(0) = v_0, \ddot{\theta}(0) = a_0 \\ \theta(T) = \theta_T, \dot{\theta}(T) = v_T, \ddot{\theta}(T) = a_T \end{cases} \tag{5}$$

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。

3.3 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）：

$$\begin{cases} \boldsymbol{p}_1 = [x_c - L/2, y_c - L/2, z_c, \alpha, \beta, \gamma]^T \\ \boldsymbol{p}_2 = [x_c + L/2, y_c - L/2, z_c, \alpha, \beta, \gamma]^T \\ \boldsymbol{p}_3 = [x_c + L/2, y_c + L/2, z_c, \alpha, \beta, \gamma]^T \\ \boldsymbol{p}_4 = [x_c - L/2, y_c + L/2, z_c, \alpha, \beta, \gamma]^T \end{cases} \tag{6}$$

其中 $L = 0.1$ m 为边长。

3.4 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：

$$\begin{cases} x(t) = x_c + R\cos(2\pi t/T) \\ y(t) = y_c + R\sin(2\pi t/T) \\ z(t) = z_c \end{cases} \tag{7}$$

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。

3.5 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 $\boldsymbol{p}_{\text{apex}}$，半锥角为 $\alpha_{\text{cone}} = 30°$。
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：

$$\boldsymbol{a}(\varphi) = \begin{pmatrix} \sin(\alpha_{\text{cone}})\cos(\varphi) \\ \sin(\alpha_{\text{cone}})\sin(\varphi) \\ \cos(\alpha_{\text{cone}}) \end{pmatrix} \tag{8}$$

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 $\boldsymbol{R} = [\boldsymbol{n}, \boldsymbol{o}, \boldsymbol{a}]$，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 代码实现思路

4.1 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。

4.2 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。

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

   (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：

```python
# 计算四条边的五次多项式系数
for i in range(4):
    start_angles = square_angles[i]
    end_angles = square_angles[(i+1) % 4]
    edge_coeffs = []
    for j in range(6):
        coeffs = quintic_trajectory_coefficients(
            start_angles[j], 0, 0,
            end_angles[j], 0, 0,
            time_per_edge
        )
        edge_coeffs.append(coeffs)
    self.square_edge_coeffs.append(edge_coeffs)
```

4.3 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。

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

2. 使用参数方程计算圆周上的点。

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

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。

4.4 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。

```python
cone_apex = [0.05, 0.4, 0.25]
cone_half_angle_deg = 30
```

2. 计算圆锥面上的方向向量。

```python
for i in range(num_points):
    phi = 2 * np.pi * i / num_points
    a_axis = np.array([
        np.sin(half_angle_rad) * np.cos(phi),
        np.sin(half_angle_rad) * np.sin(phi),
        np.cos(half_angle_rad)
    ])
```

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。

```python
best_angles, best_spin = find_best_spin_for_pose(
    iks, position, a_axis, prev_angles, prev_spin,
    joint_limits, num_samples=SPIN_SEARCH_SAMPLES
)
```

4.5 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

```python
best_sol = None
min_dist = float('inf')
for sol_idx in range(angles.shape[1]):
    candidate = angles[:, sol_idx]
```

```
        # 检查关节限位
        if not all(joint_limits[j,0] <= candidate[j] <= joint_limits[j,1]
                   for j in range(6)):
            continue
        # 计算距离
        dist = np.sum((candidate - prev_angles)**2)
        if dist < min_dist:
            min_dist = dist
            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

§ 5 核心代码实现

5.1 逆运动学求解器（自行编写的）

Python

```python
class myIKSolver:
    def solve(self, p):
        d_x, d_y, d_z, alpha, beta, gamma = p
        d_1, a_2, a_3, d_4, d_5, d_6 = 0.230, 0.185, 0.170, 0.023, 0.077, 0.0855
        n_x, n_y, n_z = np.cos(gamma)* np.cos(beta), np.cos(alpha) * np.sin(gamma) + np.sin(alpha) * np.sin(beta) * np.cos(gamma), np.sin(alpha) * np.sin(gamma) - np.cos(alpha) * np.sin(beta) * np.cos(gamma)
        o_x, o_y, o_z = -np.sin(gamma) * np.cos(beta), np.cos(alpha) * np.cos(gamma) - np.sin(alpha) * np.sin(beta) * np.sin(gamma), np.sin(alpha) * np.cos(gamma) + np.cos(alpha) * np.sin(beta) * np.sin(gamma)
        a_x, a_y, a_z = np.sin(beta), - np.sin(alpha) * np.cos(beta), np.cos(alpha) * np.cos(beta)

        A = d_y - d_6 * a_y
        B = d_x - d_6 * a_x
        pm = np.array([1, -1])
        theta_1 = np.arctan2(A, B) - np.arctan2(d_4,  pm * np.sqrt(A**2 + B**2 - d_4**2))

        theta_5 = np.arcsin(a_y * np.cos(theta_1) - a_x * np.sin(theta_1))

        theta_5 = np.array([[x, np.pi - x if x > 0 else -np.pi - x] for x in theta_5]).flatten()
        theta_1 = np.array([[x, x] for x in theta_1]).flatten()

```

```
        C = n_y * np.cos(theta_1) - n_x * np.sin(theta_1)
        D = o_y * np.cos(theta_1) - o_x * np.sin(theta_1)
        theta_6 = np.arctan2(C, D) - np.arctan2(np.cos(theta_5), 0)

        E = -d_5 * (np.sin(theta_6) * (n_x * np.cos(theta_1) + n_y * np.sin(theta_1)) +
        np.cos(theta_6) * (o_x * np.cos(theta_1) + o_y * np.sin(theta_1))) - d_6 * (a_x *
        np.cos(theta_1) + a_y * np.sin(theta_1)) + d_x * np.cos(theta_1) + d_y *
        np.sin(theta_1)
        F = -d_1 - a_z * d_6 - d_5 * (o_z * np.cos(theta_6) + n_z * np.sin(theta_6)) + d_z
        theta_3 = np.arccos((E**2 + F**2 - a_2**2 - a_3**2) / (2 * a_2 * a_3))

        theta_3 = np.array([[x, -x] for x in theta_3]).flatten()
        theta_1 = np.array([[x, x] for x in theta_1]).flatten()
        theta_5 = np.array([[x, x] for x in theta_5]).flatten()
        theta_6 = np.array([[x, x] for x in theta_6]).flatten()
        E = np.array([[x, x] for x in E]).flatten()
        F = np.array([[x, x] for x in F]).flatten()

        G = a_2 + a_3 * np.cos(theta_3)
        H = a_3 * np.sin(theta_3)
        theta_2 = np.arctan2(G * E - H * F, G * F + H * E)

        I = (n_x * np.cos(theta_1) + n_y * np.sin(theta_1)) * np.sin(theta_6) + (o_x *
        np.cos(theta_1) + o_y * np.sin(theta_1)) * np.cos(theta_6)
        J = n_z * np.sin(theta_6) + o_z * np.cos(theta_6)
        theta_4 = np.arctan2(I, J) - theta_2 - theta_3

        ans = np.array([theta_1, theta_2, theta_3, theta_4, theta_5, theta_6])
        cols_with_nan = np.isnan(ans).any(axis=0)
        ans = ans[:, ~cols_with_nan]

        ans = (ans + np.pi) % (2 * np.pi) - np.pi

        ans[:, [0, 1, 2, 3]] = ans[:, [2, 3, 0, 1]]

        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×n 的矩阵，每一列代表一组可行解。

5.2 五次多项式轨迹规划

```python
def quintic_trajectory_coefficients(p0, v0, a0, p1, v1, a1, total_time):
    T = total_time

    coeff_a0 = p0
    coeff_a1 = v0
    coeff_a2 = a0 / 2.0
    coeff_a3 = (20*p1 - 20*p0 - (8*v1 + 12*v0)*T - (3*a0 - a1)*T**2) / (2*T**3)
    coeff_a4 = (30*p0 - 30*p1 + (14*v1 + 16*v0)*T + (3*a0 - 2*a1)*T**2) / (2*T**4)
    coeff_a5 = (12*p1 - 12*p0 - (6*v1 + 6*v0)*T - (a0 - a1)*T**2) / (2*T**5)

    return [coeff_a0, coeff_a1, coeff_a2, coeff_a3, coeff_a4, coeff_a5]


def quintic_trajectory_eval(coeffs, t):
    a0, a1, a2, a3, a4, a5 = coeffs
    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。

5.3 圆锥轨迹的姿态处理

```python
def build_rotation_matrix_from_a_axis(a_vec, spin_angle=0):
    # build rotation matrix [n, o, a] where a is the approach vector
    # spin_angle controls rotation around a axis
    a = a_vec / np.linalg.norm(a_vec)

    # choose reference perpendicular to a
    if np.abs(a[2]) < 0.9:
        ref = np.array([0, 0, 1])
    else:
        ref = np.array([1, 0, 0])

    # base n and o vectors
    n_base = np.cross(a, ref)
    n_base = n_base / np.linalg.norm(n_base)
    o_base = np.cross(a, n_base)

    # apply spin rotation around a axis
    cos_s = np.cos(spin_angle)
    sin_s = np.sin(spin_angle)
```

```
    n = cos_s * n_base + sin_s * o_base
    o = -sin_s * n_base + cos_s * o_base

    R = np.column_stack([n, o, a])
    return R


def rotation_matrix_to_euler_zyx(R):
    beta = np.arctan2(-R[2, 0], np.sqrt(R[0, 0]**2 + R[1, 0]**2))

    if np.abs(np.cos(beta)) > 1e-6:
        alpha = np.arctan2(R[2, 1] / np.cos(beta), R[2, 2] / np.cos(beta))
        gamma = np.arctan2(R[1, 0] / np.cos(beta), R[0, 0] / np.cos(beta))
    else:
        alpha = 0
        gamma = np.arctan2(-R[0, 1], R[1, 1])

    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 $\pm 90°$ 时会出现奇异）。这里用了一个阈值判断来避免除零错误。

5.4 最优自旋角搜索

```python
def find_best_spin_for_pose(iks, position, a_axis, prev_angles, prev_spin,
                            joint_limits, num_samples=20):
    # find best spin angle for smooth continuation from previous pose
    # searches wider range and returns both angles and spin

    # narrow search around previous spin for continuity
    narrow_range = np.pi / 2
    test_spins_narrow = np.linspace(prev_spin - narrow_range,
                                    prev_spin + narrow_range, num_samples)

    best_angles = None
    best_spin = prev_spin
    best_score = float('inf')

    # narrow search first
```

```
    for spin in test_spins_narrow:
        R = build_rotation_matrix_from_a_axis(a_axis, spin)
        alpha, beta, gamma = rotation_matrix_to_euler_zyx(R)
        pose = np.concatenate([position, [alpha, beta, gamma]])

        try:
            angles = iks.solve(pose)
            if angles.shape[1] == 0:
                continue

            for sol_idx in range(angles.shape[1]):
                candidate = angles[:, sol_idx]

                # check limits
                if not all(joint_limits[j, 0] <= candidate[j] <= joint_limits[j, 1]
                           for j in range(6)):
                    continue

                # score: heavily prioritize continuity
                dist_score = np.sum((candidate - prev_angles)**2)
                max_score = np.max(np.abs(candidate)) * 0.05
                score = dist_score + max_score

                if score < best_score:
                    best_score = score
                    best_angles = candidate
                    best_spin = spin
        except:
            continue

    # if no good solution or big jump, try wider search
    if best_angles is None or np.max(np.abs(best_angles - prev_angles)) > 0.5:
        test_spins_wide = np.linspace(prev_spin - np.pi,
                                      prev_spin + np.pi, num_samples * 2)

        for spin in test_spins_wide:
            R = build_rotation_matrix_from_a_axis(a_axis, spin)
            alpha, beta, gamma = rotation_matrix_to_euler_zyx(R)
            pose = np.concatenate([position, [alpha, beta, gamma]])

            try:
                angles = iks.solve(pose)
                if angles.shape[1] == 0:
                    continue

                for sol_idx in range(angles.shape[1]):
                    candidate = angles[:, sol_idx]

```

```
                    if not all(joint_limits[j, 0] <= candidate[j] <= joint_limits[j, 1]
                               for j in range(6)):
                        continue

                    dist_score = np.sum((candidate - prev_angles)**2)
                    max_score = np.max(np.abs(candidate)) * 0.05
                    score = dist_score + max_score

                    if score < best_score:
                        best_score = score
                        best_angles = candidate
                        best_spin = spin
            except:
                continue

    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（±90°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 ±180° 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。