机械臂关节角路径和底座移动路径优化模型

摘要

在工业化和自动化的背景下，机械臂的使用已经成为生产中不可或缺的一环。机械臂关节角是影响机械臂精度和能效的核心，因此，对机械臂关节角路径在各种场景和任务下的优化设计具有重要的现实意义。

对于问题一, 首先将参数表提供的关节数据转化为**齐次变换矩阵**形式，再将矩阵累乘，就可以计算出每个连杆的末端位置，得到零位状态的六自由度机械臂简图如图 1所示。随后，根据欧几里得距离列出末端误差的计算式，并将 θ_i 关节变化范围限制作为约束条件。根据**标准 D-H 参数法**中的知识可知，累乘后得到的矩阵中的 3 维向量，即为末端的空间坐标。由此可以建立以最小化末端误差为目标的关节角路径优化模型，通过**自适应遗传算法**进行逆向求解，得到**最优关节角路径为**
$(1.34°, -22.65°, 80°, -3.46°, -105.58°, -8.4°)$。

对于问题二, 在问题一模型的基础上，先根据末端质量和高度差计算重力势能损耗，再根据转动惯量和平均角速度计算关节转动能耗，将两者相加得到总能量损耗的计算公式。把总能量损耗的计算公式和末端误差不超过 $\pm 200mm$ 作为两个新增的**约束条件**加入问题一建立的优化模型中，即建立了问题二在满足和能量损耗最小的情况下的关节角路径优化模型。求解算法与问题一类似，得到**最优关节角路径**
$(0.33°, 4.4°, -92°, 21.03°, -0.84°, 66.59°)$，可视化结果如图 6所示。

对于问题三, 首先对机械臂底座在给定的栅格图障碍物中网格状移动的最优路径，使用**曼哈顿距离**计算单步路径，随后采用 **A* 算法**逐步对逐步对路径优劣进行比较评估，建立底座移动路径优化模型，求出最佳路径可视化结果为表 5。因底座移动时的能耗不计，直接采用问题二建立的关节角路径优化模型进行计算，得到最优关节角路径与问题二一致。

对于问题四, 在问题三模型的基础上，首先读入附件中的货物和障碍物位置，并对目标货物之间的距离进行编号。将问题转化为**经典 TSP 旅行商问题**，采用**模拟退火算法**进行路径寻优，即通过不断地随机交换路径上的两个城市进行长度比较，寻找最短路径，并使用 **Metropolis 准则**来确定是否更新路径，建立了问题四的底座移动路径优化模型，得到**最优底座移动路径顺序为：Start -> target1 -> target2 -> target3 -> target5 -> target4 -> Start**，栅格图结果见图 7。随后，采用问题二以最小化末端误差和能耗为目标的关节角路径优化模型，得到最优关节角路径见图 8，同时**总末端误差为 12.02mm**, **总能耗为 24.11J**。

关键字： 正运动学标准 D-H 参数法　自适应遗传算法　模拟退火算法　A* 算法　经典 TSP 模型

一、 问题重述

1.1 问题背景

在机械化和自动化进程飞速发展的今天，由机械臂组成的自动化装置已经遍布国民生产的方方面面，有着不可替代的作用。机械臂关节角路径的优化设计是机械臂研究的重点，因其涉及到多个相互掣肘的因素，优化时需要制定合理的方案，才能达到利益最大化。在实际操作中，我们要以最小化输出误差为首要目标，同时尽可能缩小因关节转动、重力各种能耗。

本文在不同的任务环境要求下建立数学模型，尝试给出最小化末端误差和能耗的方案，对相关工业实践具有一定的指导价值。

1.2 问题要求

问题 1 要求首先根据机械臂初始 D-H 参数表，1. 绘制出零位状态的六自由度机械臂简图。2. 假设机械臂末端需要移动到目标点 (1500mm,1200mm,200mm) 进行抓取，建立数学模型，计算出在末端误差最小时机械臂的关节角路径。

问题 2 要求在问题一的基础上，1. 根据题目提供的关节转动能耗参数计算总关节转动能耗。2. 根据机械臂质量和末端质量之和计算消耗的重力势能。3. 在末端误差允许范围是 $\pm 200mm$ 的情况下，建立数学模型，计算出在末端误差和能耗最小时机械臂的关节角路径。

问题 3 要求在问题二的基础上，假设机械臂要靠其底座以网格状移动的方式先通过一片障碍物和网格如“附件.xlsx”sheet1 所示的栅格区域，再进行抓取工作，同时返回出发点，1. 只考虑抓取时消耗的动能，建立数学模型，计算出在末端误差和能耗最小时底座的最优移动路径。2. 计算出在末端误差和能耗最小时机械臂的最优关节角路径。3. 以栅格图的样式将底座移动路径进行可视化。

问题 4 要求将问题三中的只有路径结束点有需要抓取的货物的情形变成栅格中存在 5 个目标货物需要在一次任务中全部抓取，货物和障碍物位置见“附件.xlsx”sheet2 并同样要求返回出发点，1. 只考虑抓取时消耗的动能，建立数学模型，计算出在末端误差和能耗最小时底座的最优移动路径。2. 计算出在末端误差和能耗最小时机械臂的最优关节角路径。3. 以栅格图的样式将底座移动路径进行可视化。4. 给出总末端误差和总能耗的值。

二、 问题分析

2.1 问题一分析

对于问题一，首先要求绘制零位状态的六自由度机械臂简图，只需要使用标准 D-H 参数法，将参数表中提供的参数转化为齐次变换矩阵形式，随后将矩阵逐个相乘（也就是乘法迭代），就可以计算出每个连杆的末端位置，再绘制连杆和关节，就可以得到零位状态的六自由度机械臂简图。

随后，题目要求将机械臂末端移到目标点 (1500mm,1200mm,200mm) 附近进行抓取，并建模求出末端误差最小时的关节角路径。对此，我们只需要列出约束条件，即根据欧几里得距离列出末端误差的计算式，和 θ_i 关节变化范围限制。随后我们根据标准 D-H 参数法中的知识，将参数表中的数据转化为齐次变换矩阵后将六个齐次矩阵相乘，因为其左上角的的 3×3 矩阵为末端旋转矩阵，其中的 3 维向量即末端的空间坐标位置，所以我们以此建模，最后通过自适应遗传算法求解这个数学模型即可。

2.2 问题二分析

对于问题二，在问题一的要求上加上了能量损耗，我们只需要根据末端质量和高度差计算重力势能损耗，根据转动惯量和平均角速度计算关节转动能耗，将两者相加即得到总能量损耗。

要求在满足末端距离不超过 $\pm 200mm$ 和能量损耗最小的情况下求解最优关节角路径，只需要把新增的能量损耗作为约束条件加入问题一建立的优化模型中即可。求解算法和问题一一致。

2.3 问题三分析

对于问题三，在问题二的基础上增加了机械臂底座在给定的栅格图障碍物中网格状移动的过程，再进行抓取。因为底座移动时的能耗不计，因此要求解最优底座移动路径，我们只需用曼哈顿距离计算单步路径，采用 A* 算法逐步对路劲优劣进行比较即可建立数学模型，求出最佳路径并可视化。而要求末端距离不超过 $\pm 200mm$ 和能量损耗最小的情况下求解最优关节角路径，其模型与问题二一致。

2.4 问题四分析

对于问题四，将问题三的条件变成了栅格图中的多个目标货物。很容易发现要求解最优底座移动路径，实际上就是一个经典的 TSP 旅行商问题，我们只要采用模拟退火算法进行寻优，通过不断随机交换路径上的两个城市来寻找最短路径，并使用 Metropolis

准则来确定是否更新路径，以此建模求解即可。而要求末端距离不超过 $\pm 200mm$ 和能量损耗最小的情况下求解最优关节角路径，其模型与问题二一致。

三、 模型假设

为简化问题，本文做出以下假设：

- 假设 1 假设机械臂本身的质量远远小于末端载重，即假设末端载重自身的重量是 5kg，且机械臂本身硬度够大。
- 假设 2 假设在一次绕过障碍物抓取多个货物的任务中，机械臂底座有足够大的空间和载重能力存放抓取到的货物，且在工作过程中放置货物到自己托盘的能耗不计。

四、 符号说明

符号	说明	单位
m	质量	kg
V	体积	m^3
D	欧几里得距离	mm
E	末端位置偏差	mm
a_{i-1}	连杆长度	mm
d_i	关节偏距	mm
L_1	曼哈顿距离	mm
α_{i-1}	连杆扭角	°
θ_i	关节角	°
W	能量	J
ΔE_p	重力势能损耗	J
I	转动惯量	$kg{\cdot}m^2$
ω	平均角速度	rad/s
Tem	系统温度	$°C$
Rot	旋转矩阵	/
${}^0_nR$	末端在基坐标系中的旋转矩阵	/
$Trans$	平移矩阵	/
${}^0_nT$	转换齐次矩阵	/
Ag	基坐标系	/
An	末端坐标系	/
p	接受新状态的概率	/
k	玻尔兹曼常数	/
ΔE	目标函数值差	/
${}^0_np$	末端在基坐标系中的空间向量	/
$f(n)$	从初始状态经由状态 n 到目标状态的代价估计	/
$g(n)$	从初始状态到状态 n 的实际代价	/
$h(n)$	从状态 n 到目标状态的最佳路径估计代价	/

五、 问题一的模型的建立和求解

5.1 问题一模型的建立

5.1.1 末端误差计算

首先,在基坐标系中,我们令机械臂的末端位置为 (x, y, z),目标位置记为 (x_0, y_0, z_0)。易知，末端位置与目标位置之间的欧几里得距离 D 为:

$$D = \sqrt{(x_0 - x)^2 + (y_0 - y)^2 + (z_0 - z)^2} \tag{1}$$

由题意，末端误差描述的是一次任务中机械臂末端位置与目标位置之间的位置偏差，即我们可以简单视为上述 (1) 中计算得到的欧式距离等价于题目要求的位置偏差。

我们将位置偏差记作 E，并代入目标位置 (1500,1200,200)，得到

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{2}$$

此时问题一的要求可以转化为，建立数学模型求解机械臂最优关节角路径，使得对应的 E 的最小。

5.1.2 齐次矩阵形式变换

由 D-H 参数法原理可知，只要每两个相邻连杆之间的四个参数 a_{i-1},α_{i-1},d_i,θ_i 是确定的，就可以直接写出其变换齐次矩阵，完成正向解算。

因此，为便于正向求解，首先我们需要将题目提供的机械臂初始 D-H 参数表转化为齐次矩阵形式表述。首先，对参数表均有:

$$^{n}_{n+1}T = \text{Rot}\,(X, \theta_{n+1})\,\text{Trans}\,(d_{n+1}, 0, 0)\,\text{Rot}\,(Z, \alpha_n)\,\text{Trans}\,(0, 0, a_n) \tag{3}$$

其中 Rot 为旋转矩阵，$Trans$ 为平移矩阵。

$$= \begin{bmatrix} \cos(\theta_{n+1}) & -\sin(\theta_{n+1})\cos(\alpha_n) & \sin(\theta_{n+1})\sin(\alpha_n) & \cos(\theta_{n+1})\,a_n \\ \sin(\theta_{n+1}) & \cos(\theta_{n+1})\cos(\alpha_n) & -\cos(\theta_{n+1})\sin(\alpha_n) & \sin(\theta_{n+1})\,a_n \\ 0 & \sin(\alpha_n) & \cos(\alpha_n) & a_n \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

对一个 n 自由度机器人，最终可以得到其从基坐标系 Ag 到末端坐标系 An 的转换齐次矩阵:

$$^{0}_{n}T = {}^{0}_{1}T\,{}^{1}_{2}T \cdots {}^{n}_{n-1}T \tag{5}$$

得到 $^{0}_{n}T$ 后，其矩阵 $^{0}_{n}T = \begin{bmatrix} & ^{0}_{n}R & & ^{0}_{n}p \\ 0 & 0 & 0 & 1 \end{bmatrix}$ 的左上角 3×3 矩阵 $^{0}_{n}R$ 即为末端在基坐标系中的旋转矩阵，**向量 $^{0}_{n}p$ 即为末端在基坐标系中的空间坐标位置**。

5.1.3 六自由度机械臂齐次矩阵形式

由 5.1.2 的推理可知，对于题目给定的 6 自由度机器人，我们可以通过求解向量 ${}^{0}_{6}p$ 来求出机械臂的末端空间坐标位置。

为了求出向量 ${}^{0}_{6}p$，我们首先需要依次求解 ${}^{0}_{1}T, {}^{1}_{2}T ... {}^{5}_{6}T$，随后求出 ${}^{0}_{6}T$，再由矩阵 ${}^{0}_{n}R$ 即可得到。

于是将题目给定的参数表中的数值代入，得到

$$
{}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos(\alpha_0) & \sin(\theta_1)\sin(\alpha_0) & \cos(\theta_1)\,a_0 \\ \sin(\theta_1) & \cos(\theta_1)\cos(\alpha_0) & -\cos(\theta_1)\sin(\alpha_0) & \sin(\theta_1)\,a_0 \\ 0 & \sin(\alpha_0) & \cos(\alpha_0) & a_0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}
$$

$$
= \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos 0^\circ & \sin(\theta_1)\sin 0^\circ & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos 0^\circ & -\cos(\theta_1)\sin 0^\circ & 0 \\ 0 & \sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7}
$$

此时 $\theta_1 \in (-160^\circ, 160^\circ)$。

同理，我们可以按同样的方法代入参数，得到 ${}^{1}_{2}T, {}^{1}_{2}T ... {}^{6}_{5}T$ 的齐次矩阵形式。

因此，对题设给出的 6 自由度机器人，可以得到其从基坐标系 Ag 到末端坐标系 An 的转换齐次矩阵:

$$
{}^{0}_{6}T = {}^{0}_{1}T\,{}^{1}_{2}T ... {}^{6}_{5}T \tag{8}
$$

5.1.4 关节角路径的优化模型

问题一实际上是一个约束条件下的优化问题，我们需要明确约束条件并建立数学优化模型。

根据 5.1.1 和 5.1.3 的分析，可以得到**约束条件**:

1. 末端误差 $E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2}$ 要尽可能取最小值

2.θ_i 的角度必须在题给的范围内变化

$$
\begin{cases} \theta_1 \in (-160^\circ, 160^\circ) \\ \theta_2 \in (-150^\circ, 15^\circ) \\ \theta_3 \in (-200^\circ, 80^\circ) \\ \theta_4 \in (-180^\circ, 180^\circ) \\ \theta_5 \in (-120^\circ, 120^\circ) \\ \theta_6 \in (-180^\circ, 180^\circ) \end{cases} \tag{9}
$$

因此，在明确约束条件的情况下，我们得以建立**求解关节角最优路径基于标准 D-H 参数法的数学模型**：

$$
\begin{cases}
{}_1^0T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos 0^\circ & \sin(\theta_1)\sin 0^\circ & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos 0^\circ & -\cos(\theta_1)\sin 0^\circ & 0 \\ 0 & \sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\cdots\cdots \\
{}_6^0T = {}_1^0T{}_2^1T \ldots {}_6^5T \\
\begin{cases}
\theta_1 \in (-160^\circ, 160^\circ) \\
\theta_2 \in (-150^\circ, 15^\circ) \\
\theta_3 \in (-200^\circ, 80^\circ) \\
\theta_4 \in (-180^\circ, 180^\circ) \\
\theta_5 \in (-120^\circ, 120^\circ) \\
\theta_6 \in (-180^\circ, 180^\circ)
\end{cases} \\
E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \\
E_{min}
\end{cases}
\tag{10}
$$

同理，我们也建立了**求解关节角最优路径基于改进 D-H 参数法的数学模型**：

$$
\begin{cases}
{}_1^0T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1)\cos 0^\circ & \cos(\theta_1)\cos 0^\circ & -\sin 0^\circ & 0 \\ \sin(\theta_1)\sin 0^\circ & \cos(\theta_1)\sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\cdots\cdots \\
{}_6^0T = {}_1^0T{}_2^1T \ldots {}_6^5T \\
\begin{cases}
\theta_1 \in (-160^\circ, 160^\circ) \\
\theta_2 \in (-150^\circ, 15^\circ) \\
\theta_3 \in (-200^\circ, 80^\circ) \\
\theta_4 \in (-180^\circ, 180^\circ) \\
\theta_5 \in (-120^\circ, 120^\circ) \\
\theta_6 \in (-180^\circ, 180^\circ)
\end{cases} \\
E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \\
E_{min}
\end{cases}
\tag{11}
$$

5.2 问题一模型的求解

Step1：使用标准 D-H 参数法绘制零位状态六自由度机械臂简图

(1). 定义题给连杆参数 params 列表，包含了机械臂每个连杆的参数，包括连杆长度 a、连

杆扭角α、连杆偏移d、连杆关节角θ,以及关节角的上下限。定义目标位置$targetposition$,机械臂末端执行器需要尽可能接近的位置。

(2). 使用标准 D-H 参数法，根据连杆参数计算从当前连杆坐标系到下一个连杆坐标系的变换矩阵。

(3). 使用 matplotlib 库绘制机械臂的初始状态，通过迭代乘法变换矩阵计算每个连杆的末端位置，并绘制连杆和关节，得到零位状态六自由度机械臂简图，如图 1所示。

Step2：自适应遗传算法

(1). 定义目标函数: 将计算机械臂末端执行器的位置与目标位置之间的欧几里得距离作为优化问题的目标函数。

(2). 初始化：生成初始种群，每个个体代表一组关节角度，代表机器人臂的一种可能配置，每个关节的角度在其允许的范围内随机生成。

(3). 遗传算法主循环：对每一代进行迭代，直到达到指定的代数。

a. 评估适应度：计算每个个体的适应度，即其目标函数值的倒数。目标函数计算的是末端执行器的实际位置与目标位置之间的欧几里得距离。

b. 排序种群：根据适应度对种群进行排序，并保留适应度最高的个体。

c. 更新最佳解：如果当前代的最佳适应度高于之前的最佳适应度，则更新最佳解。

d. 精英保留：保留适应度最高的几个个体作为精英个体，不参与后续的选择、交叉和变异操作。

e. 选择操作：使用轮盘赌选择方法从剩余个体中选择一定数量的个体，用于后续的交叉操作。

f. 交叉操作：通过多点交叉生成新的个体，即后代。交叉操作以一定的概率发生。

g. 变异操作：以一定的概率对后代的每个关节角度进行小范围的随机调整，以增加种群的多样性。

h. 局部搜索：对每个后代进行局部搜索，以进一步优化其关节角度。

i. 更新种群：将精英个体和后代合并，形成新一代的种群。

(4). 调整参数：根据种群的改善情况动态调整交叉率和变异率，以平衡搜索的探索和利用能力。

(5). 输出结果：输出每一代的最佳适应度和对应的关节角度。最后输出最终找到的最佳解，包括关节角度、适应度和末端执行器的误差。

Step3：模拟退火优化结果

(1). 使用自适应遗传算法找到的最佳解作为模拟退火算法的初始解。

(2). 初始化：设置初始温度、冷却因子和最大迭代次数。

(3). 模拟退火主循环：在每次迭代中，算法首先生成一个新的候选解，这是通过在当前

解的基础上添加小的随机扰动来实现的。随后计算新候选解的适应度（即目标函数值），并与当前解的适应度进行比较。

a. 如果新候选解的适应度更高（即目标函数值更小，因为目标是最小化末端执行器与目标位置之间的欧几里得距离），则接受新候选解作为当前解。

b. 如果新候选解的适应度更低，但满足一定的接受概率（基于温度和适应度差异），则也可能接受新候选解。

(4). 冷却过程：在每次迭代后，根据冷却因子降低温度。这减少了接受较差候选解的概率，使算法逐渐收敛到最优解。

a. 如果当前解在连续迭代中显著改善，则加快冷却速度（即减小冷却因子）。

b. 如果当前解在一段时间内没有显著改善，且温度仍然较高，则减慢冷却速度（即增大冷却因子），以增加搜索的多样性。

(5). 终止条件：当达到最大迭代次数时，模拟退火过程终止。

(6). 输出结果：模拟退火过程结束后，输出最优解（即关节角度）、最优解的适应度（即目标函数值的倒数）以及末端执行器与目标位置之间的最终误差。

Step5：使用改进 D-H 参数法绘制零位状态六自由度机械臂简图用改进 D-H 参数法再一次运行上述三个步骤

Step6：比较标准 D-H 参数法和改进 D-H 参数法所求解结果的误差通过比较标准 D-H 参数法和改进 D-H 参数法所求解结果的误差，得出更优解，以确定整道题使用哪种 D-H 参数法解题

5.3 求解结果

5.3.1 基于标准 D-H 参数法的零位状态六自由度机械臂简图

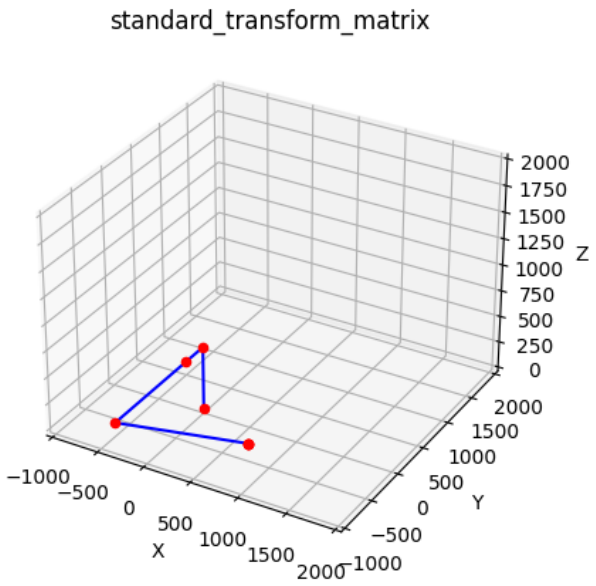


图 1　基于标准 D-H 参数法的零位状态六自由度机械臂简图

5.3.2 基于标准 D-H 参数法的最优关节角路径

表 1

Angles	1	2	3	4	5	6
	1.34	-22.65	80.0	-3.46	-105.58	-8.40

5.3.3 基于标准 D-H 参数法的最优关节角路径末端误差

根据欧几里得距离公式

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{12}$$

求得最优关节角路径末端误差为: 55.49 mm

5.3.4 基于改进 D-H 参数法的零位状态六自由度机械臂的简图

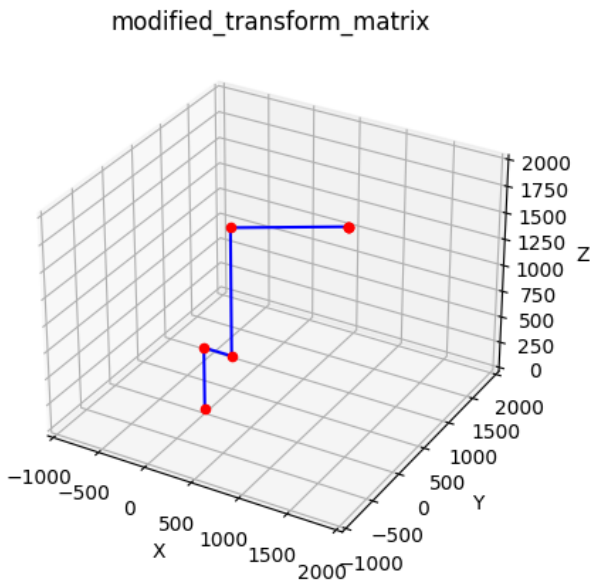


图 2　基于改进 D-H 参数法的零位状态六自由度机械臂简图

5.3.5 基于改进 D-H 参数法的最优关节角路径

表 2

Angles	1	2	3	4	5	6
	-56.29	15.00	-2.81	-137.97	-46.46	69.91

5.3.6 基于改进 D-H 参数法的最优关节角路径末端误差

根据欧几里得距离公式

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{13}$$

求得最优关节角路径末端误差为: 1297.10 mm

5.3.7 误差比较结果

基于标准 D-H 参数法的最优关节角路径末端误差：55.49 mm

基于改进 D-H 参数法的最优关节角路径末端误差：1297.10 mm

易得 55.49 mm < 1297.10 mm

对于本题，标准 D-H 参数法明显优于改进 D-H 参数法。

5.3.8 结论

我们选取标准 D-H 参数法求解该题。

六、 问题二的模型的建立和求解

6.1 问题二模型的建立

6.1.1 关节转动损失能耗

首先，根据物理学知识，我们可以得到能量 W 与转动惯量 I、平均角速度 ω 的关系式:

$$W = 0.5 \times I \times \omega^2 \tag{14}$$

因此，根据问题二提供的各个关节的转动惯量和平均角速度的数据，我们可以得到机械臂关节转动的损失能耗为

$$W = \sum 0.5 \times I_i \times \omega^2 \tag{15}$$

其中 W 为总机械臂关节转动能耗，I_i 为关节 i 的转动惯量。

6.1.2 总能耗损失

机械臂的能耗不仅仅是关节转动损失能耗，还有抓取重物时因为重力势能损失的能耗。

问题二中提供的机械臂质量和末端载重之和为 5kg，因此，我们假设几乎所有的质量都集中在机械臂的末端和末端载重上，就可以从整体上来计算重力势能的损耗:

$$\Delta E_p = m \cdot g \cdot \Delta h \tag{16}$$

其中 ΔE_p 代表重力势能损耗，m 是机械臂质量和末端载重之和，为 5kg, g 是重力系数，Δh 是零位状态和末态使机械臂末端的水平高度差。

因此，机械臂总能耗损失可以表示为:

$$W_{sum} = W + \Delta E_p = \sum 0.5 \times I_i \times \omega^2 + m \cdot g \cdot \Delta h \tag{17}$$

6.1.3 关节角路径优化模型

问题二是一个多约束条件的优化问题，即在问题一的基础上，我们还要对机械臂的总能耗进行约束。

因此得到**约束条件**:

1. 末端误差 E 允许范围 $\pm 200mm$

2.θ_i 的角度必须在题给的范围内变化

3. 总能耗损失必须最小化

约束条件可以用**目标函数**简单表示为

$$L = \beta W_{sum} + \gamma E \tag{18}$$

由此，在第一问的基础上，我们可以得到**问题二的关节角路径优化模型**:

$$\begin{cases} {}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos 0^\circ & \sin(\theta_1)\sin 0^\circ & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos 0^\circ & -\cos(\theta_1)\sin 0^\circ & 0 \\ 0 & \sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ \cdots\cdots \\ {}^{0}_{6}T = {}^{0}_{1}T\,{}^{1}_{2}T \ldots {}^{5}_{6}T \\ \begin{cases} \theta_1 \in (-160^\circ, 160^\circ) \\ \theta_2 \in (-150^\circ, 15^\circ) \\ \theta_3 \in (-200^\circ, 80^\circ) \\ \theta_4 \in (-180^\circ, 180^\circ) \\ \theta_5 \in (-120^\circ, 120^\circ) \\ \theta_6 \in (-180^\circ, 180^\circ) \end{cases} \\ E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \\ E \leq \pm 200mm \\ W_{sum} = \sum 0.5 \times I_i \times \omega^2 + m \cdot g \cdot \Delta h \\ L = \beta W_{sum} + \gamma E \\ L_{min} \end{cases} \tag{19}$$

6.2 问题二模型的求解

该问题的求解步骤和问题一的求解步骤大体一致。

但是在目标函数 $objective function$ 的设定中不仅考虑了末端执行器与目标位置之间的欧几里得距离（位置误差），还考虑了能量消耗。它通过计算关节的动能（基于给定的转动惯量 I 和角速度 ω）以及重力势能的变化来估算能量消耗，并将这些能量消耗作为优化目标的一部分，并通过动态调整能量权重来平衡位置误差和能量消耗之间的权重。

a. 当位置误差较大时，减少能量消耗的权重；

b. 当位置误差较小时，增加能量消耗的权重。

最后增加了轨迹生成部分，生成从初始关节角度到最终关节角度随时间变化的三次多项式轨迹图，以可视化关节角度变化的全过程，增强可读性。

6.3 求解结果

6.3.1 最优关节角位置

表 3

Angles	1	2	3	4	5	6
	0.33	4.40	-92	21.03	-0.84	66.59

6.3.2 最优关节角路径末端误差

根据欧几里得距离公式

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{20}$$

求得最优关节角路径末端误差为: 12.02 mm

6.3.3 最优关节角路径

图三纵坐标为角度，横坐标为时间。

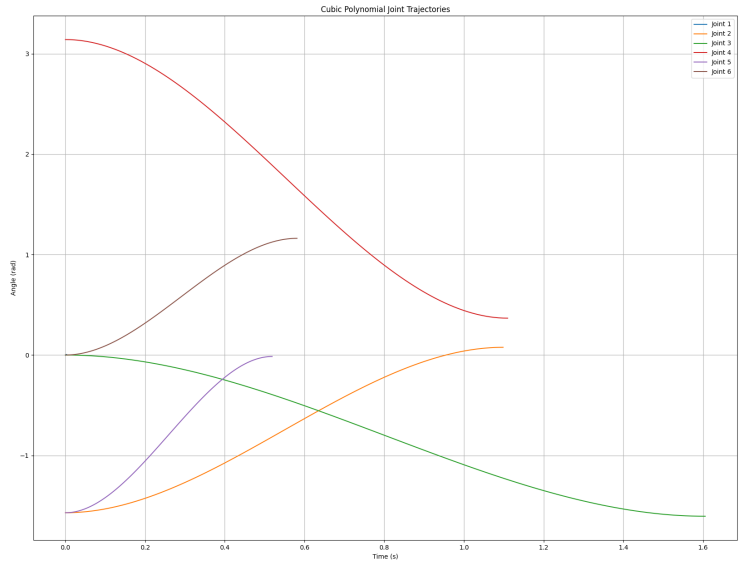


图 3　机械臂最优关节角路径

七、 问题三的模型的建立和求解

7.1 问题三模型的建立

在问题二模型的基础上，机械臂底座需要按网状路线移动，绕过障碍物并抓取目标回到起点。因此，我们必须在保持原有条件最优的情况下实现机械臂移动的最短路径规划。

7.1.1 曼哈顿距离

由于问题三要求机械臂底座必须在栅格图中网格状移动，因此我们可以用二维平面上的曼哈顿距离来计算机械臂移动的路径，从而大大简化问题的求解。

曼哈顿距离是一种简单有效的度量网格状场景的方法，在本题情境的栅格图中相比直接计算具有明显的优势。在二维平面中，曼哈顿距离可以简单理解为两个正交坐标方向的距离之和。我们设栅格图上的两个坐标点分别为 $A(x_1, y_1), B(x_2, y_2)$, 则 A,B 两点之间的曼哈顿距离 L_1 可以表示为

$$L_1 = |x_1 - x_2| + |y_1 - y_2| \tag{21}$$

我们使用曼哈顿距离来计算单步移动距离。

7.1.2 A* 算法

为在栅格图中完成避开障碍物的最优路径规划，我们使用启发式的 A* 算法来建模求解。A* 算法是最有效的静态路网中求解最短路径的搜索算法，其基于启发函数构建了代价函数，既考虑了新结点距离初始点的代价，又考虑了新结点距离目标点的代价。

首先我们建立 A* 算法的路径优劣评价公式

$$f(n) = g(n) + h(n) \tag{22}$$

其中 f(n) 是从初始状态经由状态 n 到目标状态的代价估计，g(n) 是从初始状态到状态 n 的实际代价，h(n) 是从状态 n 到目标状态的最佳路径估计代价。状态 n 即移动的次数，n=1,2,3,...

随后，我们从初始点（即 start）处开始搜索，检查相邻的方格，直到找到目标。以从 start 出发的第一步（即状态 n=1 时）为例，如下图所示，(已知栅格的边长 l 为 200mm);

首先考察 g，由于从 start 到下一个点是斜着走的，可知 $g = \sqrt{2} \times 200mm = 283mm$;

接着考察估计代价 h, 按照 7.1.1 中曼哈顿距离的计算方法，计算只做横向或者纵向移动的累计代价：横向移动了一格，纵向移动了一格，因此 $h = 200 + 200mm = 400mm$;

因此从 start 移动到下一个点的总移动代价是 $f = g + h = 283 + 400mm = 683mm$。

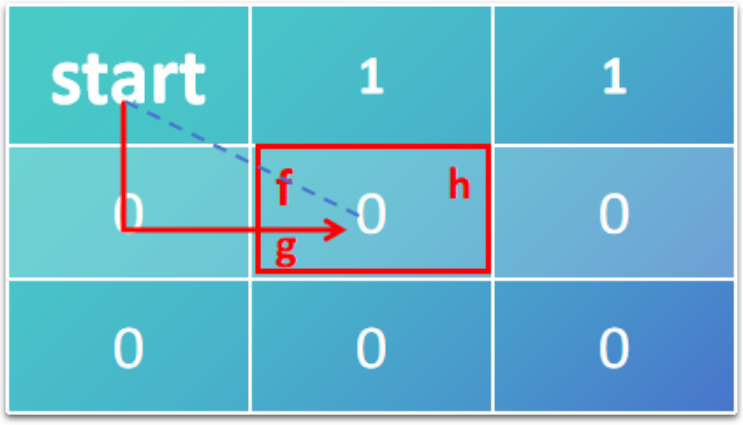


图 4　A* 算法示意图

依次类推，不断重复，直到移动到目标点 end 完成运动，这样我们就得到了最优的机械臂底座移动路径。

7.1.3 问题三路径优化模型

与问题二建立的数学模型相比，求解最优关节角路径的数学模型不变，因此我们只要把求解最优底座移动路径的模型加入到原有的数学模型中即可。

由 7.1.1 和 7.1.2 的分析可知，求解**最优底座移动路径的模型**可以写成：

$$\begin{cases} L_1 = |x_1 - x_2| + |y_1 - y_2| \\ f(n) = g(n) + h(n) \\ l = 200 \end{cases} \tag{23}$$

于是，我们把最优底座移动路径的模型放入到原有的问题二数学模型中，得到**问题**

三的最优模型：

$$
\begin{cases}
L_1 = |x_1 - x_2| + |y_1 - y_2| \\
f(n) = g(n) + h(n) \\
l = 200 \\
{}^{0}_{1}T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos 0^\circ & \sin(\theta_1)\sin 0^\circ & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos 0^\circ & -\cos(\theta_1)\sin 0^\circ & 0 \\ 0 & \sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\cdots\cdots \\
{}^{0}_{6}T = {}^{0}_{1}T\,{}^{1}_{2}T \ldots {}^{5}_{6}T \\
\begin{cases}
\theta_1 \in (-160^\circ, 160^\circ) \\
\theta_2 \in (-150^\circ, 15^\circ) \\
\theta_3 \in (-200^\circ, 80^\circ) \\
\theta_4 \in (-180^\circ, 180^\circ) \\
\theta_5 \in (-120^\circ, 120^\circ) \\
\theta_6 \in (-180^\circ, 180^\circ)
\end{cases} \\
E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \\
E \le \pm 200mm \\
W_{sum} = \sum 0.5 \times I_i \times \omega^2 + m \cdot g \cdot \Delta h \\
L = \beta W_{sum} + \gamma E \\
L_{min}
\end{cases}
\tag{24}
$$

7.2 问题三模型的求解

Step1：使用 A* 算法寻找最优底座移动路径

(1). 初始化：

a. 定义起点和终点：首先，从给定的网格（grid）数据中读取起点（start）和终点（end）的位置。

b. 创建起点和终点节点：将起点和终点转换为 Node 对象，并初始化它们的 g（从起点到当前节点的成本）、h（启发式函数估计从当前节点到终点的成本）、f（总成本，即 g + h）值。

c. 初始化开放列表（open list）和关闭列表（closed list）：开放列表用于存放待考察的节点，而关闭列表用于存放已经考察过的节点。

(2). 算法主循环

a. 从开放列表中选取成本最低的节点：每次循环都从开放列表中选取 f 值最小的节点作为当前节点，并将其从开放列表中移除，添加到关闭列表中。

b. 检查是否到达终点：如果当前节点是终点节点，则算法结束，通过回溯当前节点的父节点来构建从起点到终点的路径。

c. 生成当前节点的邻居节点：考虑当前节点的上、下、左、右四个相邻节点作为邻居节点。

(3). 邻居节点处理

a. 检查邻居节点是否在网格范围内：排除那些超出网格边界的邻居节点。

b. 检查邻居节点是否为障碍物：如果邻居节点对应的网格值为非零（表示障碍物），则忽略该邻居节点。

c. 创建新的邻居节点：对于每个有效的邻居节点，创建一个新的 Node 对象，并将其父节点设置为当前节点。

d. 计算新节点的 g、h、f 值：

新节点的 g 值为父节点的 g 值加上从父节点到新节点的成本（这里假设为 1，因为每次移动的成本相同）；

新节点的 h 值使用曼哈顿距离启发式函数计算;

新节点的 f 值为 g 和 h 之和。

e. 使用曼哈顿距离启发式函数计算；新节点的 f 值为 g 和 h 之和。

f. 检查新节点是否已在开放列表或关闭列表中：

如果新节点已在关闭列表中，则忽略该节点。

如果新节点在开放列表中，但当前路径的成本更高（即 g 值更大），则更新该节点在开放列表中的位置（通过堆排序保证最低成本的节点始终在堆顶），否则添加新节点到开放列表中。

(4). 路径构建:

a. 一旦到达终点节点，算法通过回溯父节点来构建从起点到终点的路径。

b. 将路径中的节点位置反转，因为路径是通过回溯构建的，最初添加的节点实际上是路径的终点。

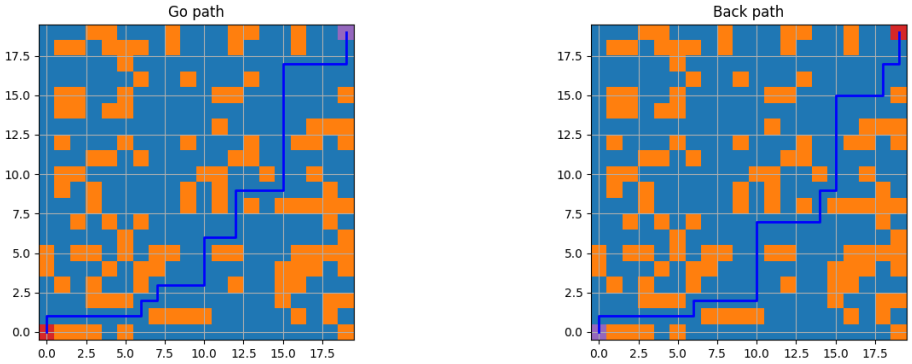
(5). 路径绘制:

使用 matplotlib 库将网格和找到的路径绘制出来，以便直观地查看结果。

Step2：寻找最优关节角路径

同问题二的求解

7.3 求解结果



(a) Go path **(b) Back path**

图 5　Best path

7.3.1 最优底座路径

(0, 0) -> (1, 0) -> (1, 1) -> (1, 2) -> (1, 3) -> (1, 4) -> (1, 5) -> (1, 6) -> (2, 6) -> (2, 7) -> (3, 7) -> (3, 8) -> (3, 9) -> (3, 10) -> (4, 10) -> (5, 10) -> (6, 10) -> (6, 11) -> (6, 12) -> (7, 12) -> (8, 12) -> (9, 12) -> (9, 13) -> (9, 14) -> (9, 15) -> (10, 15) -> (11, 15) -> (12, 15) -> (13, 15) -> (14, 15) -> (15, 15) -> (16, 15) -> (17, 15) -> (17, 16) -> (17, 17) -> (17, 18) -> (17, 19) -> (18, 19) -> (19, 19) -> (18, 19) -> (17, 19) -> (17, 18) -> (16, 18) -> (15, 18) -> (15, 17) -> (15, 16) -> (15, 15) -> (14, 15) -> (13, 15) -> (12, 15) -> (11, 15) -> (10, 15) -> (9, 15) -> (9, 14) -> (8, 14) -> (7, 14) -> (7, 13) -> (7, 12) -> (7, 11) -> (7, 10) -> (6, 10) -> (5, 10) -> (4, 10) -> (3, 10) -> (2, 10) -> (2, 9) -> (2, 8) -> (2, 7) -> (2, 6) -> (1, 6) -> (1, 5) -> (1, 4) -> (1, 3) -> (1, 2) -> (1, 1) -> (1, 0) -> (0, 0)

7.3.2 最优关节角位置

表 4

Angles	1	2	3	4	5	6
	0.33	4.40	-92	21.03	-0.84	66.59

7.3.3 最优关节角路径末端误差

根据欧几里得距离公式

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{25}$$

求得最优关节角路径末端误差为: 12.02 mm

7.3.4 最优关节角路径

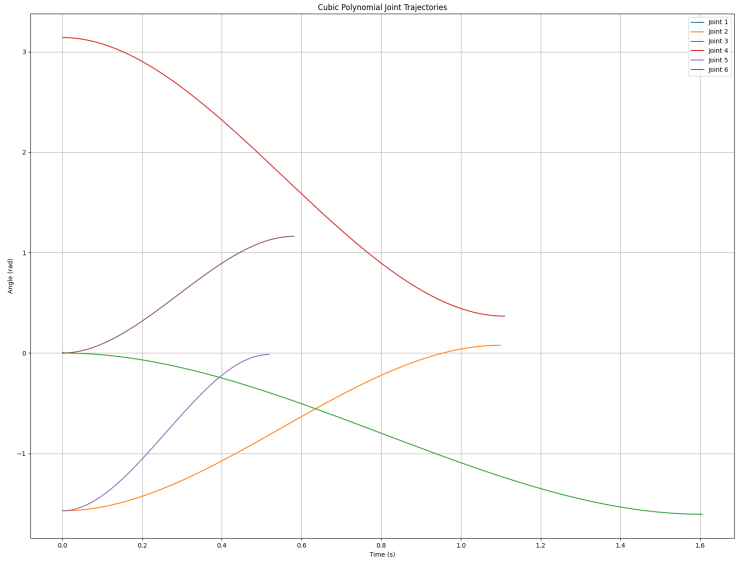


图 6　机械臂最优关节角路径

八、 问题四的模型的建立和求解

8.1 问题四模型的建立

和问题三相比，本题要求在一次任务中完成多个目标货物的抓取，同样回到起点。由于栅格图中同时存在多个障碍物和目标货物，路径的规划也变得多样复杂。这时我们可以用图论中的 **TSP 旅行商问题**来简化本题情境。

8.1.1 经典 TSP 问题

因为本题情境中，目标货物的数量和位置是确定的，起始点和结束点的位置也是确定的，因此其恰好符合经典 TSP 问题的情境：给定 n 个点位，以及这 n 个点位之间的距离，需要找出一条最短的路径，使得恰好能经过每一个点位且仅经过一次。

TSP 问题实际上是组合优化类问题，由于其没有直接的多项式来建模，我们只能使用启发式算法来进行建模求解。

8.1.2 模拟退火迭代寻优

在本题中，我们采用了启发式算法中的模拟退火算法。

首先，将每个目标货物已经由题目给定编号 target1-5, 我们随后对每个目标货物之间的距离进行编号表示，用 $d_(i,j)$ **来表示从 targeti 到 targetj 的曼哈顿距离**。即

$$\begin{cases} d_{(i,j)} = L(targeti - targetj) \\ i,j \in 1,2,3,4,5 \end{cases} \tag{26}$$

随后，我们不断随机选择 i 和 j 的值，通过迭代法尝试随机交换当前路径中的某两个城市，然后计算总路径长度。就这样不断地迭代，不断减小每一个城市之间的距离，尝试找到最短的总路径长度和路径规划方案。

8.1.3 Metropolis 准则

Metropolis 准则是一种采样方法。在使用模拟退火进行迭代时，我们采用 Metropolis 准则来决定是否对原有状态进行更新。

Metropolis 准则的具体公式为：

$$p = \exp\left(-\frac{\Delta E}{k \cdot Tem}\right) \tag{27}$$

其中,p 是接受新状态的概率。ΔE 是新状态与当前状态之间的能量差（或目标函数值的差）。如果新状态的能量低于当前状态，则 $\Delta E < 0$；如果新状态的能量高于当前状态，则 $\Delta E > 0$。k 是玻尔兹曼常数，它是一个物理常数，用于描述热力学系统中的能量与温度之间的关系。Tem 是系统的温度。

8.1.4 问题四路径优化模型

在问题三的路径优化模型基础上，我们只需要加入对 TSP 的建模过程，即可得到**问题四路径优化模型**：

$$
\begin{cases}
d_{(i,j)} = L(targeti - targetj) \\
i, j \in 1,2,3,4,5 \\
p = \exp\left(-\frac{\Delta E}{kT}\right) \\
L_1 = |x_1 - x_2| + |y_1 - y_2| \\
f(n) = g(n) + h(n) \\
l = 200 \\
{}_1^0T = \begin{bmatrix} \cos(\theta_1) & -\sin(\theta_1)\cos 0^\circ & \sin(\theta_1)\sin 0^\circ & 0 \\ \sin(\theta_1) & \cos(\theta_1)\cos 0^\circ & -\cos(\theta_1)\sin 0^\circ & 0 \\ 0 & \sin 0^\circ & \cos 0^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
\cdots\cdots \\
{}_6^0T = {}_1^0T\,{}_2^1T \ldots {}_6^5T \\
\begin{cases}
\theta_1 \in (-160^\circ, 160^\circ) \\
\theta_2 \in (-150^\circ, 15^\circ) \\
\theta_3 \in (-200^\circ, 80^\circ) \\
\theta_4 \in (-180^\circ, 180^\circ) \\
\theta_5 \in (-120^\circ, 120^\circ) \\
\theta_6 \in (-180^\circ, 180^\circ)
\end{cases} \\
E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \\
E \le \pm 200mm \\
W_{sum} = \sum 0.5 \times I_i \times \omega^2 + m \cdot g \cdot \Delta h \\
L = \beta W_{sum} + \gamma E \\
L_{min}
\end{cases}
\tag{28}
$$

8.2 问题四模型的求解

Step1：经典 TSP 问题的模拟退火 A* 算法

(1). 初始化: 现假定每个任务点为一座城市，由障碍物组成的场地假定为迷宫（maze）。

a. 城市列表（cities）：从迷宫网格（maze）中读取城市的位置，并将起始点（Start）和城市目标点（如”target1”, ”target2” 等）的位置记录到列表中。

b. 当前路径（current path）：初始化为一个长度为 n 的数组，表示城市的访问顺序，其中 n 是城市的数量。

c. 最佳路径（best path）和最佳路径长度（best length）：初始化为当前路径的副本及其长度。

d. 温度（temperature）：设置为初始温度 initial temperature。

(2). 模拟退火主循环

循环执行 iterations 次迭代，每次迭代尝试通过随机交换当前路径中的两个城市来生成新的路径。

(3). 生成新路径

在当前路径中随机选择两个不同的城市索引 i 和 j，并交换这两个城市在路径中的位置，生成新的路径 new path。

(4). 计算路径长度

使用 $pathlength$ 函数分别计算当前路径 current path 和新路径 new path 的总长度。这里考虑的是城市之间的曼哈顿距离，但由于迷宫的存在，实际路径长度会不同。

(5). 接受准则

a. 如果新路径的长度小于旧路径的长度，则无条件接受新路径。

b. 如果新路径的长度不小于旧路径的长度，则根据 Metropolis 准则以一定的概率接受新路径。这个概率取决于新旧路径的长度差和当前温度。

(6). 更新最佳路径

如果新路径的长度小于当前记录的最佳路径长度，则更新最佳路径及其长度。(7). 冷却

每次迭代后，根据 cooling rate 降低温度。

(8). 生成最终路径（考虑迷宫）

a. 在得到城市访问顺序的最佳路径后，使用 A* 算法（$astar$ 函数）在迷宫中找到从当前城市到下一个城市的实际路径。

b. 遍历最佳路径中的每个城市对，使用 A* 算法找到它们之间的路径，并将这些路径连接起来形成最终的完整路径。

(9). 输出结果

输出最佳路径的长度、顺序以及通过迷宫的实际路径。

Step2：寻找最优关节角路径

同问题二的求解

8.3 求解结果

8.3.1 最优底座移动路径

最佳路径长度: 88

最佳路径顺序: Start -> target1 -> target2 -> target3 -> target5 -> target4 -> Start

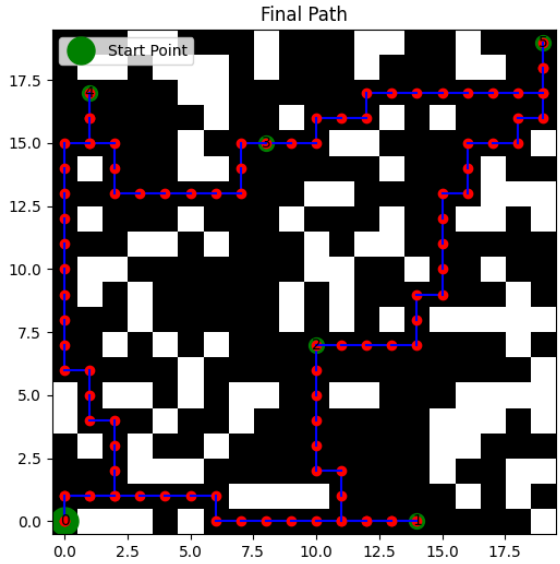


图 7　栅格图可视化结果

8.3.2 最优关节角位置

表 5

Angles	1	2	3	4	5	6
	0.33	4.40	-92	21.03	-0.84	66.59

8.3.3 最优关节角路径末端误差

根据问题一中得到的末端误差计算公式

$$E = \sqrt{(1500 - x)^2 + (1200_0 - y)^2 + (200 - z)^2} \tag{29}$$

求得最优关节角路径末端误差为: 12.02 mm

8.3.4 最优关节角路径

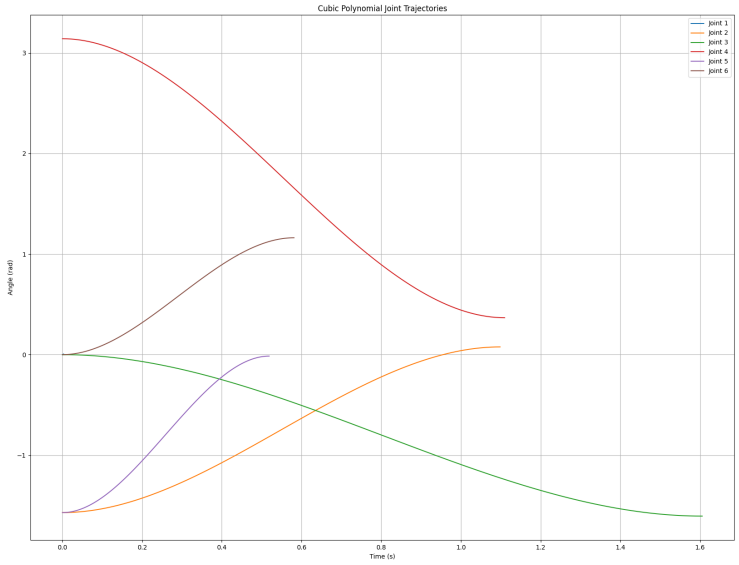


图 8　机械臂最优关节角路径

8.3.5 最优关节角路径总能耗

最优关节角路径总能耗: 24.11 J

九、 模型的分析与检验

9.1 误差分析

上述模型的误差主要来自两个方面:

- 方面 1

运用了自适应遗传算法和模拟退火算法这两个智能算法，由于引入了随机数，导致算法在迭代过程中可能会出现较大的波动，对最终结果的误差产生不可避免的影响，容易出现波动，不稳定。

- 方面 2

由于题给条件不足，上述模型假设机械臂本身的质量远远小于末端载重，即假设末端载重自身的重量是 5kg、机械臂本身的质量忽略不计，即机械臂和末端载重的整体重心就是末端载重的重心，也即整体消耗的重力势能等于末端载重消耗的重力势能。但是现实中并不可能出现这种理想化机械臂。因此在现实中，机械臂和末端载重的整体重心并不是末端载重的重心，而是在整体空间中的一点，这就导致了机械

臂和末端载重整体重心变化的高度并不一定等于末端载重重心变化的高度，根据重力势能表达式

$$\Delta E_p = m \cdot g \cdot \Delta h \tag{30}$$

可得，整体消耗的重力势能并不一定等于末端载重消耗的重力势能。因此，消耗的重力势能的理想化在一定程度上影响了结果的准确性，导致误差的增大。

十、 模型的评价

10.1 模型的优点

- 优点 1

灵活性高：用到一系列算法，如自适应遗传算法、模拟退火算法、A* 算法等。这些算法可以应用于多种优化问题，包括但不限于路径规划、参数优化等。

- 优点 2

全局搜索能力：模拟退火和遗传算法具有跳出局部最优解的能力，能够探索更大的解空间，从而可能找到全局最优解。

- 优点 3

自适应性：遗传算法通过调整交叉率和变异率等参数，可以自适应地改进搜索性能。

- 优点 4

并行处理能力：遗传算法等进化算法可以并行计算，提高计算效率。

10.2 模型的缺点

- 缺点 1

参数敏感性：模拟退火和遗传算法的性能对参数设置（如初始温度、冷却率、交叉率、变异率等）非常敏感，需要仔细调整。

- 缺点 2

计算成本高：对于大规模问题，这些算法的计算成本可能非常高，需要较长的运行时间。

- 缺点 3

收敛速度不一：算法收敛速度可能较慢，特别是在问题规模较大或解空间复杂时。

- 缺点 4

结果随机性：由于算法中涉及随机操作（如变异、交叉），每次运行结果可能不同，难以保证结果的稳定性和可重复性。

参考文献

[1] 张博伦, 于涛. 六自由度机械臂轨迹规划方法的研究[J]. 机电产品开发与创新, 2024,37(1):103-106.

[2] 张海涛, 程荫杭. 基于 A* 算法的全局路径搜索[J]. 微计算机信息, 2007(17):3.

附录 A　文件列表

文件名	功能描述
Q1.py	问题一程序代码
Q2.py	问题二程序代码
Q3.py	问题三程序代码
Q4.py	问题四程序代码
Q31.png	问题一求解结果 1
Q32.png	问题一求解结果 2
Q2.png	问题二求解结果
Q3_go.png	问题三求解结果
Q3_back.png	问题三求解结果
Q4.png	问题四求解结果

附录 B　代码

Q1.py

```
import numpy as np
import matplotlib.pyplot as plt
import random


# 计算变换矩阵,传统D-H法
def standard_calc_transform_matrix(param):
    theta, d, a, alpha = param['theta'], param['d'], param['a'], param['alpha']

    return np.array([[np.cos(theta),-np.sin(theta)*np.cos(alpha),np.sin(theta)*np.sin(alpha),a*np.cos(theta)],
                     [np.sin(theta),np.cos(theta)*np.cos(alpha),-np.cos(theta)*np.sin(alpha),a*np.sin(theta)],
                     [0,np.sin(alpha),np.cos(alpha),d],
                     [0,0,0,1]])


```

```python
# 目标函数：计算末端位置与目标位置之间的欧几里得距离
def objective_function(angles, target_position):
    T = np.eye(4)
    for i, param in enumerate(params[1:], start=1):
        param['theta'] = angles[i-1]  # 更新关节角
        T = T @ standard_calc_transform_matrix(param)
    end_effector_position = T[:3, 3]
    return float(np.linalg.norm(end_effector_position - target_position))

# 辅助函数：确保角度在给定的范围内
def clip_angle(theta, lower_limit, high_limit):
    """确保角度在给定的范围内"""
    return max(lower_limit, min(high_limit, theta))

# 局部搜索：梯度下降
def local_search(individual, learning_rate=0.001, iterations=1000):
    for _ in range(iterations):
        gradient = np.zeros_like(individual)
        for i in range(len(individual)):
            epsilon = 1e-10
            individual_plus = individual.copy()
            individual_plus[i] += epsilon
            individual_minus = individual.copy()
            individual_minus[i] -= epsilon
            gradient[i] = (objective_function(individual_plus, target_position) -
                           objective_function(individual_minus, target_position)) / (2 * epsilon)

        # 更新个体
        individual -= learning_rate * gradient
        # 确保角度在有效范围内
        for i in range(len(individual)):
```

```
47              individual[i] = clip_angle(individual[i], params[i
    ]['lower_limit'], params[i]['high_limit'])
48      return individual
49
50  # 多点交叉
51  def multipoint_crossover(parent1, parent2, num_points=2):
52      crossover_points = sorted(random.sample(range(1, len(
    parent1)), num_points))
53      child1, child2 = parent1.copy(), parent2.copy()
54
55      for i in range(num_points):
56          start = crossover_points[i]
57          end = crossover_points[(i+1) % num_points] if i <
    num_points - 1 else len(parent1)
58          child1[start:end], child2[start:end] = child2[start:
    end], child1[start:end]
59      return child1, child2
60
61  # 定义用于调整参数的函数
62  def adjust_parameters(crossover_rate, mutation_rate,
    best_fitness, prev_best_fitness):
63      improvement_threshold = 0.001  # 改善阈值
64      diversity_threshold = 0.01  # 多样性阈值
65
66      # 检查是否有显著改善
67      if best_fitness > prev_best_fitness +
    improvement_threshold:
68          mutation_rate *= 0.09  # 减少变异率
69      else:
70          mutation_rate *= 1.01  # 增加变异率
71
72      # 检查种群多样性
73      if mutation_rate > 0.5:
74          crossover_rate *= 0.09  # 减少交叉率
75      else:
```

```
76          crossover_rate *= 1.01  # 增加交叉率
77
78      # 限制参数范围
79      crossover_rate = max(min(crossover_rate, 1.0), 0.0)
80      mutation_rate = max(min(mutation_rate, 1.0), 0.0)
81
82      return crossover_rate, mutation_rate
83
84  # 定义模拟退火算法
85  def adaptive_simulated_annealing(initial_solution,
    initial_temperature, cooling_factor, max_iterations,
    objective_function, target_position, improvement_threshold
    =0.001, diversity_threshold=0.01):
86      current_solution = initial_solution
87      current_fitness = objective_function(current_solution,
    target_position)
88      best_solution = current_solution
89      best_fitness = current_fitness
90
91      temperature = initial_temperature
92      prev_best_fitness = best_fitness
93
94      for iteration in range(max_iterations):
95          # Generate a new candidate solution
96          candidate_solution = current_solution + np.random.
    uniform(-0.01, 0.01, size=len(current_solution))
97          for i in range(len(candidate_solution)):
98              candidate_solution[i] = clip_angle(
    candidate_solution[i], params[i]['lower_limit'], params[i]['
    high_limit'])
99
100         candidate_fitness = objective_function(
    candidate_solution, target_position)
101
102         # Calculate the change in fitness
```

```
        delta_fitness = candidate_fitness - current_fitness

        # Acceptance probability
        if delta_fitness < 0 or np.exp(-delta_fitness / temperature) > np.random.rand():
            current_solution = candidate_solution
            current_fitness = candidate_fitness

            # Update the best solution
            if current_fitness < best_fitness:
                best_solution = current_solution
                best_fitness = current_fitness
                prev_best_fitness = best_fitness

        # Adaptive cooling
        if best_fitness > prev_best_fitness + improvement_threshold:
            cooling_factor *= 0.9  # Faster cooling
        elif temperature > diversity_threshold:
            cooling_factor *= 1.1  # Slower cooling

        # Cool down the temperature
        temperature *= cooling_factor

    return best_solution, best_fitness


# 定义连杆参数
params = [
    {'a': 0,   'alpha': 0,          'd': 600, 'theta': 0,      'lower_limit':-160 * np.pi / 180, 'high_limit':160 * np.pi / 180},  # 连杆1
    {'a': 300, 'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi / 2,'lower_limit':-150 * np.pi / 180, 'high_limit':15 * np.pi / 180},  # 连杆2
```

```
    {'a': 1200,'alpha': 0,          'd': 0,   'theta': 0,
       'lower_limit':-200 * np.pi / 180, 'high_limit':80 * np.
   pi / 180},  # 连杆3
    {'a': 300, 'alpha': -np.pi / 2, 'd': 1200,'theta': np.pi,
       'lower_limit':-180 * np.pi / 180, 'high_limit':180 * np.
   pi / 180},  # 连杆4
    {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi
   / 2,'lower_limit':-120 * np.pi / 180, 'high_limit':120 * np.
   pi / 180},  # 连杆5
    {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': 0,
       'lower_limit':-180 * np.pi / 180, 'high_limit':180 * np.
   pi / 180}  # 连杆6
]
target_position = np.array([1500, 1200, 200])
num_joints = len(params)

# ----------Classic DH----------
# 绘制机械臂
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# 初始化变换矩阵
T=np.eye(4)
T_i = np.eye(4)

# 计算底座的变换矩阵
T_i = T_i @ standard_calc_transform_matrix({'a': 0, 'alpha':
   0, 'd': 0, 'theta': 0  })

points = [T_i[:3, 3]]  # 基座位置

# 绘制底座点
ax.scatter(*points[0], color="r")

# 绘制每个连杆
```

```python
for param in params:
    T_i = T_i @ standard_calc_transform_matrix(param)
    position_i = T_i[:3, 3]
    points.append(position_i)
    print(points[-1])
    ax.scatter(*position_i, color="r")
    # 绘制连杆线段
    ax.plot3D(*zip(points[-2], points[-1]), color="b")

# 绘制末端执行器点
ax.scatter(*points[-1], color="r")

# 设置坐标轴
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_xlim([-1000, 2000])
ax.set_ylim([-1000, 2000])
ax.set_zlim([0, 2000])
plt.title("standard_transform_matrix")
plt.savefig("Q1_standard_transform_matrix.png")
plt.show()


# 自适应遗传算法
# 遗传算法参数
population_size = 100
num_generations = 50
crossover_rate = 0.8
mutation_rate = 0.1
elite_count = 2  # 精英个体的数量
num_crossover_points = 2  # 多点交叉的点数

# 初始化种群
population = [
```

```
    [clip_angle(np.random.uniform(params[i]['lower_limit'], params[i]['high_limit']),
                params[i]['lower_limit'], params[i]['high_limit'])
     for i in range(num_joints)]
    for _ in range(population_size)
]

# 初始化变量用于记录最佳解
best_solution = None
best_fitness = float('-inf')
best_solution_generation = 0

# 遗传算法主循环
prev_best_fitness = 0.0
for generation in range(num_generations):
    # 评估适应度
    fitness = [1.0 / (1.0 + objective_function(individual, target_position)) for individual in population]

    # 排序种群
    sorted_indices = np.argsort(fitness)[::-1]
    sorted_population = [population[i] for i in sorted_indices]
    sorted_fitness = [fitness[i] for i in sorted_indices]

    # 更新最佳解
    if sorted_fitness[0] > best_fitness:
        best_fitness = sorted_fitness[0]
        best_solution = sorted_population[0]
        best_solution_generation = generation

    # 精英保留
    elite_population = sorted_population[:elite_count]
    elite_fitness = sorted_fitness[:elite_count]
```

```
224
225     # 选择操作：轮盘赌选择，去除精英个体
226     probabilities = [f / sum(sorted_fitness[elite_count:]) for
        f in sorted_fitness[elite_count:]]
227
228     # 使用 random.choices 进行选择
229     selected_population = random.choices(sorted_population[
    elite_count:],
230                                          weights=probabilities
    ,
231                                          k=population_size -
    elite_count)
232
233     # 交叉操作
234     offspring = []
235     while len(offspring) < population_size - elite_count:
236         if random.random() < crossover_rate:
237             parent1, parent2 = random.choices(
    selected_population, weights=probabilities, k=2)
238             child1, child2 = multipoint_crossover(parent1,
    parent2, num_crossover_points)
239             offspring.extend([child1, child2])
240
241     # 变异操作
242     for individual in offspring:
243         for j in range(num_joints):
244             if random.random() < mutation_rate:
245                 # 变异
246                 individual[j] += random.uniform(-0.01, 0.01)
247                 # 确保角度在有效范围内
248                 individual[j] = clip_angle(individual[j],
    params[j]['lower_limit'], params[j]['high_limit'])
249
250     # 局部搜索
251     offspring = [local_search(individual) for individual in
```

```
    offspring]

    # 更新种群
    population = elite_population + offspring[:population_size
    - elite_count]

    # 输出当前最佳解
    print(
        f"Generation {generation}: Best Fitness {
    sorted_fitness[0]}, Best Angles {sorted_population[0]}")

    # 调整参数
    crossover_rate, mutation_rate = adjust_parameters(
    crossover_rate, mutation_rate, sorted_fitness[0],
    prev_best_fitness)
    prev_best_fitness = sorted_fitness[0]

# 将最终解从弧度转换为角度
best_solution_degrees = [angle * 180 / np.pi for angle in
    best_solution]

# 输出最佳解
print("Best Solution Found:")
print("Generation:", best_solution_generation)
print("Joint Angles:", best_solution_degrees)
print("Fitness:", best_fitness)
print("End-Effector Error:", objective_function(best_solution,
    target_position))


# Adaptive Simulated Annealing Parameters
initial_temperature = 100
cooling_factor = 0.9995
max_iterations = 100000

```

```
# Apply Adaptive Simulated Annealing to the best solution found by the GA
best_solution, best_fitness = adaptive_simulated_annealing(best_solution, initial_temperature, cooling_factor, max_iterations, objective_function, target_position)

# 将最终解从弧度转换为角度
best_solution_degrees = [angle * 180 / np.pi for angle in best_solution]

# Output the refined best solution
print("Refined Best Solution Found:")
print("Joint Angles:", best_solution_degrees)
print("Fitness:", best_fitness)
print("End-Effector Error:", objective_function(best_solution, target_position))
```

Q2.py

```
import numpy as np
import matplotlib.pyplot as plt
import random


# 计算变换矩阵,传统D-H法
def standard_calc_transform_matrix(param):
    theta, d, a, alpha = param['theta'], param['d'], param['a'], param['alpha']

    return np.array([[np.cos(theta),-np.sin(theta)*np.cos(alpha),np.sin(theta)*np.sin(alpha),a*np.cos(theta)],
                     [np.sin(theta),np.cos(theta)*np.cos(alpha),-np.cos(theta)*np.sin(alpha),a*np.sin(theta)],
                     [0,np.sin(alpha),np.cos(alpha),d],
                     [0,0,0,1]])

```

```
# 目标函数：计算末端位置与目标位置之间的欧几里得距离和能耗
def objective_function(angles, target_position):
    T = np.eye(4)
    energy_consumption = gravitational_potential
    for i, param in enumerate(params[1:], start=1):
        param['theta'] = angles[i-1]
        T = T @ standard_calc_transform_matrix(param)
        energy_consumption += 0.5 * param['I'] * param['omega']**2
    end_effector_position = T[:3, 3]
    position_error = np.linalg.norm(end_effector_position - target_position)
    if position_error > 200:
        position_error = 200  # 误差阈值处理
    # 动态调整能量权重
    energy_weight = 0.01 + 0.001 * max(0, 200 - position_error) / 200
    total_error = position_error + energy_weight * energy_consumption
    return float(total_error)

# 辅助函数：确保角度在给定的范围内
def clip_angle(theta, lower_limit, high_limit):
    #确保角度在给定的范围内
    return max(lower_limit, min(high_limit, theta))

# 局部搜索：梯度下降
def local_search(individual, learning_rate=0.01, iterations=100):
    for _ in range(iterations):
        gradient = np.zeros_like(individual)
        for i in range(len(individual)):
            epsilon = 1e-16
            individual_plus = individual.copy()
            individual_plus[i] += epsilon
```

```
            individual_minus = individual.copy()
            individual_minus[i] -= epsilon
            gradient[i] = (objective_function(individual_plus, target_position) -
                           objective_function(individual_minus, target_position)) / (2 * epsilon)

        # 更新个体
        individual -= learning_rate * gradient
        # 确保角度在有效范围内
        for i in range(len(individual)):
            individual[i] = clip_angle(individual[i], params[i]['lower_limit'], params[i]['high_limit'])
    return individual

# 多点交叉
def multipoint_crossover(parent1, parent2, num_points=2):
    crossover_points = sorted(random.sample(range(1, len(parent1)), num_points))
    child1, child2 = parent1.copy(), parent2.copy()

    for i in range(num_points):
        start = crossover_points[i]
        end = crossover_points[(i+1) % num_points] if i < num_points - 1 else len(parent1)
        child1[start:end], child2[start:end] = child2[start:end], child1[start:end]
    return child1, child2

# 定义用于调整参数的函数
def adjust_parameters(crossover_rate, mutation_rate, best_fitness, prev_best_fitness):
    improvement_threshold = 0.01  # 改善阈值
    diversity_threshold = 0.01  # 多样性阈值

```

```
    # 检查是否有显著改善
    if best_fitness > prev_best_fitness + improvement_threshold:
        mutation_rate *= 0.9  # 减少变异率
    else:
        mutation_rate *= 1.1  # 增加变异率

    # 检查种群多样性
    if mutation_rate > 0.5:
        crossover_rate *= 0.9  # 减少交叉率
    else:
        crossover_rate *= 1.1  # 增加交叉率

    # 限制参数范围
    crossover_rate = max(min(crossover_rate, 1.0), 0.0)
    mutation_rate = max(min(mutation_rate, 1.0), 0.0)

    return crossover_rate, mutation_rate

# 定义模拟退火算法
def adaptive_simulated_annealing(initial_solution, initial_temperature, cooling_factor, max_iterations, objective_function, target_position, improvement_threshold=0.001, diversity_threshold=0.01):
    current_solution = initial_solution
    current_fitness = objective_function(current_solution, target_position)
    best_solution = current_solution
    best_fitness = current_fitness

    temperature = initial_temperature
    prev_best_fitness = best_fitness

    for iteration in range(max_iterations):
        # Generate a new candidate solution
```

```
        candidate_solution = current_solution + np.random.uniform(-0.01, 0.01, size=len(current_solution))
        for i in range(len(candidate_solution)):
            candidate_solution[i] = clip_angle(candidate_solution[i], params[i]['lower_limit'], params[i]['high_limit'])

        candidate_fitness = objective_function(candidate_solution, target_position)

        # Calculate the change in fitness
        delta_fitness = candidate_fitness - current_fitness

        # Acceptance probability
        if delta_fitness < 0 or np.exp(-delta_fitness / temperature) > np.random.rand():
            current_solution = candidate_solution
            current_fitness = candidate_fitness

            # Update the best solution
            if current_fitness < best_fitness:
                best_solution = current_solution
                best_fitness = current_fitness
                prev_best_fitness = best_fitness

        # Adaptive cooling
        if best_fitness > prev_best_fitness + improvement_threshold:
            cooling_factor *= 0.09  # Faster cooling
        elif temperature > diversity_threshold:
            cooling_factor *= 1.01  # Slower cooling

        # Cool down the temperature
        temperature *= cooling_factor

```

```
    return best_solution, best_fitness

def generate_trajectory(start_angle, end_angle, duration, time_steps):
    """
    Generates a cubic polynomial trajectory from start_angle to end_angle over a given duration.

    :param start_angle: Initial angle of the joint.
    :param end_angle: Final angle of the joint.
    :param duration: Total time for the trajectory.
    :param time_steps: Number of time steps to divide the trajectory into.
    :return: A list of angles at each time step.
    """
    # Coefficients for the cubic polynomial
    a0 = start_angle
    a1 = 0
    a2 = 3 * (end_angle - start_angle) / (duration ** 2)
    a3 = -2 * (end_angle - start_angle) / (duration ** 3)

    # Time vector
    t = np.linspace(0, duration, time_steps)

    # Calculate angles at each time step
    angles = a0 + a1 * t + a2 * t ** 2 + a3 * t ** 3

    return angles

# 定义连杆参数
params = [
    {'a': 0,   'alpha': 0,          'd': 600, 'theta': 0,
        'lower_limit': -np.pi*160/180, 'high_limit': np.pi*160/180, 'I': 0.5, 'omega': 2.0},  # 连杆1
    {'a': 300, 'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi
```

```
    / 2, 'lower_limit': -np.pi*150/180, 'high_limit': np.pi
    *15/180,  'I': 0.3, 'omega': 1.5},  # 连杆2
     {'a': 1200,'alpha': 0,          'd': 0,   'theta': 0,
         'lower_limit': -np.pi*200/180, 'high_limit': np.pi
    *80/180,  'I': 0.4, 'omega': 1.0},  # 连杆3
     {'a': 300, 'alpha': -np.pi / 2, 'd': 1200,'theta': np.pi,
         'lower_limit': -np.pi*180/180, 'high_limit': np.pi
    *180/180, 'I': 0.6, 'omega': 2.5},  # 连杆4
     {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi
    / 2, 'lower_limit': -np.pi*120/180, 'high_limit': np.pi
    *120/180, 'I': 0.2, 'omega': 3.0},  # 连杆5
     {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': 0,
         'lower_limit': -np.pi*180/180, 'high_limit': np.pi
    *180/180, 'I': 0.4, 'omega': 2.0}   # 连杆6
]
target_position = np.array([1500, 1200, 200])
num_joints = len(params)

# 初始化变换矩阵
T=np.eye(4)
T_i=np.eye(4) @ standard_calc_transform_matrix({'a': 0, 'alpha
    ': 0, 'd': 0, 'theta': 0  })
for param in params:
    T_i = T_i @ standard_calc_transform_matrix(param)
end_position = T_i[:3, 3]
change_h=(end_position[2]-target_position[2])/1000
M = 5
g = 10
gravitational_potential= M * g * change_h

# 自适应遗传算法
# 遗传算法参数
population_size = 100
num_generations = 50
crossover_rate = 0.8
```

```
mutation_rate = 0.1
elite_count = 2  # 精英个体的数量
num_crossover_points = 2  # 多点交叉的点数

# 初始化种群
population = [
    [clip_angle(np.random.uniform(params[i]['lower_limit'], params[i]['high_limit']),
                params[i]['lower_limit'], params[i]['high_limit'])
     for i in range(num_joints)]
    for _ in range(population_size)
]

# 初始化变量用于记录最佳解
best_solution = None
best_fitness = float('-inf')
best_solution_generation = 0

# 遗传算法主循环
prev_best_fitness = 0.0
for generation in range(num_generations):
    # 评估适应度
    fitness = [1.0 / (1.0 + objective_function(individual, target_position)) for individual in population]

    # 排序种群
    sorted_indices = np.argsort(fitness)[::-1]
    sorted_population = [population[i] for i in sorted_indices]
    sorted_fitness = [fitness[i] for i in sorted_indices]

    # 更新最佳解
    if sorted_fitness[0] > best_fitness:
        best_fitness = sorted_fitness[0]
```

```
        best_solution = sorted_population[0]
        best_solution_generation = generation

    # 精英保留
    elite_population = sorted_population[:elite_count]
    elite_fitness = sorted_fitness[:elite_count]

    # 选择操作：轮盘赌选择，去除精英个体
    probabilities = [f / sum(sorted_fitness[elite_count:]) for
    f in sorted_fitness[elite_count:]]

    # 使用 random.choices 进行选择
    selected_population = random.choices(sorted_population[
elite_count:],
                                         weights=probabilities
,
                                         k=population_size -
elite_count)

    # 交叉操作
    offspring = []
    while len(offspring) < population_size - elite_count:
        if random.random() < crossover_rate:
            parent1, parent2 = random.choices(
selected_population, weights=probabilities, k=2)
            child1, child2 = multipoint_crossover(parent1,
parent2, num_crossover_points)
            offspring.extend([child1, child2])

    # 变异操作
    for individual in offspring:
        for j in range(num_joints):
            if random.random() < mutation_rate:
                # 变异
                individual[j] += random.uniform(-0.01, 0.01)
```

```
                # 确保角度在有效范围内
                individual[j] = clip_angle(individual[j], params[j]['lower_limit'], params[j]['high_limit'])

    # 局部搜索
    offspring = [local_search(individual) for individual in offspring]

    # 更新种群
    population = elite_population + offspring[:population_size - elite_count]

    # 输出当前最佳解
    print(
        f"Generation {generation}: Best Fitness {sorted_fitness[0]}, Best Angles {sorted_population[0]}")

    # 调整参数
    crossover_rate, mutation_rate = adjust_parameters(crossover_rate, mutation_rate, sorted_fitness[0], prev_best_fitness)
    prev_best_fitness = sorted_fitness[0]

best_solution_degrees = [angle * 180 / np.pi for angle in best_solution]
# 输出最佳解
print("Best Solution Found:")
print("Generation:", best_solution_generation)
print("Joint Angles:", best_solution_degrees)
print("Fitness:", best_fitness)
print("End-Effector Error:", objective_function(best_solution, target_position))


# Adaptive Simulated Annealing Parameters
```

```
initial_temperature = 100
cooling_factor = 0.995
max_iterations = 10000
# Apply Adaptive Simulated Annealing to the best solution found by the GA
best_solution, best_fitness = adaptive_simulated_annealing(best_solution, initial_temperature, cooling_factor, max_iterations, objective_function, target_position)
best_solution_degrees = [angle * 180 / np.pi for angle in best_solution]

# Output the refined best solution
print("Refined Best Solution Found:")
print("Joint Angles:", best_solution_degrees)
print("Fitness:", best_fitness)
print("End-Effector Error:", objective_function(best_solution, target_position))

time_steps = 100  # 时间步长
joint_trajectory=[]

# 绘制轨迹
plt.figure(figsize=(20,15))
for i in range(len(best_solution)):
    joint_trajectory=generate_trajectory(params[i]["theta"], best_solution[i], abs(best_solution[i]-params[i]["theta"])/params[i]["omega"], time_steps)
    plt.plot(np.linspace(0, abs(best_solution[i]-params[i]["theta"])/params[i]["omega"], time_steps), joint_trajectory, label=f'Joint {i+1}')
plt.xlabel('Time (s)')
plt.ylabel('Angle (rad)')
plt.title('Cubic Polynomial Joint Trajectories')
plt.legend()
plt.grid(True)
```

```
plt.savefig('Q2_trajectory.png')
plt.show()
```

Q3.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
import heapq
from matplotlib.colors import ListedColormap


# ------------------------A*算法寻找最优底座移动路径函数------------------------
class Node:
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position

        self.g = 0  # Cost from start to node
        self.h = 0  # Heuristic cost from node to goal
        self.f = 0  # Total cost

    def __lt__(self, other):
        return self.f < other.f

    def __eq__(self, other):
        return self.position == other.position


#曼哈顿距离
def heuristic(node, goal):
    # Manhattan distance heuristic
    return abs(node.position[0] - goal.position[0]) + abs(node.position[1] - goal.position[1])
```

```
30
31
32  # A*算法
33  def a_star_search(grid, start, end):
34      # Initialize open and closed lists
35      open_list = []
36      closed_list = []
37
38      # Create start and end nodes
39      start_node = Node(None, start)
40      start_node.g = start_node.h = start_node.f = 0
41      end_node = Node(None, end)
42      end_node.g = end_node.h = end_node.f = 0
43
44      # Add the start node
45      heapq.heappush(open_list, start_node)
46
47      while len(open_list) > 0:
48          # Get the current node with the lowest total cost
49          current_node = heapq.heappop(open_list)
50          closed_list.append(current_node)
51
52          # Check if we have reached the goal
53          if current_node == end_node:
54              path = []
55              current = current_node
56              while current is not None:
57                  path.append(current.position)
58                  current = current.parent
59              return path[::-1]  # Return reversed path
60
61          # Generate children
62          (x, y) = current_node.position
63          neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y
    + 1)]  # Adjacent squares
```

```
64
65          for next in neighbors:
66              # Check if it's within the grid boundaries
67              if next[0] > (len(grid) - 1) or next[0] < 0 or
    next[1] > (len(grid[len(grid) - 1]) - 1) or next[1] < 0:
68                  continue
69
70              # Check if it's an obstacle
71              if grid[next[0]][next[1]] != 0:
72                  continue
73
74              # Create a neighbor node
75              new_node = Node(current_node, next)
76
77              # Append the new node to the open list
78              if new_node in closed_list:
79                  continue
80
81              # Compute the tentative g value
82              new_node.g = current_node.g + 1
83              new_node.h = heuristic(new_node, end_node)
84              new_node.f = new_node.g + new_node.h
85
86              # Check if the node is already in the open list
87              for index, item in enumerate(open_list):
88                  if new_node == item and new_node.g >= item.g:
89                      continue
90
91              # Add the child to the open list
92              heapq.heappush(open_list, new_node)
93
94      return None  # No path found
95
96
97  def plot_path(grid, path, start, end, state):
```

```
    # Define colors for different parts of the grid
    colors = {
        0: 'white',  # Free space
        1: 'black',  # Obstacle
        'path': 'blue',  # Path
        'start': 'green',  # Start point
        'end': 'red'  # End point
    }

    # Create a copy of the grid for plotting
    plot_grid = [[colors[cell] for cell in row] for row in grid]

    # Mark the start and end points
    plot_grid[start[0]][start[1]] = colors['start']
    plot_grid[end[0]][end[1]] = colors['end']

    # Create a colormap based on the color names
    color_names = list(colors.values())
    cmap = ListedColormap([plt.get_cmap('tab10')(i) for i in range(len(color_names))])

    # Convert the color names to RGBA values
    plot_grid_rgba = [[plt.get_cmap('tab10')(color_names.index(color)) for color in row] for row in plot_grid]

    # Plot the grid using the colormap
    plt.imshow(plot_grid_rgba, cmap=cmap, origin='lower')

    # Plot the path as a line
    if path:
        xs, ys = zip(*path)
        plt.plot(ys, xs, color=colors['path'], linewidth=2)

    plt.grid(True)
```

```
    if state == 0:
        plt.title("Go path")
        plt.savefig('Q3_go_path.png')
    else:
        plt.title("Back path")
        plt.savefig('Q3_back_path.png')

    plt.show()


# ------------------------寻找最优关节角路径函数
    ------------------------
# 计算变换矩阵,传统D-H法
def standard_calc_transform_matrix(param):
    theta, d, a, alpha = param['theta'], param['d'], param['a'
    ], param['alpha']

    return np.array([[np.cos(theta), -np.sin(theta) * np.cos(
    alpha), np.sin(theta) * np.sin(alpha), a * np.cos(theta)],
                     [np.sin(theta), np.cos(theta) * np.cos(
    alpha), -np.cos(theta) * np.sin(alpha), a * np.sin(theta)],
                     [0, np.sin(alpha), np.cos(alpha), d],
                     [0, 0, 0, 1]])


# 目标函数：计算末端位置与目标位置之间的欧几里得距离和能耗
def objective_function(angles, target_position):
    T = np.eye(4)
    energy_consumption = gravitational_potential
    for i, param in enumerate(params[1:], start=1):
        param['theta'] = angles[i - 1]
        T = T @ standard_calc_transform_matrix(param)
        energy_consumption += 0.5 * param['I'] * param['omega'
    ] ** 2
    end_effector_position = T[:3, 3]
```

```
160     position_error = np.linalg.norm(end_effector_position -
    target_position)
161     if position_error > 200:
162         position_error = 200  # 误差阈值处理
163     # 动态调整能量权重
164     energy_weight = 0.01 + 0.001 * max(0, 200 - position_error
    ) / 200
165     total_error = position_error + energy_weight *
    energy_consumption
166     return float(total_error)
167
168
169 # 辅助函数：确保角度在给定的范围内
170 def clip_angle(theta, lower_limit, high_limit):
171     # 确保角度在给定的范围内
172     return max(lower_limit, min(high_limit, theta))
173
174
175 # 局部搜索：梯度下降
176 def local_search(individual, learning_rate=0.01, iterations
    =100):
177     for _ in range(iterations):
178         gradient = np.zeros_like(individual)
179         for i in range(len(individual)):
180             epsilon = 1e-16
181             individual_plus = individual.copy()
182             individual_plus[i] += epsilon
183             individual_minus = individual.copy()
184             individual_minus[i] -= epsilon
185             gradient[i] = (objective_function(individual_plus,
     target_position) -
186                            objective_function(individual_minus
    , target_position)) / (2 * epsilon)
187
188         # 更新个体
```

```
189         individual -= learning_rate * gradient
190         # 确保角度在有效范围内
191         for i in range(len(individual)):
192             individual[i] = clip_angle(individual[i], params[i
    ]['lower_limit'], params[i]['high_limit'])
193     return individual
194 
195 
196 # 多点交叉
197 def multipoint_crossover(parent1, parent2, num_points=2):
198     crossover_points = sorted(random.sample(range(1, len(
    parent1)), num_points))
199     child1, child2 = parent1.copy(), parent2.copy()
200 
201     for i in range(num_points):
202         start = crossover_points[i]
203         end = crossover_points[(i + 1) % num_points] if i <
    num_points - 1 else len(parent1)
204         child1[start:end], child2[start:end] = child2[start:
    end], child1[start:end]
205     return child1, child2
206 
207 
208 # 定义用于调整参数的函数
209 def adjust_parameters(crossover_rate, mutation_rate,
    best_fitness, prev_best_fitness):
210     improvement_threshold = 0.01  # 改善阈值
211     diversity_threshold = 0.01  # 多样性阈值
212 
213     # 检查是否有显著改善
214     if best_fitness > prev_best_fitness +
    improvement_threshold:
215         mutation_rate *= 0.9  # 减少变异率
216     else:
217         mutation_rate *= 1.1  # 增加变异率
```

```
218
219     # 检查种群多样性
220     if mutation_rate > 0.5:
221         crossover_rate *= 0.9  # 减少交叉率
222     else:
223         crossover_rate *= 1.1  # 增加交叉率
224
225     # 限制参数范围
226     crossover_rate = max(min(crossover_rate, 1.0), 0.0)
227     mutation_rate = max(min(mutation_rate, 1.0), 0.0)
228
229     return crossover_rate, mutation_rate
230
231
232 # 定义模拟退火算法
233 def adaptive_simulated_annealing(initial_solution,
    initial_temperature, cooling_factor, max_iterations,
234                                  objective_function,
    target_position, improvement_threshold=0.001,
235                                  diversity_threshold=0.01):
236     current_solution = initial_solution
237     current_fitness = objective_function(current_solution,
    target_position)
238     best_solution = current_solution
239     best_fitness = current_fitness
240
241     temperature = initial_temperature
242     prev_best_fitness = best_fitness
243
244     for iteration in range(max_iterations):
245         # Generate a new candidate solution
246         candidate_solution = current_solution + np.random.
    uniform(-0.01, 0.01, size=len(current_solution))
247         for i in range(len(candidate_solution)):
248             candidate_solution[i] = clip_angle(
```

```
    candidate_solution[i], params[i]['lower_limit'], params[i]['
    high_limit'])
249
250         candidate_fitness = objective_function(
    candidate_solution, target_position)
251
252         # Calculate the change in fitness
253         delta_fitness = candidate_fitness - current_fitness
254
255         # Acceptance probability
256         if delta_fitness < 0 or np.exp(-delta_fitness /
    temperature) > np.random.rand():
257             current_solution = candidate_solution
258             current_fitness = candidate_fitness
259
260             # Update the best solution
261             if current_fitness < best_fitness:
262                 best_solution = current_solution
263                 best_fitness = current_fitness
264                 prev_best_fitness = best_fitness
265
266         # Adaptive cooling
267         if best_fitness > prev_best_fitness +
    improvement_threshold:
268             cooling_factor *= 0.09  # Faster cooling
269         elif temperature > diversity_threshold:
270             cooling_factor *= 1.01  # Slower cooling
271
272         # Cool down the temperature
273         temperature *= cooling_factor
274
275     return best_solution, best_fitness
276
277
278 if __name__ == '__main__':
```

```
279     # ------------------------寻找最优底座移动路径
        ------------------------
280     df = pd.read_excel('附件.xlsx', sheet_name='Sheet1')
281     grid = []
282     start = (0, 0)
283     end = (0, 0)
284     for i in range(len(df)):
285         grid.append(df.iloc[i].tolist())
286
287     for i in range(len(grid)):
288         for j in range(len(grid[i])):
289             if grid[i][j] == "Start":
290                 start = (i, j)
291                 grid[i][j] = 0
292             if grid[i][j] == "End":
293                 end = (i, j)
294                 grid[i][j] = 0
295
296     path = a_star_search(grid, start, end)
297     print("Path:", path)
298     plot_path(grid, path, start, end, 0)
299
300     # 回到起点
301     start, end = end, start
302     path = a_star_search(grid, start, end)
303     print("BackPath:", path)
304     plot_path(grid, path, start, end, 1)
305
306     # ------------------------寻找最优关节角路径
        ------------------------
307     # 定义连杆参数
308     params = [
309         {'a': 0,   'alpha': 0,          'd': 600, 'theta': 0,
            'lower_limit': -np.pi*160/180, 'high_limit': np.pi
        *160/180, 'I': 0.5, 'omega': 2.0},  # 连杆1
```

```
        {'a': 300, 'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi / 2, 'lower_limit': -np.pi*150/180, 'high_limit': np.pi*15/180,  'I': 0.3, 'omega': 1.5},  # 连杆2
        {'a': 1200,'alpha': 0,          'd': 0,   'theta': 0,       'lower_limit': -np.pi*200/180, 'high_limit': np.pi*80/180,  'I': 0.4, 'omega': 1.0},  # 连杆3
        {'a': 300, 'alpha': -np.pi / 2, 'd': 1200,'theta': np.pi,      'lower_limit': -np.pi*180/180, 'high_limit': np.pi*180/180, 'I': 0.6, 'omega': 2.5},  # 连杆4
        {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': -np.pi / 2, 'lower_limit': -np.pi*120/180, 'high_limit': np.pi*120/180, 'I': 0.2, 'omega': 3.0},  # 连杆5
        {'a': 0,   'alpha': -np.pi / 2, 'd': 0,   'theta': 0,       'lower_limit': -np.pi*180/180, 'high_limit': np.pi*180/180, 'I': 0.4, 'omega': 2.0}   # 连杆6
    ]
    target_position = np.array([1500, 1200, 200])
    num_joints = len(params)

    # 初始化变换矩阵
    T = np.eye(4)
    T_i=np.eye(4) @ standard_calc_transform_matrix({'a': 0, 'alpha': 0, 'd': 0, 'theta': 0  })
    for param in params:
        T_i = T_i @ standard_calc_transform_matrix(param)
    end_position = T_i[:3, 3]
    change_h=(end_position[2]-target_position[2])/1000
    M = 5
    g = 10
    gravitational_potential= M * g * change_h

    # 自适应遗传算法
    # 遗传算法参数
    population_size = 100
    num_generations = 50
```

```
    crossover_rate = 0.8
    mutation_rate = 0.1
    elite_count = 2  # 精英个体的数量
    num_crossover_points = 2  # 多点交叉的点数

    # 初始化种群
    population = [
        [clip_angle(np.random.uniform(params[i]['lower_limit'], params[i]['high_limit']),
                    params[i]['lower_limit'], params[i]['high_limit'])
         for i in range(num_joints)]
        for _ in range(population_size)
    ]

    # 初始化变量用于记录最佳解
    best_solution = None
    best_fitness = float('-inf')
    best_solution_generation = 0

    # 遗传算法主循环
    prev_best_fitness = 0.0
    for generation in range(num_generations):
        # 评估适应度
        fitness = [1.0 / (1.0 + objective_function(individual, target_position)) for individual in population]

        # 排序种群
        sorted_indices = np.argsort(fitness)[::-1]
        sorted_population = [population[i] for i in sorted_indices]
        sorted_fitness = [fitness[i] for i in sorted_indices]

        # 更新最佳解
        if sorted_fitness[0] > best_fitness:
```

```
            best_fitness = sorted_fitness[0]
            best_solution = sorted_population[0]
            best_solution_generation = generation

        # 精英保留
        elite_population = sorted_population[:elite_count]
        elite_fitness = sorted_fitness[:elite_count]

        # 选择操作：轮盘赌选择，去除精英个体
        probabilities = [f / sum(sorted_fitness[elite_count:]) for f in sorted_fitness[elite_count:]]

        # 使用 random.choices 进行选择
        selected_population = random.choices(sorted_population[elite_count:],
                                             weights=probabilities,
                                             k=population_size - elite_count)

        # 交叉操作
        offspring = []
        while len(offspring) < population_size - elite_count:
            if random.random() < crossover_rate:
                parent1, parent2 = random.choices(selected_population, weights=probabilities, k=2)
                child1, child2 = multipoint_crossover(parent1, parent2, num_crossover_points)
                offspring.extend([child1, child2])

        # 变异操作
        for individual in offspring:
            for j in range(num_joints):
                if random.random() < mutation_rate:
                    # 变异
```

```
                    individual[j] += random.uniform(-0.01, 0.01)
                    # 确保角度在有效范围内
                    individual[j] = clip_angle(individual[j], params[j]['lower_limit'], params[j]['high_limit'])

        # 局部搜索
        offspring = [local_search(individual) for individual in offspring]

        # 更新种群
        population = elite_population + offspring[:population_size - elite_count]

        # 输出当前最佳解
        print(
            f"Generation {generation}: Best Fitness {sorted_fitness[0]}, Best Angles {sorted_population[0]}")

        # 调整参数
        crossover_rate, mutation_rate = adjust_parameters(crossover_rate, mutation_rate, sorted_fitness[0],
                                                          prev_best_fitness)
        prev_best_fitness = sorted_fitness[0]

    best_solution_degrees = [angle * 180 / np.pi for angle in best_solution]

    # 输出最佳解
    print("Best Solution Found:")
    print("Generation:", best_solution_generation)
    print("Joint Angles:", best_solution_degrees)
    print("Fitness:", best_fitness)
    print("End-Effector Error:", objective_function(
```

```
    best_solution, target_position))

    # Adaptive Simulated Annealing Parameters
    initial_temperature = 100
    cooling_factor = 0.995
    max_iterations = 10000
    # Apply Adaptive Simulated Annealing to the best solution found by the GA
    best_solution, best_fitness = adaptive_simulated_annealing(best_solution, initial_temperature, cooling_factor,

     max_iterations, objective_function, target_position)
    best_solution_degrees = [angle * 180 / np.pi for angle in best_solution]

    # Output the refined best solution
    print("Refined Best Solution Found:")
    print("Joint Angles:", best_solution_degrees)
    print("Fitness:", best_fitness)
    print("End-Effector Error:", objective_function(best_solution, target_position))
```

Q4.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random
from heapq import heappop, heappush


#-----------------------tsp函数-----------------------
# 计算两点之间的曼哈顿距离
def manhattan_distance(p1, p2):
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

```

```
13
14  # A* 算法
15  def a_star(maze, start, goal):
16      n = len(maze)
17      open_set = [(0, start)]
18      came_from = {}
19      g_score = {tuple(start): 0}
20      f_score = {tuple(start): manhattan_distance(start, goal)}
21
22      while open_set:
23          _, current = heappop(open_set)
24
25          if tuple(current) == tuple(goal):
26              path = [goal]
27              while tuple(path[-1]) != tuple(start):
28                  path.append(came_from[tuple(path[-1])])
29              return list(reversed(path))
30
31          for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
32              neighbor = (current[0] + dx, current[1] + dy)
33              if 0 <= neighbor[0] < n and 0 <= neighbor[1] < n
    and maze[neighbor[0]][neighbor[1]] == 0:
34                  tentative_g_score = g_score[tuple(current)] +
    1
35                  if tuple(neighbor) not in g_score or
    tentative_g_score < g_score[tuple(neighbor)]:
36                      came_from[tuple(neighbor)] = current
37                      g_score[tuple(neighbor)] =
    tentative_g_score
38                      f_score[tuple(neighbor)] =
    tentative_g_score + manhattan_distance(neighbor, goal)
39                      heappush(open_set, (f_score[tuple(neighbor
    )], neighbor))
40
41      return None
```

```
42
43
44  # 计算路径总长度
45  def path_length(path, cities):
46      total_distance = 0
47      for i in range(len(path) - 1):
48          total_distance += manhattan_distance(cities[path[i]],
    cities[path[i + 1]])
49      total_distance += manhattan_distance(cities[path[-1]],
    cities[path[0]])  # 返回起点
50      return total_distance
51
52
53  # 模拟退火算法
54  def tsp_simulated_annealing(cities, maze, initial_temperature
    =1000, cooling_rate=0.999999, iterations=1000000):
55      n = len(cities)
56      # 从第一个城市开始
57      current_path = np.arange(n)
58      # 不需要打乱路径，因为我们要从第一个城市开始
59      best_path = current_path.copy()
60      best_length = path_length(best_path, cities)
61
62      temperature = initial_temperature
63
64      for _ in range(iterations):
65          new_path = current_path.copy()
66          # 随机选择两个位置进行交换
67          i, j = np.random.choice(n, 2, replace=False)
68          new_path[i], new_path[j] = new_path[j], new_path[i]
69
70          current_length = path_length(current_path, cities)
71          new_length = path_length(new_path, cities)
72
73          if new_length < current_length or np.exp((
```

```
    current_length - new_length) / temperature) > np.random.rand
    ():
74              current_path = new_path
75              if new_length < best_length:
76                  best_path = new_path.copy()
77                  best_length = new_length
78
79          temperature *= cooling_rate
80
81      # 调整路径顺序，确保从索引为 0 的城市开始
82      zero_index = np.where(best_path == 0)[0][0]
83      best_path = np.concatenate((best_path[zero_index:],
    best_path[:zero_index]))
84
85      # 将城市坐标转换为迷宫中的路径
86      path = [cities[best_path[0]]]
87      for i in range(1, len(best_path)):
88          path.extend(a_star(maze, path[-1], cities[best_path[i
    ]])[1:])
89      path.extend(a_star(maze, path[-1], cities[best_path[0]])
    [1:])  # 返回起点
90
91      return best_path, path, best_length
92
93
94  # ------------------------寻找最优关节角路径函数
    ------------------------
95  # 计算变换矩阵,传统D-H法
96  def standard_calc_transform_matrix(param):
97      theta, d, a, alpha = param['theta'], param['d'], param['a'
    ], param['alpha']
98
99      return np.array([[np.cos(theta), -np.sin(theta) * np.cos(
    alpha), np.sin(theta) * np.sin(alpha), a * np.cos(theta)],
100                     [np.sin(theta), np.cos(theta) * np.cos(
```

```
    alpha), -np.cos(theta) * np.sin(alpha), a * np.sin(theta)],
                    [0, np.sin(alpha), np.cos(alpha), d],
                    [0, 0, 0, 1]])


# 目标函数：计算末端位置与目标位置之间的欧几里得距离和能耗
def objective_function(angles, target_position):
    T = np.eye(4)
    energy_consumption = gravitational_potential
    for i, param in enumerate(params[1:], start=1):
        param['theta'] = angles[i - 1]
        T = T @ standard_calc_transform_matrix(param)
        energy_consumption += 0.5 * param['I'] * param['omega'] ** 2
    end_effector_position = T[:3, 3]
    position_error = np.linalg.norm(end_effector_position - target_position)
    if position_error > 200:
        position_error = 200  # 误差阈值处理
    # 动态调整能量权重
    energy_weight = 0.01 + 0.001 * max(0, 200 - position_error) / 200
    total_error = position_error + energy_weight * energy_consumption
    return float(total_error)


# 辅助函数：确保角度在给定的范围内
def clip_angle(theta, lower_limit, high_limit):
    # 确保角度在给定的范围内
    return max(lower_limit, min(high_limit, theta))


# 局部搜索：梯度下降
def local_search(individual, learning_rate=0.01, iterations
```

```
    =100):
131     for _ in range(iterations):
132         gradient = np.zeros_like(individual)
133         for i in range(len(individual)):
134             epsilon = 1e-16
135             individual_plus = individual.copy()
136             individual_plus[i] += epsilon
137             individual_minus = individual.copy()
138             individual_minus[i] -= epsilon
139             gradient[i] = (objective_function(individual_plus,
     target_position) -
140                           objective_function(individual_minus
    , target_position)) / (2 * epsilon)
141 
142         # 更新个体
143         individual -= learning_rate * gradient
144         # 确保角度在有效范围内
145         for i in range(len(individual)):
146             individual[i] = clip_angle(individual[i], params[i
    ]['lower_limit'], params[i]['high_limit'])
147     return individual
148 
149 
150 # 多点交叉
151 def multipoint_crossover(parent1, parent2, num_points=2):
152     crossover_points = sorted(random.sample(range(1, len(
    parent1)), num_points))
153     child1, child2 = parent1.copy(), parent2.copy()
154 
155     for i in range(num_points):
156         start = crossover_points[i]
157         end = crossover_points[(i + 1) % num_points] if i <
    num_points - 1 else len(parent1)
158         child1[start:end], child2[start:end] = child2[start:
    end], child1[start:end]
```

```
159     return child1, child2
160
161
162 # 定义用于调整参数的函数
163 def adjust_parameters(crossover_rate, mutation_rate,
    best_fitness, prev_best_fitness):
164     improvement_threshold = 0.01  # 改善阈值
165     diversity_threshold = 0.01  # 多样性阈值
166
167     # 检查是否有显著改善
168     if best_fitness > prev_best_fitness +
    improvement_threshold:
169         mutation_rate *= 0.9  # 减少变异率
170     else:
171         mutation_rate *= 1.1  # 增加变异率
172
173     # 检查种群多样性
174     if mutation_rate > 0.5:
175         crossover_rate *= 0.9  # 减少交叉率
176     else:
177         crossover_rate *= 1.1  # 增加交叉率
178
179     # 限制参数范围
180     crossover_rate = max(min(crossover_rate, 1.0), 0.0)
181     mutation_rate = max(min(mutation_rate, 1.0), 0.0)
182
183     return crossover_rate, mutation_rate
184
185
186 # 定义模拟退火算法
187 def adaptive_simulated_annealing(initial_solution,
    initial_temperature, cooling_factor, max_iterations,
188                                  objective_function,
    target_position, improvement_threshold=0.001,
189                                  diversity_threshold=0.01):
```

```
    current_solution = initial_solution
    current_fitness = objective_function(current_solution, target_position)
    best_solution = current_solution
    best_fitness = current_fitness

    temperature = initial_temperature
    prev_best_fitness = best_fitness

    for iteration in range(max_iterations):
        # Generate a new candidate solution
        candidate_solution = current_solution + np.random.uniform(-0.01, 0.01, size=len(current_solution))
        for i in range(len(candidate_solution)):
            candidate_solution[i] = clip_angle(candidate_solution[i], params[i]['lower_limit'], params[i]['high_limit'])

        candidate_fitness = objective_function(candidate_solution, target_position)

        # Calculate the change in fitness
        delta_fitness = candidate_fitness - current_fitness

        # Acceptance probability
        if delta_fitness < 0 or np.exp(-delta_fitness / temperature) > np.random.rand():
            current_solution = candidate_solution
            current_fitness = candidate_fitness

            # Update the best solution
            if current_fitness < best_fitness:
                best_solution = current_solution
                best_fitness = current_fitness
                prev_best_fitness = best_fitness
```

```
219
220         # Adaptive cooling
221         if best_fitness > prev_best_fitness +
    improvement_threshold:
222             cooling_factor *= 0.09  # Faster cooling
223         elif temperature > diversity_threshold:
224             cooling_factor *= 1.01  # Slower cooling
225
226         # Cool down the temperature
227         temperature *= cooling_factor
228
229     return best_solution, best_fitness
230
231
232 if __name__ == '__main__':
233     # ------------------------寻找最优底座移动路径
    ------------------------
234     df = pd.read_excel('附件.xlsx', sheet_name='Sheet2')
235     grid = []
236     start = (0, 0)
237     target= {"Start" : (0, 0)}
238     cities=[start]
239     for i in range(len(df)):
240         grid.append(df.iloc[i].tolist())
241
242     for i in range(len(grid)):
243         for j in range(len(grid[i])):
244             if grid[i][j] == "Start":
245                 start = (i, j)
246                 grid[i][j] = 0
247             if type(grid[i][j]) == str and grid[i][j][:6] == "
    target":
248                 target[grid[i][j]]=(i,j)
249                 cities.append((i,j))
250                 grid[i][j] = 0
```

```
    grid=np.array(grid)
    reversed_dict = {value: key for key, value in target.items()}

    # 使用模拟退火算法求解
    best_path_indices, best_path, best_length = tsp_simulated_annealing(cities, grid)
    print("最佳路径长度:", best_length)
    print("最佳路径顺序 (索引):", best_path_indices)
    for i in range(len(best_path_indices)):
        print(reversed_dict[cities[best_path_indices[i]]], end=' -> ')
    print(reversed_dict[cities[best_path_indices[0]]])

    # 可视化结果
    plt.figure(figsize=(10, 6))
    plt.imshow(grid, cmap='gray', origin='lower')

    # 标记起始点
    start_point = cities[0]
    plt.plot(start_point[1], start_point[0], 'go', markersize=18, label='Start Point')

    # 标记每个城市的编号
    for i, city in enumerate(cities):
        plt.text(city[1], city[0], str(i), color='black', fontsize=10, ha='center', va='center')
        plt.plot(city[1], city[0], 'go', markersize=10)

    for i in range(len(best_path)):
        plt.plot(best_path[i][1], best_path[i][0], 'ro')
        if i < len(best_path) - 1:
            plt.plot([best_path[i][1], best_path[i + 1][1]], [best_path[i][0], best_path[i + 1][0]], 'b-')

```

```
280     plt.legend()
281     plt.title('Final Path')
282     plt.savefig('Q4_path.png')
283     plt.show()
284 
285     # ------------------------寻找最优关节角路径
    ------------------------
286     # 定义连杆参数
287     params = [
288         {'a': 0, 'alpha': 0, 'd': 600, 'theta': 0, '
    lower_limit': -np.pi * 160 / 180, 'high_limit': np.pi * 160
    / 180,
289          'I': 0.5, 'omega': 2.0},  # 连杆1
290         {'a': 300, 'alpha': -np.pi / 2, 'd': 0, 'theta': -np.
    pi / 2, 'lower_limit': -np.pi * 150 / 180,
291          'high_limit': np.pi * 15 / 180, 'I': 0.3, 'omega':
    1.5},  # 连杆2
292         {'a': 1200, 'alpha': 0, 'd': 0, 'theta': 0, '
    lower_limit': -np.pi * 200 / 180, 'high_limit': np.pi * 80 /
     180,
293          'I': 0.4, 'omega': 1.0},  # 连杆3
294         {'a': 300, 'alpha': -np.pi / 2, 'd': 1200, 'theta': np
    .pi, 'lower_limit': -np.pi * 180 / 180,
295          'high_limit': np.pi * 180 / 180, 'I': 0.6, 'omega':
    2.5},  # 连杆4
296         {'a': 0, 'alpha': -np.pi / 2, 'd': 0, 'theta': -np.pi
    / 2, 'lower_limit': -np.pi * 120 / 180,
297          'high_limit': np.pi * 120 / 180, 'I': 0.2, 'omega':
    3.0},  # 连杆5
298         {'a': 0, 'alpha': -np.pi / 2, 'd': 0, 'theta': 0, '
    lower_limit': -np.pi * 180 / 180,
299          'high_limit': np.pi * 180 / 180, 'I': 0.4, 'omega':
    2.0}  # 连杆6
300     ]
301     target_position = np.array([1500, 1200, 200])
```

```
    num_joints = len(params)

    # 初始化变换矩阵
    T = np.eye(4)
    T_i = np.eye(4) @ standard_calc_transform_matrix({'a': 0, 'alpha': 0, 'd': 0, 'theta': 0})
    for param in params:
        T_i = T_i @ standard_calc_transform_matrix(param)
    end_position = T_i[:3, 3]
    change_h = (end_position[2] - target_position[2]) / 1000
    M = 5
    g = 10
    gravitational_potential = M * g * change_h

    # 自适应遗传算法
    # 遗传算法参数
    population_size = 100
    num_generations = 50
    crossover_rate = 0.8
    mutation_rate = 0.1
    elite_count = 2  # 精英个体的数量
    num_crossover_points = 2  # 多点交叉的点数

    # 初始化种群
    population = [
        [clip_angle(np.random.uniform(params[i]['lower_limit'], params[i]['high_limit']),
                    params[i]['lower_limit'], params[i]['high_limit'])
         for i in range(num_joints)]
        for _ in range(population_size)
    ]

    # 初始化变量用于记录最佳解
    best_solution = None
```

```
    best_fitness = float('-inf')
    best_solution_generation = 0

    # 遗传算法主循环
    prev_best_fitness = 0.0
    for generation in range(num_generations):
        # 评估适应度
        fitness = [1.0 / (1.0 + objective_function(individual, target_position)) for individual in population]

        # 排序种群
        sorted_indices = np.argsort(fitness)[::-1]
        sorted_population = [population[i] for i in sorted_indices]
        sorted_fitness = [fitness[i] for i in sorted_indices]

        # 更新最佳解
        if sorted_fitness[0] > best_fitness:
            best_fitness = sorted_fitness[0]
            best_solution = sorted_population[0]
            best_solution_generation = generation

        # 精英保留
        elite_population = sorted_population[:elite_count]
        elite_fitness = sorted_fitness[:elite_count]

        # 选择操作：轮盘赌选择，去除精英个体
        probabilities = [f / sum(sorted_fitness[elite_count:]) for f in sorted_fitness[elite_count:]]

        # 使用 random.choices 进行选择
        selected_population = random.choices(sorted_population[elite_count:],
                                             weights=probabilities,
```

```
                                                k=population_size - elite_count)

        # 交叉操作
        offspring = []
        while len(offspring) < population_size - elite_count:
            if random.random() < crossover_rate:
                parent1, parent2 = random.choices(selected_population, weights=probabilities, k=2)
                child1, child2 = multipoint_crossover(parent1, parent2, num_crossover_points)
                offspring.extend([child1, child2])

        # 变异操作
        for individual in offspring:
            for j in range(num_joints):
                if random.random() < mutation_rate:
                    # 变异
                    individual[j] += random.uniform(-0.01, 0.01)
                    # 确保角度在有效范围内
                    individual[j] = clip_angle(individual[j], params[j]['lower_limit'], params[j]['high_limit'])

        # 局部搜索
        offspring = [local_search(individual) for individual in offspring]

        # 更新种群
        population = elite_population + offspring[:population_size - elite_count]

        # 输出当前最佳解
        print(
            f"Generation {generation}: Best Fitness {
```

```
    sorted_fitness[0]}, Best Angles {sorted_population[0]}")
392
393         # 调整参数
394         crossover_rate, mutation_rate = adjust_parameters(
    crossover_rate, mutation_rate, sorted_fitness[0],
395
    prev_best_fitness)
396         prev_best_fitness = sorted_fitness[0]
397
398     best_solution_degrees = [angle * 180 / np.pi for angle in
    best_solution]
399     # 输出最佳解
400     print("Best Solution Found:")
401     print("Generation:", best_solution_generation)
402     print("Joint Angles:", best_solution_degrees)
403     print("Fitness:", best_fitness)
404     print("End-Effector Error:", objective_function(
    best_solution, target_position))
405
406     # Adaptive Simulated Annealing Parameters
407     initial_temperature = 100
408     cooling_factor = 0.995
409     max_iterations = 10000
410
411     # Apply Adaptive Simulated Annealing to the best solution
    found by the GA
412     best_solution, best_fitness = adaptive_simulated_annealing
    (best_solution, initial_temperature, cooling_factor,
413
     max_iterations, objective_function, target_position)
414     best_solution_degrees = [angle * 180 / np.pi for angle in
    best_solution]
415
416     # Output the refined best solution
417     print("Refined Best Solution Found:")
```

```
418     print("Joint Angles:", best_solution_degrees)
419     print("Fitness:", best_fitness)
420     print("End-Effector Error:", objective_function(
    best_solution, target_position))
```