舞龙队前行模型

摘要

板凳龙是浙闽地区极具地域和文化特色的非物质文化遗产，由几十条至上百条板凳首尾相连形成蜿蜒的龙形。在舞龙队能够自如盘入盘出的情况下，其所需面积越小、行进速度越快，观赏效果越佳。本文建立数学模型，运用各类几何方法、向量分析、构造碰撞检测算法等多种方法研究盘龙时龙身的位置速度信息、碰撞检测以及调头问题。

针对问题一，我们通过求解含**极坐标下曲线积分**的方程组得到任意时刻龙头把手的位置坐标，并根据相邻把手位置间的固定距离及其所在的曲线方程，依次**迭代求解**得到全部龙头把手的位置信息。然后推导相邻把手间速度夹角的关系并依据把手**沿刚体方向的速度分量相等**的原理，依次迭代求解得到全部龙头把手的速度信息。

针对问题二，我们根据相邻把手的位置信息确定任意时刻每段龙身所处的矩形区域，然后构造**碰撞检测算法**检测任意时刻舞龙队的碰撞情况，并对碰撞算法进行改进以**降低时间复杂度**，最终确定了舞龙队盘入的终止时刻 $t = 412.47s$。

针对问题三，我们通过计算确定任意螺距的等距螺线与调头区域边界的**交点**，然后结合第二问的碰撞检测算法，约束舞龙队在到达交点前不能发生碰撞，最终通过**求解约束条件下的最值问题**得到最小螺距 $H^*_{min} = 0.37m$。

针对问题四，我们定义了**调头路径模型**，并采用**定一动一**的思想对调头路径的约束条件进行了**几何分析**，通过补全圆弧和分割等距螺线，求得了满足约束条件下 S 形曲线的**全部解空间**。在确定调头路径之后，我们在问题一的基础上进一步扩展得到了含调头路径和盘出螺线的**位置和速度求解模型**。最终求得最优调头曲线最短弧长 $|S|_{min} = 13.32m$。

针对问题五，我们在问题四得到的最优调头曲线的基础上，通过**速度求解模型**得到任意龙头把手下的全部把手速度信息，并在满足舞龙队把手速度峰值约束的前提下，根据**梯度下降**求得最大龙头把手速度 $v_{max} = 1.89m/s$。

关键字： 等距螺线　曲线积分　刚体　碰撞算法　定一动一　迭代求解　梯度下降

一、 问题重述

板凳龙是浙闽地区的一种传统地方民俗文化活动 [1]，在元宵节等节日中广泛举办。它通过首尾相连的板凳连接成蜿蜒曲折的龙形，长度灵活、队形多变。通常情况下，舞龙队伍的行进速度和盘龙时所需面积是影响其观赏效果的关键因素。为了增加其观赏性、避免发生碰撞，我们需要建立数学模型解决以下问题：

问题一：已知舞龙队伍沿螺距为 55 cm 的等距螺线顺时针盘入，龙头的行进速度恒为 1 m/s，计算从 0 秒到 300 秒之间，每秒龙头、龙身及龙尾的位置和速度，并提供特定时刻的详细数据。

问题二：在舞龙队沿问题一螺线盘入的过程中，确定板凳之间不发生碰撞的终止时刻，并给出此时舞龙队各部分的位置和速度。

问题三：为使龙头顺利进入直径为 9 米的圆形调头空间，确定最小螺距使得龙头能够沿相应螺线盘入至调头空间的边界。

问题四：舞龙队通过顺时针盘入到逆时针盘出的过程中，调头路径由两段半径比例为 2：1 且相切的圆弧组成，需要优化调头曲线，使其长度最短，并给出调头过程中各部分的位置和速度变化。

问题五：舞龙队沿问题四确定的路径移动，要求各把手行进速度不超过 2 m/s，建立数学模型计算龙头的最大行进速度，使得队伍各部分的速度均满足该限制条件。

二、 问题分析

2.1 问题一的分析

问题一要求我们求解舞龙队沿给定螺距下的等距螺线盘入后指定时刻的位置与速度。首先我们通过求**极坐标下的弧长积分公式**得到每一时刻龙头把手的位置坐标，然后根据相邻把手**两点间距离公式**依次迭代求得全部把手的位置，接着推导相邻把手间的**速度夹角关系**并依据**沿刚体方向速度分量相同**的原理依次迭代求得各个龙身把手的速度。

2.2 问题二的分析

在问题二中，舞龙队仍沿问题一中的螺线盘入，要求找到发生碰撞时的临界情形与对应的临界状态。首先我们根据每个时刻把手的位置信息确定各板凳所占的**矩形区域**，然后构造**碰撞检测算法**检测矩形区域的重合情况并对其加以改进来**降低算法的时间复杂度**，最后将**时间离散化**检测每一时刻的碰撞情况。

2.3 问题三的分析

问题三要求确定使得龙头把手能沿螺线进入掉头空间边界的最小螺距。首先我们计算得到任一螺距的螺线与掉头区域边界的**交点**，然后约束舞龙队在到达交点前不能发生碰撞并利用问题二的碰撞检测算法进行检测，最后通过**求解约束条件下的最值问题**得到最小螺距。

2.4 问题四的分析

问题四需要寻找由两个相切圆弧构成、前一段圆弧半径为后一段的二倍且与盘入盘出曲线相切的曲线，并调整圆弧使其长度最短，以及计算各个时刻舞龙队的位置和速度信息。我们首先构造由**半径、角度**决定的目标函数，通过**几何关系**分析其自由度并采用**定一动一**的思想得到曲线的全部解空间，然后采用和问题一类似的思想确定各把手相关信息。

2.5 问题五的分析

问题五要求确定舞龙队在沿 S 形调头曲线行进时，龙头的最大行进速度，使得所有把手的速度均不超过 2 m/s。由问题一的分析可知，龙头速度的变化会影响每个把手的速度，因此需要综合考虑**曲率和速度的传递关系**。通过建立**运动学模型**，可以计算每个把手的速度，并逐步调整龙头的速度，确保所有把手的速度都不超过 2 m/s，最终确定龙头的最大行进速度。

三、 模型假设

1. 假设龙头把手在运动过程中视为质点
2. 假设所有板凳处于同一水平面上
3. 假设板凳为刚体
4. 认为板凳的厚度可以忽略不计
5. 假设相邻两节龙身之间可以以任意角度旋转

四、 符号说明

符号	含义
$P_i(t)$	第 i 个把手在 t 时刻的坐标
v_i	第 i 个把手的速度
L	螺线长度
R_i	第 i 个板凳所占矩形区域
Φ	调头区域
$f(t)$	碰撞检测算法
α_s	s 形曲线前一段的弧度
O_1	s 形曲线前一段的圆心
O_2	s 形曲线后一段的圆心
β_s	s 形曲线后一段的弧度
H	等距螺线的螺距
$\|S\|$	s 形曲线弧长
$Location(P_i)$	第 i 个把手所处的曲线类型

五、 模型的建立与求解

5.1 问题一模型的建立与求解

问题一要求我们求解舞龙队沿等距螺线盘入后指定时刻的位置与速度。首先我们通过求极坐标下的弧长积分公式得到每一时刻龙头把手的位置坐标，然后根据相邻把手的距离信息迭代求得全部把手的位置，接着几何推导夹角关系并依据沿刚体方向速度分量相同的原理依次迭代求得各个龙身把手的速度。

5.1.1 舞龙队盘入位置模型

（1）等距螺线方程

本题给出的等距螺线方程 [2] 在极坐标系中的数学表达式为：

$$r = b_0\theta$$

其中 r 是从极点到某个把手的位置的径向距离,θ 是从原点到该点逆时针旋转的角度。参数 $b_0 = \frac{0.55}{2\pi}(m/rad)$ 表示每旋转一圈时，径向距离 r 增加 0.55 米。初始时刻舞龙队龙头把手的位置坐标为 $(8.8, 32\pi)$，顺时针方向沿等距螺线轨迹以 1m/s 的速度开始运动。

（2）各把手位置的极坐标表示

舞龙队的每个把手的位置可以通过极坐标来表示。对于第 i 个把手，其位置坐标是关于 t 的函数，在极坐标系中表示为：$P_i(t) = (r_i, \theta_i)$。其中，r_i 是该把手的径向距离，θ_i 是该把手相对于极点的极角。整个舞龙队由 224 个把手组成，所有把手位置信息形成一个位置集合：

$$P = \{P_1, P_2 \dots P_{223}, P_{224}\}$$

其中 P_1 为龙头把手中心的位置，P_2 为第一节龙身前把手中心的位置，P_{224} 为龙尾后把手中心的位置。

（3）龙头把手的位置求解

舞龙队盘入等距螺线，经过 t 秒后龙头把手中心的位置坐标 $P_1(t) = (r_1, \theta_1)$ 可通过求解等距螺线未经过的弧线长度的积分式得到，即 $P_1(t) = (r_1, \theta_1)$ 满足如下方程组：

$$\begin{cases} L_{sum} - L_1(t) = b_0 \int_0^{\theta_1} \sqrt{1+\theta^2}\, d\theta \\ r_1 = b_0\theta_1 \end{cases} \tag{1}$$

其中 L_{sum} 为螺旋线总长度，由弧长积分得到：

$$L_{sum} = b_0 \int_0^{32\pi} \sqrt{1+\theta^2}\, d\theta$$

$L_1(t)$ 为经过 t 秒后龙头把手沿等距螺线走过的长度，其满足：

$$L_1(t) = t$$

（4）后续把手位置的迭代求解

由于相邻两个把手之间的板长长度固定，可通过相邻把手间的距离来依次迭代计算后续把手的位置坐标。相邻把手间位置坐标的距离 $|P_iP_{i+1}|$ 可由两点之间的距离公式给出：

$$|P_iP_{i+1}| = \sqrt{(r_i cos\theta_i - r_{i+1}cos\theta_{i+1})^2 + (r_i sin\theta_i - r_{i+1}sin\theta_{i+1})^2}$$

由于龙头和后续龙身的板长不同，相邻把手间的距离也分为不同情况，其具体满足：

$$\begin{cases} |P_iP_{i+1}| = 2.86 \quad if \;\; i = 1 \\ |P_iP_{i+1}| = 1.65 \quad if \;\; i \geq 2 \\ r_i = b_0\theta_i \\ \theta_{i+1} - \theta_i < 2\pi \end{cases}$$

其中龙头位置坐标 P_1 由前文给出，后续把手的位置坐标可通过求解上述方程组唯一得到，即可迭代计算出经过任意 t 秒后的全部 224 个把手的位置信息。

5.1.2 舞龙队盘入速度模型

在舞龙队沿等距螺线行进的过程中，对于相邻的两个把手，其速度关系如图1所示：

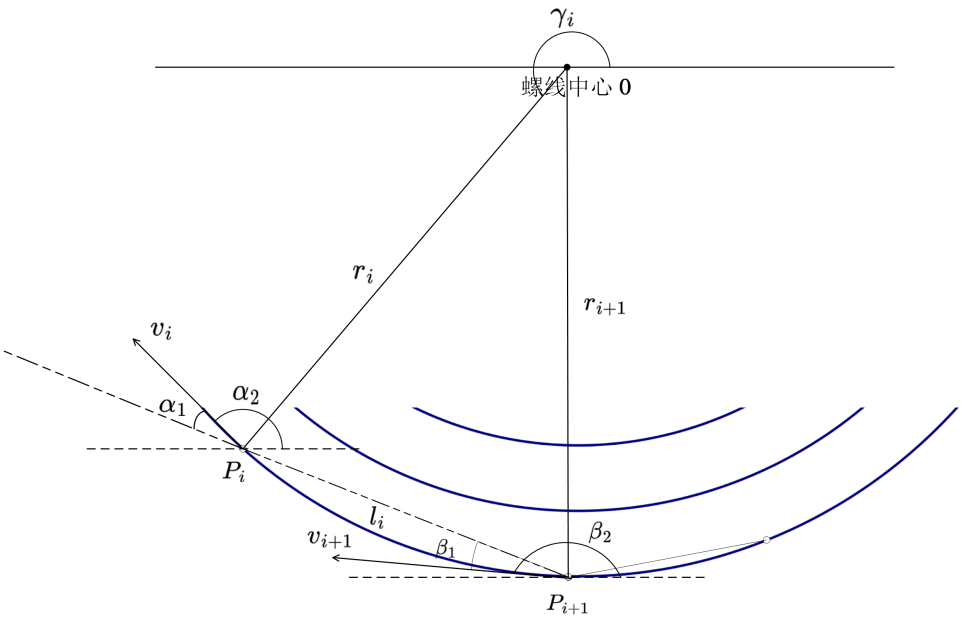


图 **1**　速度关系图

对于第 i 个龙头把手 P_i，其速度 v_i 方向为沿当前位置的等距螺线切线的顺时针方向。由于舞龙龙身为刚体 [3]，相邻龙头把手的速度沿龙身方向的速度分量相同。为求得每个龙头把手当前时刻沿龙身方向的速度分量，需求得相邻龙头把手的速度方向与龙身方向的夹角关系。对于第 i 个龙头把手 P_i，其速度与刚体方向的夹角 α_1 在几何关系上满足：

$$\alpha_1 = \gamma_i - \alpha_2 - \angle OP_iP_{i+1} + k\pi \tag{2}$$

其中 γ_i 由当前位置的极角 θ_i 给出：

$$\gamma_i \equiv \theta_i \pmod{2\pi}$$

α_2 为当前速度方向对应的倾斜角，可通过微分计算该点处的导数来求解：

$$\alpha_2 = arctan(\frac{sin\theta_i + \theta_i cos\theta_i}{cos\theta_i - \theta_i sin\theta_i})$$

对于 P_i, P_{i+1} 之间的距离 l_i，其满足条件在前文中已经给出，P_i, P_{i+1} 对应的极径在前文中位置信息的求解过程中也已经给出。则在 $\triangle OP_iP_{i+1}$ 中可通过余弦定理求解 $\angle OP_iP_{i+1}$：

$$\angle OP_iP_{i+1} = \frac{r_i^2 + l_i^2 - r_{i+1}^2}{2r_il_i}$$

至此求得 α_1。同理可求解第 $i+1$ 个龙头把手 P_{i+1} 速度沿刚体方向的夹角 β_1：

$$\beta_1 = \gamma_{i+1} - \beta_2 - \angle P_iP_{i+1}O + k\pi$$

相邻龙头把手的速度满足沿刚体方向的速度分量大小相同，即：

$$|v_i cos\alpha_1| = |v_{i+1} cos\beta_1| \tag{3}$$

当 $i = 1$ 时，$v_1 = 1m/s$，即可迭代求解全部把手的速度信息。

5.1.3 模型求解与灵敏度分析

根据上述建立的模型，将题目中给定的参数代入并利用 Python 编写程序求解，即可得到不同时刻舞龙队的位置和速度，所得结果如表1、表2所示。

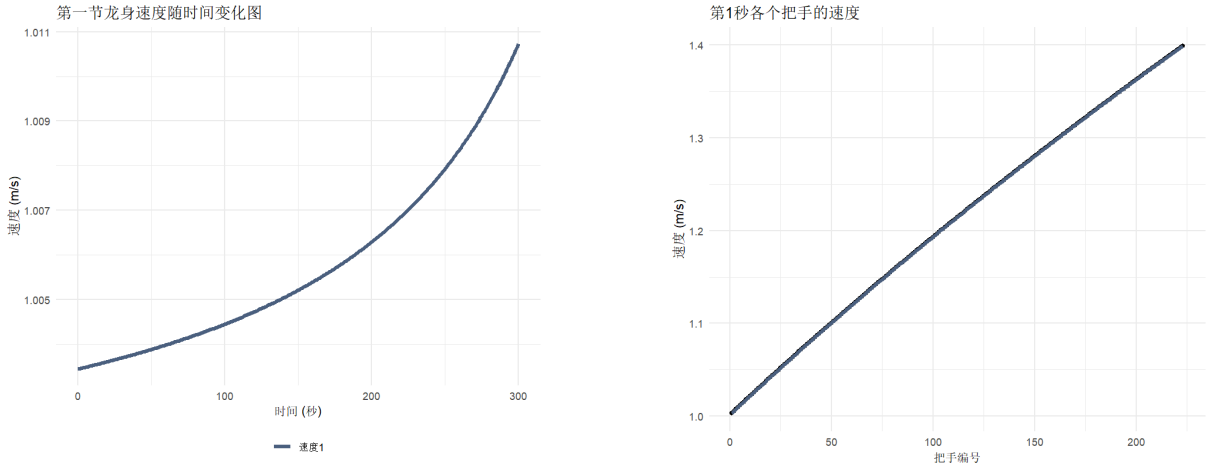
表 **1**　问题一位置结果

	0 s	60 s	120 s	180 s	240 s	300 s
龙头 x (m)	8.800000	5.799209	-4.084887	-2.963609	2.594494	4.420274
龙头 y (m)	0.000000	-5.771092	-6.304479	6.094780	-5.356743	2.320429
第 1 节龙身 x (m)	8.363824	7.456758	-1.445473	-5.237118	4.821221	2.459489
第 1 节龙身 y (m)	2.826544	-3.440399	-7.405883	4.359627	-3.561949	4.402476
第 51 节龙身 x (m)	-9.518732	-8.686317	-5.543150	2.890455	5.980011	-6.301346
第 51 节龙身 y (m)	1.341137	2.540108	6.377946	7.249289	-3.827758	0.465829
第 101 节龙身 x (m)	2.913983	5.687116	5.361939	1.898794	-4.917371	-6.237722
第 101 节龙身 y (m)	-9.918311	-8.001384	-7.557638	-8.471614	-6.379874	3.936008
第 151 节龙身 x (m)	10.861726	6.682311	2.388757	1.005154	2.965378	7.040740
第 151 节龙身 y (m)	1.828753	8.134544	9.727411	9.424751	8.399721	4.393013
第 201 节龙身 x (m)	4.555102	-6.619664	-10.627211	-9.287720	-7.457151	-7.458662
第 201 节龙身 y (m)	10.725118	9.025570	1.359847	-4.246673	-6.180726	-5.263384
龙尾（后）x (m)	-5.305444	7.364557	10.974348	7.383896	3.241051	1.785033
龙尾（后）y (m)	-10.676584	-8.797992	0.843473	7.492370	9.469336	9.301164

表 **2**　问题一速度结果

	0 s	60 s	120 s	180 s	240 s	300 s
龙头 (m/s)	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000
第 1 节龙身 (m/s)	1.003438	1.003979	1.004721	1.005805	1.007534	1.010731
第 51 节龙身 (m/s)	1.102442	1.117833	1.138683	1.168535	1.214867	1.296737
第 101 节龙身 (m/s)	1.194710	1.222972	1.260920	1.314631	1.396713	1.538569
第 151 节龙身 (m/s)	1.281533	1.321245	1.374221	1.448597	1.561083	1.752802
第 201 节龙身 (m/s)	1.363833	1.413922	1.480409	1.573193	1.712474	1.947610
龙尾（后）(m/s)	1.398795	1.453173	1.525221	1.625543	1.775742	2.028471

部分计算结果的可视化结果如下，可以看出，随时间增加，每节龙身的速度也在逐渐增大，这可能是由于龙头把手逐渐靠近螺线中心；同时，在某一特定时刻，在舞龙队位置越靠后的把手速度越大，整个舞龙队的速度峰值在龙尾把手处达到。



(a) 首节龙身 **v-t** 图　　**(b)** 第一秒各节把手速度图

图 **2**　计算结果可视化

5.2 问题二模型的建立与求解

舞龙队仍沿问题一中的螺线盘入，要求找到发生碰撞时的临界情形与对应的临界状态。我们根据每个时刻把手的位置信息确定各板凳所占的区域，然后构造碰撞检测算法检测矩形区域的重合情况并降低算法的时间复杂度，最后将时间离散化检测每一时刻的碰撞情况。

5.2.1 舞龙队碰撞检测模型

（1）龙身所占区域位置集合

当舞龙队沿问题一设定的螺线盘入，随着龙头逐渐接近螺线中心，把手所在位置的极径在逐渐变小，整个舞龙队列的龙身分布逐渐密集。考虑一种临界情况，即对于相邻两个把手确定的唯一龙身，其除了与前后两块龙身发生重叠之外，与剩余 220 块龙身的某一块也发生了重叠，此时板凳之间发生了碰撞。我们认为这一时刻即为舞龙队盘入的终止时刻，若在此时刻往后舞龙队继续沿螺线盘入，由于活动空间较小，舞龙队的秩序将会受到一定影响。确定舞龙队的盘入终止时刻，即需要判断任意时刻舞龙队是否发生碰撞。

将各点在极坐标系下的坐标转换为平面直角坐标系下的坐标得到：

$$\begin{cases} y_i = b_0\theta_i sin\theta_i \\ x_i = b_0\theta_i cos\theta_i \\ k_i = \frac{\theta_i sin\theta_i - \theta_{i+1}sin\theta_{i+1}}{\theta_i cos\theta_i - \theta_{i+1}cos\theta_{i+1}} \end{cases}$$

其中 x_i, y_i 分别为点 P_i 的横纵坐标，k_i 为直线 $P_{i+1}P_i$ 的斜率。考虑相邻的两个把手构成的组合 $(P_i(r_i,\theta_i), P_{i+1}(r_{i+1},\theta_{i+1}))$ 包含于矩形区域 R_i，其由两两平行的四条曲线围成的封闭区域唯一确定：

$$\begin{cases} L_1 : y - y_i = k_i(x - x_i) + 0.15 \\ L_2 : y - y_i = k_i(x - x_i) - 0.15 \\ L_3 : y - \frac{y_i + y_{i+1}}{2} = -\frac{1}{k_i}(x - \frac{x_i + x_{i+1}}{2}) + B_i \\ L_4 : y - \frac{y_i + y_{i+1}}{2} = -\frac{1}{k_i}(x - \frac{x_i + x_{i+1}}{2}) - B_i \end{cases}$$

其中当 $i = 1$ 时，$B_i = 1.705$，当 $i \geq 2$ 时，$B_i = 1.1$。则所有矩形区域构成的龙身（包含龙头）所占区域位置集合 R 为:

$$R = \{R_1, R_2 \dots R_{222}, R_{223}\}$$

（2）任意时刻的碰撞检测算法 $f(t)$

在舞龙队盘入后，由于把手的位置集合 P 由时间 t 唯一确定，则龙身所占区域的位置集合也由时间 t 唯一确定。对于第 i 个矩形区域 R_i，其与 R_{i-1}, R_{i+1} 一定存在部分区域重合，为了检测该矩形区域是否与其余 220 块矩形区域发生重叠，定义第 i 个矩形区域 R_i 待检测矩形区域位置集合 R_i^1：

$$R_i^1 = R - \{R_{i-1}, R_i, R_{i+1}\}$$

对于特定时刻 t，算法 $f(t)$ 依次检查每个矩形区域 R_i 其与对应的待检测矩形区域位置集合 R_i^1 中的每个矩形区域是否发生重叠，若与任意一块矩形区域检测到了重叠，则说明

此时刻舞龙队列发生了板凳碰撞,此时算法 $f(t)$ 返回 $False$，否则返回 $True$。对于这种遍历的检测算法 $f(t)$，若处理的问题规模即板凳的数量为 n，则其时间复杂度为 $O(n^2)$。

（3）对碰撞检测算法的一种改进

进一步考虑，对于第 i 个板凳对应的矩形区域 R_i，可能发生碰撞的板凳对应的把手组合一定位于 (P_i, P_{i+1}) 所在等距螺线两侧的等距螺线上。因此我们可以对第 i 个矩形区域 R_i 待检测矩形区域位置集合 R_i^1 做进一步的筛选，得到更为精简的位置集合 R_i^2。考虑 R_i^1 中每个矩形区域对应的把手组合 $\{P_j(r_j, \theta_j), P_{j+1}(r_j+1, \theta_{j+1})\}$，我们只留下位于两侧螺旋线上的把手组合，即：

$$R_i^2 = R_i^1 - R_i^*$$

$$R_i^* = \{R_j | \theta_j < \theta_i - 2\pi \vee \theta_j > \theta_i + 2\pi\}$$

则只需对第 i 个矩形区域 R_i，依次检查其与集合 R_i^2 中的每个矩形区域是否发生重叠。对于改进后的检测算法 $f(t)^*$，若处理的问题规模为 n，则该检测算法的时间复杂度为 $O(n^{\frac{3}{2}})$，改进后的算法在时间复杂上有较大提升。

（4）离散化的数值求解方法

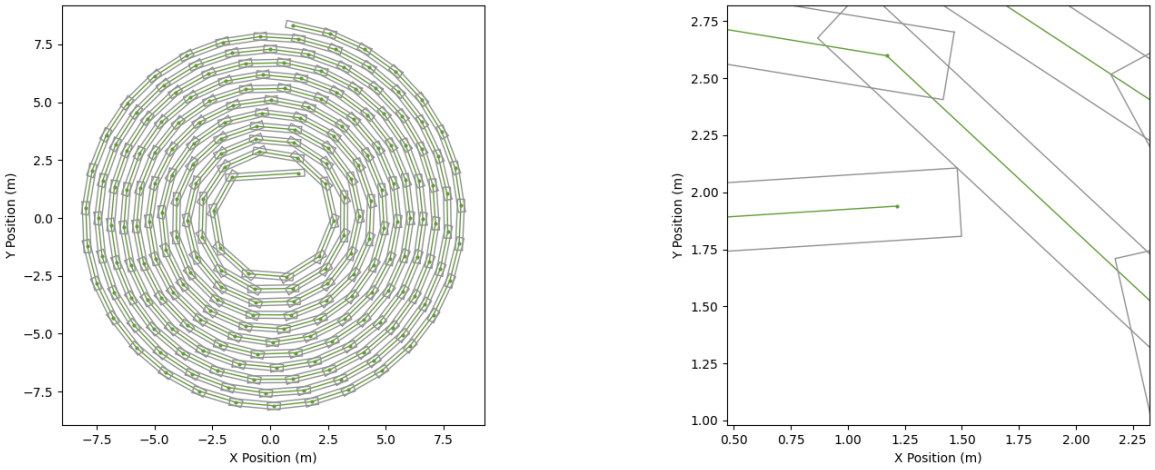
在确定了任意时刻的碰撞检测算法后，即可以从 $t=0$ 时刻开始检测是否有碰撞发生。对于连续的时间 t，我们可以离散化的进行数值求解，即确定时间步长 Δt，离散化的检测每一时刻的碰撞情况；同时，时间步长也决定了求解精度。

5.2.2 模型求解与可视化

基于前文所述的碰撞检测算法编写程序进行求解，求得舞龙队盘入的**终止时刻**为：

$$t = 412.47s$$

此时舞龙队碰撞情况及各节把手的位置速度如下：



(a) 碰撞整体示意图　　**(b)** 局部碰撞放大图

图 **3**　碰撞示意图

如图3所示，在 $t = 412.47s$ 时，舞龙队龙头把手已经较为接近中心，龙身分布较为密集，有效活动空间减少。由于第一节龙身较长，龙头首先与外层龙身发生碰撞，我们认为此时刻为盘入终止时刻，若继续盘入，可能会造成连续的碰撞从而使舞龙队失去秩序。

表 **3**　问题二结果

	横坐标 x(m)	纵坐标 y(m)	速度 (m/s)
龙头 (m/s)	1.206752	1.944933	1.000000
第 1 节龙身 (m/s)	-1.646659	1.750897	1.052601
第 51 节龙身 (m/s)	1.277627	4.327721	2.107947
第 101 节龙身 (m/s)	-0.532527	-5.880532	2.837114
第 151 节龙身 (m/s)	0.972547	-6.957009	3.437561
第 201 节龙身 (m/s)	-7.892625	-1.234460	3.962638
龙尾（后）(m/s)	0.952512	8.323200	4.176477

5.3 问题三模型的建立与求解

问题三要求确定龙头把手能够沿螺线顺利进入掉头空间边界的最小螺距。我们首先计算每个螺距对应的螺线与掉头区域边界的交点，接着根据问题二中的碰撞检测算法建

立防止板凳发生碰撞的约束条件，最后通过求解这些约束条件下的优化问题，得到最小螺距。

5.3.1 调头空间最小螺距模型

（1）舞龙队的调头区域

在舞龙队盘入到距离等距螺线中心一定距离时，进入到调头区域，并在该区域内完成由顺时针盘入调头切换为逆时针盘出的调头操作。该调头区域 Φ 为:

$$\Phi : x^2 + y^2 \leq (\frac{9}{2})^2$$

对于任意等距螺线 $r^*(\theta)$：

$$r = b^*\theta$$

其螺距 $H^* = b^* \times 2\pi$。该等距螺线与调头区域 Φ 的边缘存在交点 $A(\frac{9}{2}, \frac{9}{2b^*})$，即舞龙队的龙头把手 P_1 在经过该点后进入调头区域。

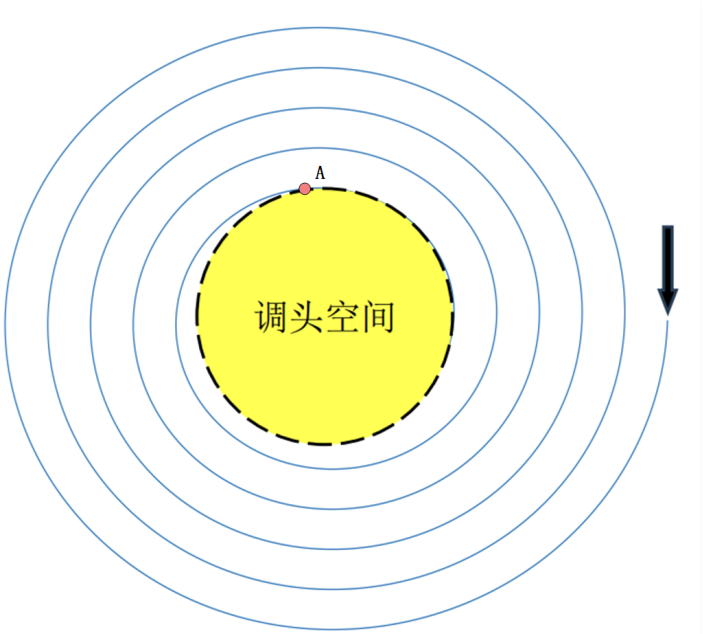


图 **4**　掉头空间示意图

（2）满足不发生碰撞的最小螺距

对于等距螺线 $r^*(\theta)$，其螺距 H^* 越小，在到达交点 A 时舞龙队的龙身分布越密集，想要使得舞龙队龙头前把手顺利沿着相应的螺线盘入到调头空间的边界，则在龙头把手到达交点 A 之前不能发生碰撞。若龙头把手到达交点 A 的时刻为 t_A，则对于碰撞检测算法 $f(t)^*$ 有：

$$f(t)^* = False \quad t \in [0, t_A]$$

其中 t_A 同样由下列方程给出：

$$L_{sum} - L_1(t_A) = b_0 \int_0^{\frac{9}{2b^*}} \sqrt{1+\theta^2}\, d\theta \tag{4}$$

5.3.2 最小螺距的求解

由前文分析可知，螺距 H^* 受一定条件约束，则该问题转化为求约束条件下单变量的最值问题。通过以一定步长逐步减少螺距 H^*, 并使用碰撞检测算法对约束条件进行判断，最终求得满足约束条件的**最小螺距** H^*_{min}:

$$H^*_{min} = 0.37m$$

5.4 问题四模型的建立与求解

问题四要求找到由两个相切圆弧组成的最短路径，其中前一段圆弧的半径为后一段的两倍，且两段圆弧分别与盘入和盘出曲线相切，计算各个时刻舞龙队的位置和速度信息。我们首先构造关于曲线长度的目标函数，通过几何分析其自由度再采用定一动一的思想得到曲线的全部解空间，然后根据板凳长度得到各把手相关信息。

5.4.1 调头路径模型

舞龙队在调头区域内的调头路径是由两段圆弧相切连接而成的 S 形曲线，其前一段圆弧的半径是后一段的 2 倍，具体示意图如下：

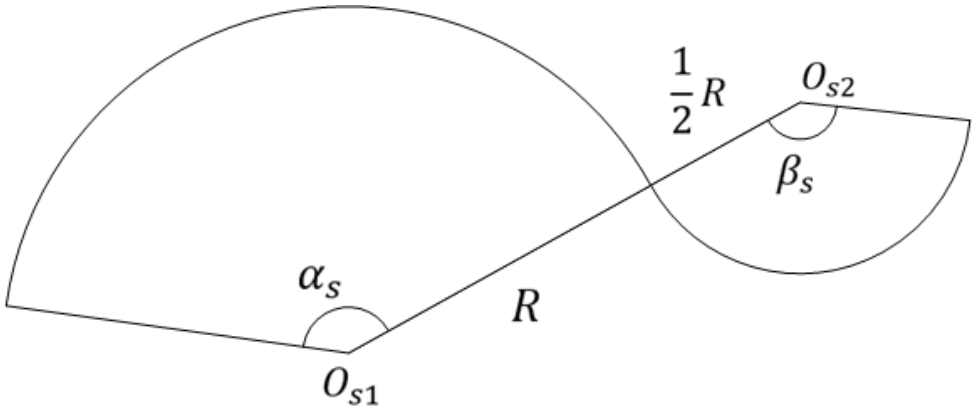


图 **5**　调头路径示意图

其中 S 形弧线的形状信息由参数 $\{\alpha_s, \beta_s, R\}$ 唯一确定，位置信息由坐标 $\{O_{s1}, O_{s2}\}$ 唯一确定。依题意调头路径需要满足如下约束：

1. 掉头路径必须包含在调头区域内，即调头操作必须在调头空间内完成；
2. S 形弧线必须与盘入、盘出螺线相切；

3. 舞龙队沿掉头路径进行调头时不能发生碰撞。

在满足以上约束条件的基础上，我们希望调整 S 形弧线的某些参数来减小调头曲线的总长度，即求满足约束条件下 S 形曲线长度 $|S|$ 的最小值：

$$min\ \ |S| = (\alpha_s + \frac{1}{2}\beta_s) \times R \tag{5}$$

5.4.2 约束条件下 S 形曲线的解空间

考虑一般情形如下图所示，其中绿色曲线为盘入曲线 $r(\theta)_{in}$：

$$r(\theta)_{in} : r = \frac{1.7}{2\pi}\theta$$

黑色曲线为盘出曲线 $r(\theta)_{out}$：

$$r(\theta)_{out} : r = -\frac{1.7}{2\pi}\theta$$

红色曲线为掉头区域 Φ 的边界:

$$x^2 + y^2 = (\frac{9}{2})^2$$

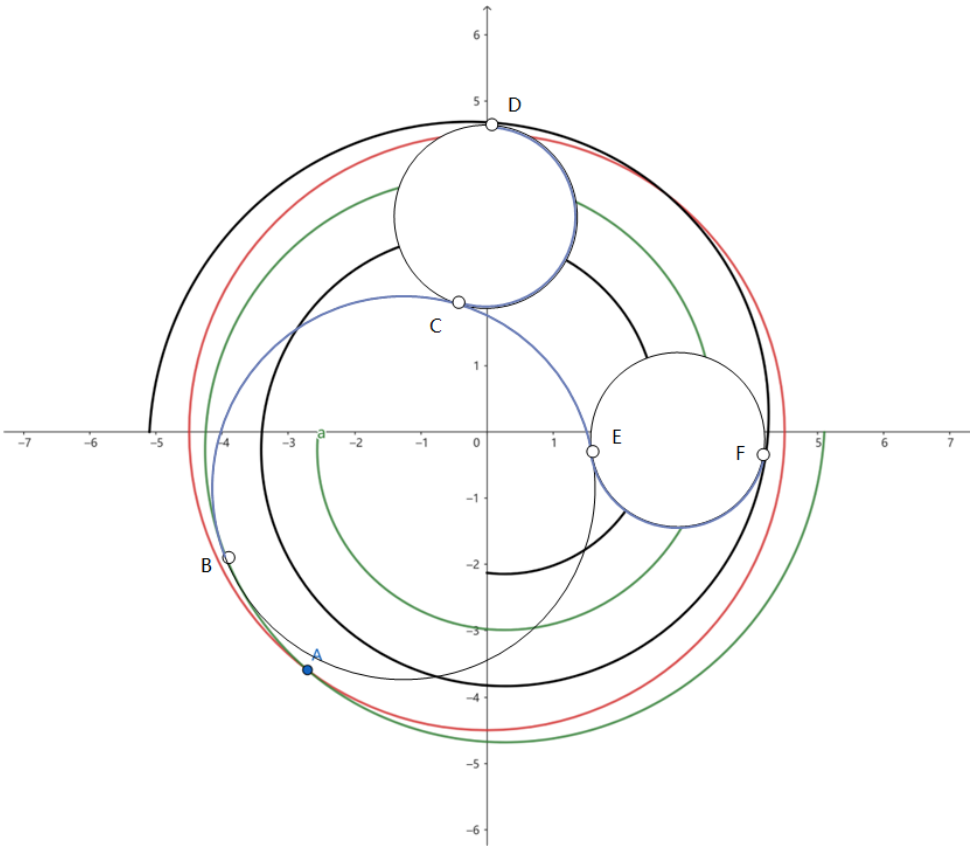


图 **6**　一般情形

A 点为调头区域 Φ 边界与等距螺线的交点，具体坐标为 $(\frac{9}{2}, \frac{90\pi}{17})$。龙头把手经过 A 点后，便可在任意合适位置进行调头。考虑一般情形，即在点 $B(r_B, \theta_B)$ 进行调头，掉头路径对应的 S 型曲线的前一段圆弧的半径为 R_B。为了便于更为准确地分析问题，我们将构成 S 形曲线的两条弧线对应的圆进行了补全，并采用定一动一的思想探寻满足约束条件下的全部解空间，即先确定调头开始的切点 $B(r_B, \theta_B)$，再调整 S 型曲线的前一段圆弧的半径 R_B，同时根据约束条件求取完整 S 型曲线。

如图7A 所示，在确定二元组 $S(\theta_B, R_B)$ 后，S 型曲线的前一段圆弧对应的圆 $O_1(x_1, y_1)$ 完全确定，由于后一段圆弧对应的圆 O_2 与圆 O_1 相切，其圆心 $O_2(x^*, y^*)$ 的运动轨迹为：

$$d: \quad (x^* - x_1)^2 + (y^* - y_1)^2 = (\frac{3R_B}{2})^2$$

圆 O_2 沿 d 进行运动，直到与螺线在某一点处相切，此时便满足全部约束条件。如图7A 中圆 O_3 与螺线相交于 F 点，此时满足相切约束条件的可行路径为 $B\overset{\frown}{C}D, B\overset{\frown}{C}EF$，由于 $B\overset{\frown}{C}EF = B\overset{\frown}{C}D + \overset{\frown}{CE}$，易知其中弧长度 $|S|$ 更小的 $B\overset{\frown}{C}D$ 为此情形下的最优调头路径。则二元组 $S(\theta_B, R_B)$ 即可唯一确定符合限制条件的调头曲线集合。

进一步分析，仅考虑满足相切约束条件，对确定的二元组 $S(\theta_B, R_B)$ 可行的完整弧线可能存在多个。如图7B，与内圈等距螺线也存在两个相切的弧线，即对于 $r(\theta)_{out}$ 中任意一段弧线 $\theta \in [\theta_k - 2\pi, \theta_k]$，均可能存在两条满足相切关系的弧线。综上，对于二元组 $S(\theta_B, R_B)$，其存在 2n 个满足相切限制的弧线，在本题限制的螺线圈数范围内 $n \leq 3$。

此外考虑图7C 情形，对于 $r(\theta)_{out}$ 中的某一段弧线 $\theta \in [\theta_k - 2\pi, \theta_k]$，仅存在一段符合相切条件的弧线，此时我们认为前文讨论中两条满足相切关系的弧线重合，仍不失一般性。

当盘出螺线相对于盘入螺线处于更内圈时，如图7D 情形所示。满足相切限制的路径为 $B\overset{\frown}{C}D, B\overset{\frown}{C}EF$，此时舞龙队沿该路径进行调头，并于 D 点或 F 点进入盘出螺线，该 S 形曲线与盘出曲线存在额外交点 G。当龙头把手到达 G 点时，由于舞龙队长度远大于弧长 $|S|$，有部分龙身尚未进入掉头路径，此时队伍将不可避免的发生碰撞，即 2n 个满足相切限制的弧线中。对于每个弧线与盘出螺线的切点坐标的极角 θ_m，按照升序排序（此时 $\theta_m < 0$）：

$$\{\theta_1, \theta_2, \ldots, \theta_m\}$$

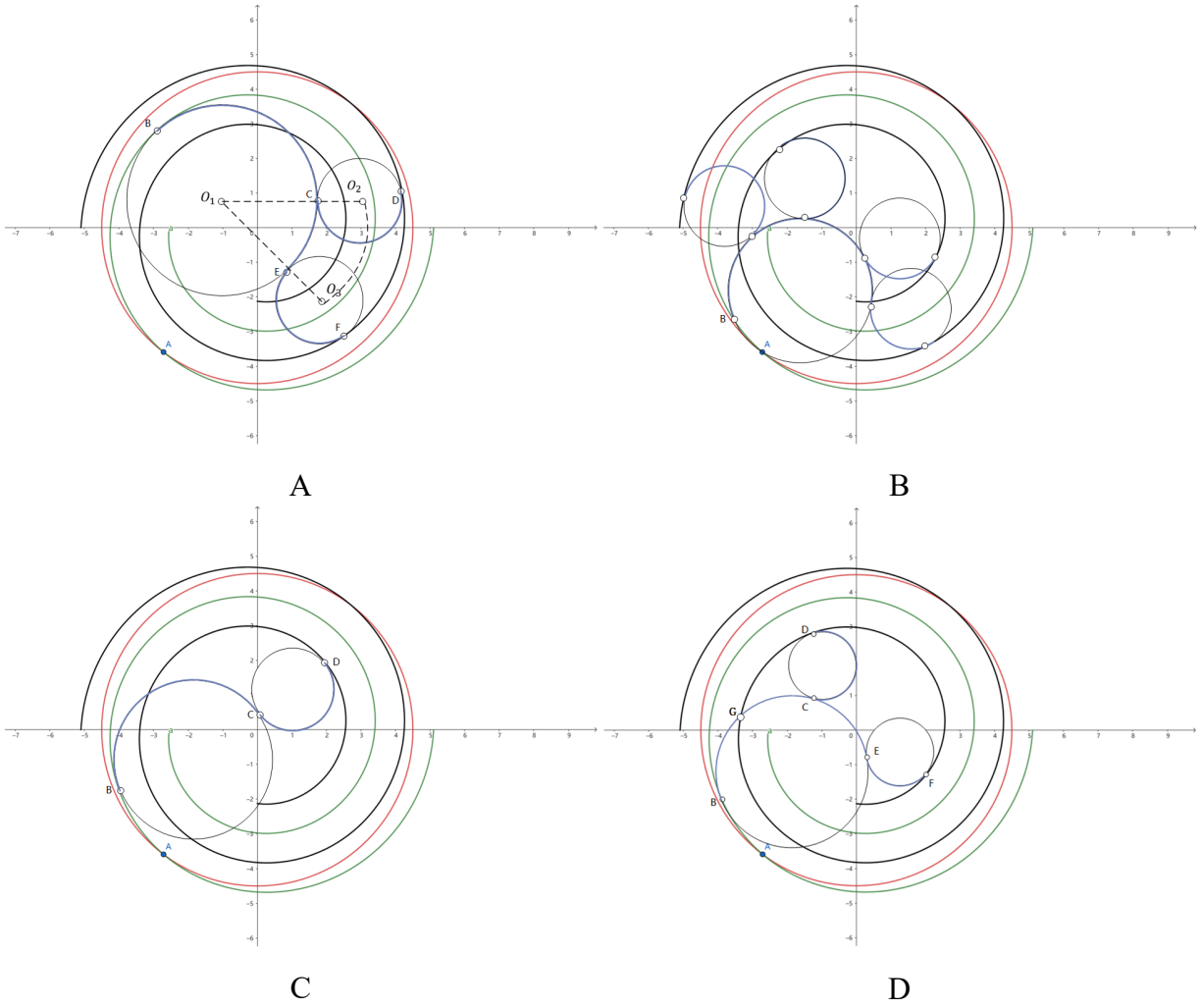


图 **7**　弧线相切示意图

结合上述讨论，仅 θ_1, θ_2 对应的掉头路径可能不会发生碰撞，即盘出螺线处于盘入螺线更外侧时对应的两个可行路径；且 θ_1 对应的弧线长 $|S_1|$ 小于 θ_2 对应的弧线长 $|S_2|$。因此，在考虑相切约束条件时，仅需考虑部分盘出螺线。进一步的对等距螺线进行分割，以确定符合需求的外圈等距螺线。如图8以 A 点为开始调头点为例，将 A 点与原点连线，依次交 $r(\theta)_{out}$ 于 B、C、D、E 四点，其中 B 点对应的极径为 θ_{Bout}，则符合的外圈弧线的极角 θ_{out} 满足 $\theta_{out} \in (\theta_{Bout} - 2\pi, \theta_{Bout})$，即曲线 $\overset{\frown}{BE}$ 部分。

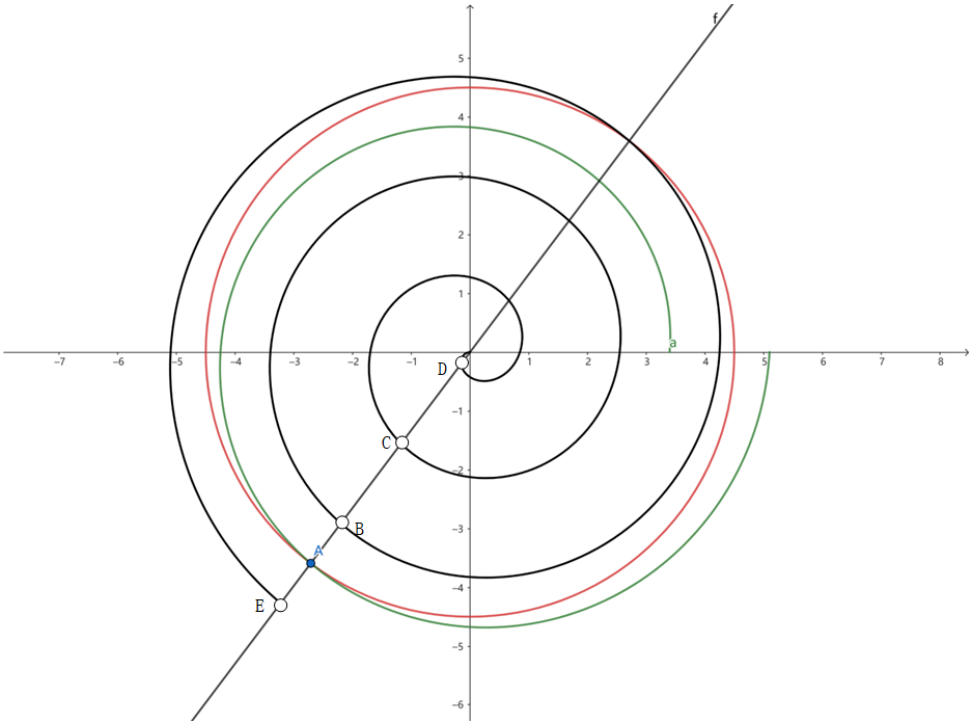


图 **8**　弧线分割图

对于满足上述约束条件的全部 S 形曲线，我们采用问题二中的碰撞检测模型 $f(t)^*$ 对调头过程进行碰撞检测，即可筛选掉会发生碰撞的掉头路径。至此，对于任意给定的掉头路径二元组 $S(\theta_B, R_B)$，我们均可找出符合全部约束条件的最优 S 形曲线。通过定一动一的思想，以一定步长 $\Delta\theta_B, \Delta R_B$ 不断调整 S 型曲线，即可得到一定精度下满足约束条件的 S 形曲线的全部解空间。

5.4.3 含掉头路径和盘出曲线的位置模型

（1）含调头路径和盘出螺线的龙头把手位置的确定

如图9所示，当舞龙队经过调头区域开始调头并依次进入盘出螺线后，把手所在的曲线可以分为四类：盘入曲线、调头路径前一段圆弧、掉头路径后一段圆弧、盘出曲线，即：

$$Location(P_i) \in \{in, turn1, turn2, out\}$$

在前文中已经求解得到最优调头曲线，将相关参数转化为平面直角坐标系，其中开始调头点为 $B(x_1, y_1)$，调头路径前一段圆弧的圆心 $O_1(x_2, y_2)$，对应的半径为 R，对应的圆心角为 α_s；两段圆弧的连接点 $C(x_3, y_3)$，调头路径后一段圆弧的圆心 $O_2(x_4, y_4)$，对应的半径为 $\frac{R}{2}$，对应的圆心角为 β_s。由于龙头把手的速度固定，则可以计算出在含调头路径和盘出螺线的路径曲线上，任意时刻龙头把手的位置坐标。

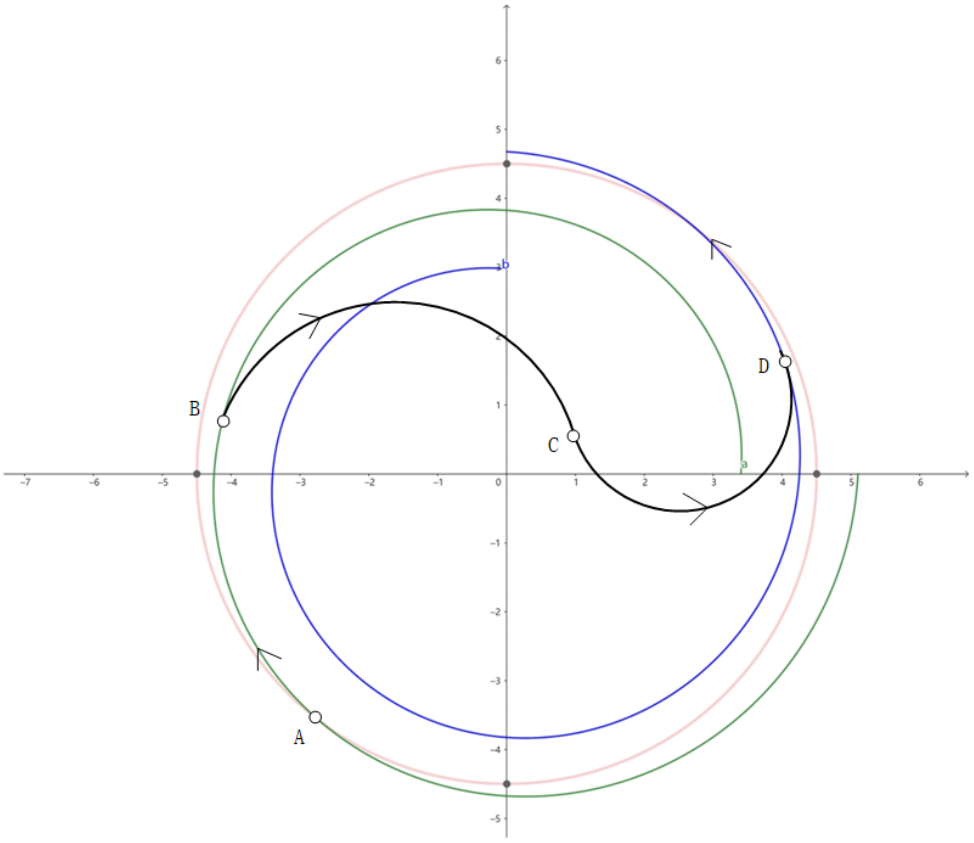


图 **9**　掉头路线示意图

$t=0$ 时刻龙头把手处于 B 点，则当 $t<0$ 时，龙头把手位于在盘入曲线上，其具体位置的求解方法在问题一中已经给出，由于 $t<0$，只需将原式中的 t 改为 $-t$，此处不再赘述；当 $0 \le t < R \times \alpha_s$ 时，龙头把手位于调头曲线的圆 O_1 上，利用向量的点乘公式，其经过任意时刻 t 的位置坐标 $P(t)=(x_t, y_t)$ 满足方程组：

$$\begin{cases} cos\frac{t}{R} = \frac{(x_1-x_2)\times(x_t-x_2)+(y_1-y_2)\times(y_t-y_2)}{R^2} \\ (x_t-x_2)^2+(y_t-y_2)^2 = R^2 \end{cases}$$

当 $R \times \alpha_s \le t < R \times (\alpha_s + \frac{\beta_s}{2})$ 时，龙头把手位于调头曲线的圆 O_2 上，其经过任意时刻 t 的位置坐标满足方程组：

$$\begin{cases} cos\frac{2(t-R\times\alpha_s)}{R} = \frac{4((x_3-x_4)\times(x_t-x_4)+(y_3-y_4)\times(y_t-y_4))}{R^2} \\ (x_t-x_4)^2+(y_t-y_4)^2 = (\frac{R}{2})^2 \end{cases}$$

当 $R \times (\alpha_s + \frac{\beta_s}{2}) \le t$ 时，龙头把手位于盘出螺线上，若掉头路径与盘出曲线的切点 D 的极坐标为 (r_{out}, θ_{out})，其经过任意时刻 t 的位置极坐标 (r_t, θ_t) 仍可通过路径积分求解 (此时 $\theta_t < 0$)：

$$\frac{1.7}{2\pi}\int_0^{-\theta_{out}} \sqrt{1+\theta^2}\,d\theta + (t-(R\times(\alpha_s+\frac{\beta_s}{2}))) = \frac{1.7}{2\pi}\int_0^{-\theta_t} \sqrt{1+\theta^2}\,d\theta$$

（2）含调头路径和盘出螺线的后续把手位置的迭代求解

对于相邻把手 (P_i, P_{i+1})，其不同之处仅在于所处曲线的方程可能不同，其余求解过程同问题一。距离条件构成的方程组同问题一：

$$\begin{cases} |P_iP_{i+1}| = 2.86 \ \ if \ \ i = 1 \\ |P_iP_{i+1}| = 1.65 \ \ if \ \ i >= 2 \\ |P_iP_{i+1}| = \sqrt{(r_icos\theta_i - r_{i+1}cos\theta_{i+1})^2 + (r_isin\theta_i - r_{i+1}sin\theta_{i+1})^2} \\ |\theta_{i+1} - \theta_i| < 2\pi \end{cases}$$

根据每个把手所处的曲线类型，分别带入不同的曲线方程：

$$\begin{cases} r_i = \frac{1.7}{2\pi}\theta_i \quad \text{if } \ Location(P_i) = in \\ (r_{i+1}cos\theta_{i+1} - x_2)^2 + (r_{i+1}sin\theta_{i+1} - y_2)^2 = R^2 \quad \text{if } \ Location(P_i) = turn1 \\ (r_{i+1}cos\theta_{i+1} - x_4)^2 + (r_{i+1}sin\theta_{i+1} - y_4)^2 = (\frac{R}{2})^2 \quad \text{if } \ Location(P_i) = turn2 \\ r_i = -\frac{1.7}{2\pi}\theta_i \quad \text{if } \ Location(P_i) = out \end{cases}$$

由于把手的位置信息是通过迭代进行的求解，在求解前其所处的曲线类型未知，因此，需通过前一个已知位置信息的把手来判断后续把手所处的曲线类型。考虑不同曲线间的连接点为 T（本文中的连接点有三个：盘入螺线与 S 形曲线前半段的连接点、S 形曲线前半段与后半段的连接点、S 形曲线后半段与盘出曲线的连接点），已知位置信息的把手坐标为 P_i^{know}，距离 P_i^{know} 最近的连接点为 T^*，则通过判断两者间的距离即可判断相邻后续把手所处的曲线类型。例如，对于 $i > 2$，若 $|P_i^{know}T^*| < 1.65, Location(P_i^{know}) = out$，则易知 $Location(P_{i+1}) = turn2$。至此即可迭代求解含调头路径和盘出螺线的全部把手位置信息。

5.4.4 含调头路径和盘出螺线的速度模型

当舞龙队在含掉头路径和盘出螺线的曲线路径上移动时，前文讨论得出每个把手沿刚体方向的角度的求解公式仍然适用，即：

$$\alpha_1 = \gamma_i - \alpha_2 - \angle OP_iP_{i+1} + k\pi$$

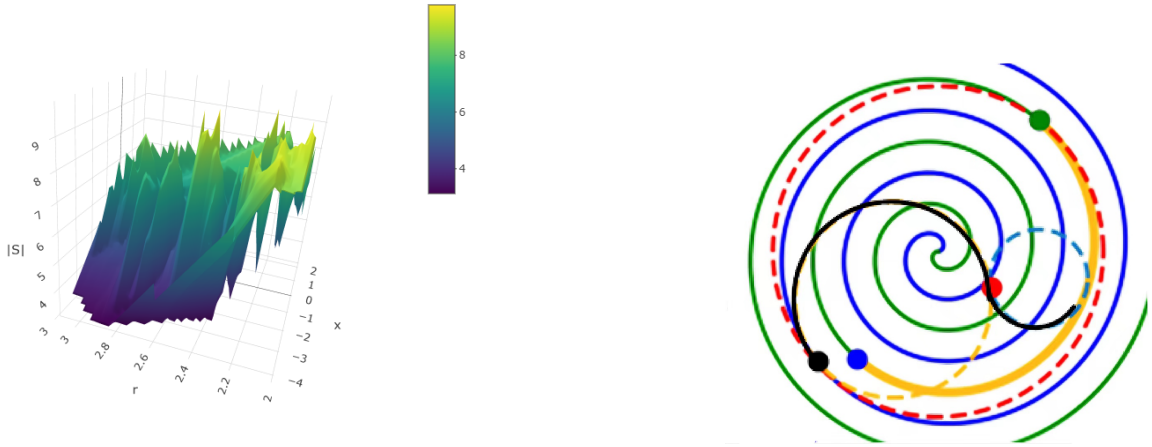
其中 $\gamma_i, \angle OP_iP_{i+1}$ 由当前把手 P_i, P_{i+1} 的位置坐标唯一确定，而含调头路径和盘出螺线的每个把手的位置信息在前文中已经求得。在不同的曲线上，α_2 由当前曲线在该点处微分求得的导数决定，下面给出不同曲线上 α_2 的求解公式：

$$\begin{cases} \alpha_2 = arctan(\frac{sin\theta_i + \theta_i cos\theta_i}{cos\theta_i - \theta_i sin\theta_i}) \ \ if \ \ Location(P_i) = in, out \\ \alpha_2 = arctan(\frac{x_2 - r_i cos\theta_i}{r_i sin\theta_i - y_2}) \ \ if \ \ Location(P_i) = turn1 \\ \alpha_2 = arctan(\frac{x_4 - r_i cos\theta_i}{r_i sin\theta_i - y_4}) \ \ if \ \ Location(P_i) = turn2 \end{cases}$$

即可根据 $|v_icos\alpha_1| = |v_{i+1}cos\beta_1|$ 依次迭代求解含调头路径和盘出螺线的全部把手速度信息。

5.4.5 模型求解与可视化

对于特定二元组 $S(\theta_B, R_B)$，其唯一确定可行的 S 型弧线及其弧长 $|S|$，即可将弧长 $|S|$ 看作二元函数 $|S| = S(\theta_B, R_B)$，并将可行解空间可视化，如图所示。



(a) 可行解空间　　**(b)** 第一秒各节把手速度图

图 **10**　第四问结果可视图

在全部解空间中求得最优掉头曲线，如图10中黑色曲线所示，其开始调头点为 $(-3.31, -2.97)$，前一段圆弧半径为 $2.67m$，两段圆弧的连接点坐标为 $(1.42, -0.93)$，调头完成点为 $(3.99, -1.19)$，最优 S 形曲线弧长 $|S|_{min} = 13.32m$。根据位置和速度求解模型得到任意时刻的把手的位置和速度，部分数据如下表所示。

表 **4**　问题四速度结果

	-100 s	-50 s	0 s	50 s	100 s
龙头 (m/s)	1	1	1	1	1
第 1 节龙身 (m/s)	1.002646	1.006325	1.005782	1.006302	1.008320
第 51 节龙身 (m/s)	1.048483	1.050027	1.051637	1.054200	1.045730
第 101 节龙身 (m/s)	1.098232	1.098405	1.098866	1.096084	1.096646
第 151 节龙身 (m/s)	1.142175	1.140774	1.143533	1.138880	1.139467
第 201 节龙身 (m/s)	1.189179	1.182567	1.184754	1.187558	1.185684

表 **5**　问题四位置结果

	-100 s	-50 s	0 s	50 s	100 s
龙头 x (m)	8.084881	6.760446	-3.309111	5.177005	1.733467
龙头 y (m)	2.912722	1.051647	-2.972295	3.227834	7.826349
第 1 节龙身 x (m)	6.761674	5.848996	-0.910954	5.952622	4.330826
第 1 节龙身 y (m)	5.448215	3.762525	-4.530642	0.475014	6.629135
第 51 节龙身 x (m)	-10.333491	-4.401477	1.624032	0.746747	-3.431044
第 51 节龙身 y (m)	3.647071	-8.584168	-7.965547	-3.118731	0.181061
第 101 节龙身 x (m)	-12.200233	9.645740	3.811180	0.746747	-3.431044
第 101 节龙身 y (m)	-3.987959	-6.685346	9.810463	-3.118731	0.181061
第 151 节龙身 x (m)	-14.427314	12.690216	-6.268288	0.746747	-3.431044
第 151 节龙身 y (m)	-1.124084	-4.621381	10.777195	-3.118731	0.181061
第 201 节龙身 x (m)	-11.355420	9.879573	-6.104858	0.746747	-3.431044
第 201 节龙身 y (m)	11.185849	-11.377724	12.760584	-3.118731	0.181061
龙尾（后）x (m)	-1.866680	1.047829	-2.776066	0.746747	-3.431044
龙尾（后）y (m)	-16.438763	15.671947	-14.561777	-3.118731	0.181061

5.5 问题五模型的建立与求解

舞龙队在前一问确定舞龙队的 S 形调头曲线，龙头的最大行进速度，使得所有把手的速度均不超过 2 m/s。由问题一的分析可知，龙头速度的变化会影响每个把手的速度，因此需要综合考虑**曲率和速度的传递关系**。通过建立**运动学模型**，可以计算每个把手的速度，并逐步调整龙头的速度，确保所有把手的速度都不超过 2 m/s，最终确定龙头的最大行进速度。

5.5.1 梯度下降法

梯度下降法是求解优化问题的一种常用迭代算法，该算法基于目标函数的梯度信息，通过不断向梯度的负方向移动、更新决策变量，从而逐步逼近最优解以最小化目标函数。在每次迭代过程中，算法计算当前点的梯度，并根据学习率更新解。由于梯度下

降法通过连续的迭代向下移动，可以有效地快速收敛，该算法的具体操作流程如下：

Algorithm 1 梯度下降法

procedure GRADIENTDESCENT($f(\mathbf{x}), \mathbf{x}_0, \eta, \epsilon$, max_iter)
    初始化 $\mathbf{x}_0$ 为初始点，设置 $k = 0$
    repeat
        计算梯度 $\mathbf{g}_k = \nabla f(\mathbf{x}_k)$
        更新 $\mathbf{x}_{k+1} = \mathbf{x}_k - \eta \mathbf{g}_k$
        $k \leftarrow k + 1$
    until $\|\mathbf{g}_k\| < \epsilon$ 或者 $k \geq$ max_iter
    return $\mathbf{x}_k$
end procedure

5.5.2 模型求解

在问题四得到的最优掉头曲线的基础上，我们通过速度求解模型求得任意龙头把手下的全部把手速度信息，并在满足龙头把手速度峰值约束的前提下，根据梯度下降求得最大龙头把手速度 v_{max}：

$$v_{max} = 1.89 m/s$$

六、 模型的评价

6.1 模型的优点

1. 利用物理学中刚体的物理性质精确地给出了相邻把手间的速度关系。
2. 分析调头曲线时，通过定一动一的思想给出了满足约束条件的全部解空间。
3. 改进后的碰撞检测算法可以有效的处理大规模情形。

2. 在速度求解过程中的数值求解方法存在一定误差。

6.2 模型的缺点

1. 将把手视为质点，虽简化了运动模型，却也丢失了如把手半径等有效信息。
2. 在速度求解过程中的数值求解方法存在一定误差。

参考文献

[1] 李谋涛. 守正创新：基于非物质文化遗产“徽州板凳龙”传承人的口述史考察. 通化师范学院学报, 45(03):31–37, 2024.

[2] 刘崇军. 等距螺旋的原理与计算. 数学的实践与认识, 48(11):165–174, 2018.

[3] 陈立群. 关于平面运动刚体两点速度和加速度关系式. 力学与实践, 36(06):786–787, 2014.

附录 A　问题一代码

```
import numpy as np
import pandas as pd
from scipy.integrate import quad
from scipy.optimize import fsolve
import math


# 参数设置
spiral_pitch = 0.55 # 螺旋线的螺距，单位：米（55 cm）
head_speed = 1 # 龙头的速度，单位：米每秒
initial_circle = 16 # 龙头初始位置所在的圈数
time_interval = 300 # 时间间隔，从0秒到300秒
head_length = 2.86 # 龙头两个把手的长度，单位：米
bench_length = 1.65 # 龙身两个把手的长度，单位：米（165 cm）
v = 1 # 龙头的速度m/s

# 初始位置计算（龙头在第16圈，A点处）
initial_angle = 2 * np.pi * initial_circle # 初始角度，单位：弧度
initial_radius = initial_circle * spiral_pitch # 初始半径计算

# 计算b
b = spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = initial_radius - b * initial_angle


# 定义螺旋线弧长积分的被积函数
def integrand(theta, a, b):
return np.sqrt(b ** 2 + (a + b * theta) ** 2)


# 定义计算弧长的函数
def arc_length_function(theta, a, b):
arc_length, _ = quad(integrand, 0, theta, args=(a, b))
return arc_length


# 定义根据弧长L反解角度theta的函数
def inverse_arc_length(t, a, b, v):
# 计算16圈弧长（积分从0到16圈对应的角度范围）
arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))

L = arc_length - v * t
```

```
# 定义方程：弧长与L的差值
def equation(theta):
return arc_length_function(theta, a, b) - L

# 初始猜测值，可以根据实际情况调整
initial_guess = 2 * np.pi

# 使用fsolve求解方程，找到theta
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution


# 根据给定的角度theta计算螺旋线上的x和y坐标。
def compute_coordinates(theta, a, b):
r = a + b * theta # 计算当前半径
x = r * np.cos(theta) # 计算x坐标
y = r * np.sin(theta) # 计算y坐标
return x, y


# 计算螺旋线在给定角度theta处的切线斜率tan 的角度
def compute_tangent_angle(theta, a, b):
# 计算当前半径
r = a + b * theta

# 计算 dx/d 和 dy/d
dx_dtheta = b * np.cos(theta) - r * np.sin(theta)
dy_dtheta = b * np.sin(theta) + r * np.cos(theta)

# 计算切线斜率
slope = dy_dtheta / dx_dtheta

# 计算角度
angle = np.arctan(slope)

return angle


# 计算三角形的三个角度，给定三边 a，b，c。
def calculate_angles(a, b, c):
# 计算角度的余弦值
cos_alpha = (b ** 2 + c ** 2 - a ** 2) / (2 * b * c)
cos_beta = (a ** 2 + c ** 2 - b ** 2) / (2 * a * c)

# 计算角度（弧度）
alpha = np.arccos(cos_alpha)
```

```
beta = np.arccos(cos_beta)

# # 转换为度数
# alpha_degrees = np.degrees(beta)
# beta_degrees = np.degrees(alpha)

return beta,alpha

# 计算点相对于圆心的极角（弧度
def angle_from_center(x, y, x_center, y_center):
k=math.atan2(y - y_center, x - x_center)
if k<0:
k=k+np.pi*2
return k
# 计算三角形的角度，给定xy。
def calculate_acute_angle(x1, y1, x2, y2):
# 计算两点之间的x和y的差值
dx = x2 - x1
dy = y2 - y1

# 计算直角三角形的斜边长度
hypotenuse = math.sqrt(dx ** 2 + dy ** 2)

# 计算锐角的角度（以弧度为单位）
# 选择dx或dy作为邻边，这里我们使用dx来计算角度
if hypotenuse == 0: # 处理两点相同的特殊情况
return 0

acute_angle_radians = math.acos(abs(dx) / hypotenuse)

if acute_angle_radians > np.pi/2:
acute_angle_radians=np.pi-acute_angle_radians

return acute_angle_radians

# 定义函数计算并保存每个把手的位置和速度
def compute_and_save_positions(time_interval, bench_length, a, b, v,
    filename="test_positions.xlsx",
filename_radii_angles="test_radii_angles.xlsx", filename_v="test_v.xlsx"):
# 初始化结果字典
results = {
  'Time (s)': [],
  'Head x (m)': [],
  'Head y (m)': []
}

results_radii_angles = {
```

```
    'Time (s)': [],
    'Head Radius (m)': [],
    'Head Angle (rad)': []
}

results_v = {
    'Time (s)': [],
    'v (m/s)': []
}

# 计算每个把手在各个时间点的位置
for t in range(time_interval + 1):
# 计算龙头的角度
head_theta = inverse_arc_length(t, a, b, v)

# 计算龙头的xy坐标
head_x, head_y = compute_coordinates(head_theta, a, b)

# 计算龙头的r
head_radius = a + b * head_theta

# 保存时间和龙头坐标
results['Time (s)'].append(t)
results['Head x (m)'].append(head_x)
results['Head y (m)'].append(head_y)

# 保存时间和龙头角度和r
results_radii_angles['Time (s)'].append(t)
results_radii_angles['Head Radius (m)'].append(head_radius)
results_radii_angles['Head Angle (rad)'].append(head_theta)

# 保存时间和龙头速度
results_v['Time (s)'].append(t)
results_v['v (m/s)'].append(v)

# 计算龙身和龙尾各把手的位置
previous_theta = head_theta
previous_x, previous_y = head_x, head_y
previous_radius = head_radius
previous_v = v

for i in range(1, 224): # 223 节: 1个龙头 + 221个龙身 + 1个龙尾
# 使用优化方法找到下一个把手的位置
if i == 1:
length = head_length
else:
length = bench_length
```

```
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - previous_x) ** 2 + (y - previous_y) ** 2)

return distance - length

# 初始猜测值为前一个把手的角度加上一个小的增量
initial_guess = previous_theta + 0.01

# 求解下一个把手的位置，确保角度递增
next_theta = fsolve(find_next_position, initial_guess)[0]

# 确保角度差值不超过2
if next_theta - previous_theta > 2 * np.pi:
next_theta = previous_theta + 2 * np.pi
print('error')

# 确保角度递增
if next_theta < previous_theta:
print('error')

# 当前把手的xy和r
next_x, next_y = compute_coordinates(next_theta, a, b)
next_radius = a + b * next_theta

# 计算速度
# angle_xy=calculate_acute_angle(previous_x, previous_y, next_x, next_y)

alpha_degrees, beta_degrees=calculate_angles(previous_radius, next_radius, length)

angle_previous = angle_from_center(previous_x, previous_y, 0,
    0)-compute_tangent_angle(previous_theta, a, b)-alpha_degrees
angle_next = angle_from_center(next_x, next_y, 0, 0)-compute_tangent_angle(next_theta, a,
    b)-beta_degrees
cos_previous = np.cos(angle_previous)
cos_next =np.cos(angle_next)
next_v = abs((previous_v * cos_previous) / cos_next)

# 保存每个把手的位置
if f'Segment {i} x (m)' not in results:
results[f'Segment {i} x (m)'] = []
results[f'Segment {i} y (m)'] = []

results[f'Segment {i} x (m)'].append(next_x)
results[f'Segment {i} y (m)'].append(next_y)
```

```
# 保存每个把手的半径和角度
if f'Segment {i} Radius (m)' not in results_radii_angles:
results_radii_angles[f'Segment {i} Radius (m)'] = []
results_radii_angles[f'Segment {i} Angle (rad)'] = []

results_radii_angles[f'Segment {i} Radius (m)'].append(next_radius)
results_radii_angles[f'Segment {i} Angle (rad)'].append(next_theta)

# 保存每个把手的速度
if f'Segment {i} v (m/s)' not in results_v:
results_v[f'Segment {i} v (m/s)'] = []

results_v[f'Segment {i} v (m/s)'].append(next_v)

# 更新前一个把手的位置和速度
previous_theta = next_theta
previous_x, previous_y = next_x, next_y
previous_radius = next_radius
previous_v = next_v
# 将结果保存到 Excel 文件
df = pd.DataFrame(results)
df.to_excel(filename, index=False)
print(f"每个把手的位置已保存到 '{filename}'.")

# 将半径和角度结果保存到另一个 Excel 文件
df_radii_angles = pd.DataFrame(results_radii_angles)
df_radii_angles.to_excel(filename_radii_angles, index=False)
print(f"每个把手的半径和角度已保存到 '{filename_radii_angles}'.")

# 将v保存到另一个 Excel 文件
df_v = pd.DataFrame(results_v)
df_v.to_excel(filename_v, index=False)
print(f"每个把手的v已保存到 '{filename_v}'.")


# 主函数
def main():
compute_and_save_positions(time_interval, bench_length, a, b, head_speed)


if __name__ == "__main__":
main()
```

附录 B　问题二重叠检测代码

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

# 读取Excel文件
file_path = 'dragon_positions_350_415.xlsx' # 使用正确的文件路径
df = pd.read_excel(file_path)

# 过滤时间数据
df_430_to_440 = df[(df.iloc[:, 0] >= 412.45) & (df.iloc[:, 0] <= 412.48)]

# 提取每隔1秒的数据
time_intervals = df_430_to_440.iloc[::1, 0] # 每隔1秒的时间点
positions = df_430_to_440.iloc[::1, 1:] # 每隔1秒的各点位置

# 定义板凳宽度和默认长度
default_bench_length = 2.2 # 米
first_bench_length = 3.41 # 米，用于第一个矩形
bench_width = 0.3 # 米

# 计算矩形的四个顶点
def calculate_rectangle_corners(x1, y1, x2, y2, length, width):
# 计算中点
center_x = (x1 + x2) / 2
center_y = (y1 + y2) / 2

# 计算相邻两个点之间的角度
angle = np.arctan2(y2 - y1, x2 - x1) # 使用 atan2 计算角度（弧度）

# 计算矩形的半长度和半宽度
half_length = length / 2
half_width = width / 2

# 计算旋转后的四个顶点坐标
dx_l = half_length * np.cos(angle) # 长度方向上的 x 位移
dy_l = half_length * np.sin(angle) # 长度方向上的 y 位移

dx_w = half_width * np.sin(angle) # 宽度方向上的 x 位移（与长度正交）
dy_w = half_width * np.cos(angle) # 宽度方向上的 y 位移

# 四个角点（根据旋转角度调整）
corner1 = (center_x - dx_l + dx_w, center_y - dy_l - dy_w)
corner2 = (center_x + dx_l + dx_w, center_y + dy_l - dy_w)
corner3 = (center_x + dx_l - dx_w, center_y + dy_l + dy_w)
corner4 = (center_x - dx_l - dx_w, center_y - dy_l + dy_w)
```

```
return [corner1, corner2, corner3, corner4]

# 分离轴定理（SAT）重叠检测
def is_overlapping(rect1, rect2):
def project_polygon(axis, polygon):
dots = [np.dot(axis, point) for point in polygon]
return min(dots), max(dots)

def normalize(vector):
length = np.linalg.norm(vector)
return vector / length if length > 0 else vector

# 获取两个矩形的四个角点
for i in range(4):
# 计算两个多边形的边缘向量
edge1 = np.array(rect1[i]) - np.array(rect1[(i + 1) % 4])
edge2 = np.array(rect2[i]) - np.array(rect2[(i + 1) % 4])

# 计算分离轴
axis1 = normalize(np.array([-edge1[1], edge1[0]]))
axis2 = normalize(np.array([-edge2[1], edge2[0]]))

# 投影两个矩形到分离轴上
min1, max1 = project_polygon(axis1, rect1)
min2, max2 = project_polygon(axis1, rect2)
if max1 < min2 or max2 < min1:
return False

min1, max1 = project_polygon(axis2, rect1)
min2, max2 = project_polygon(axis2, rect2)
if max1 < min2 or max2 < min1:
return False

return True

# 绘制每个时间点的单独图像并检测重叠
for i in range(len(time_intervals)):
time = time_intervals.iloc[i]
x_positions = positions.iloc[i, ::2] # 提取x坐标
y_positions = positions.iloc[i, 1::2] # 提取y坐标

plt.figure(figsize=(6, 6))
plt.plot(x_positions, y_positions, marker='o', linestyle='-', label=f'Time = {time} s')

# 存储该时刻的矩形
rectangles = []
```

```
# 画出相邻的两个把手的矩形框
for j in range(len(x_positions) - 1):
x1, y1 = x_positions.iloc[j], y_positions.iloc[j]
x2, y2 = x_positions.iloc[j + 1], y_positions.iloc[j + 1]

# 针对第一个矩形，使用特殊的长度
if j == 0:
length = first_bench_length
else:
length = default_bench_length

# 获取矩形的四个顶点
corners = calculate_rectangle_corners(x1, y1, x2, y2, length, bench_width)
rectangles.append(corners)

# 依次连接四个点，画出矩形
rect_x = [corners[0][0], corners[1][0], corners[2][0], corners[3][0], corners[0][0]]
rect_y = [corners[0][1], corners[1][1], corners[2][1], corners[3][1], corners[0][1]]
plt.plot(rect_x, rect_y, 'r-')

plt.title(f'Positions of Dragon at Time {time} s')
plt.xlabel('X Position (m)')
plt.ylabel('Y Position (m)')
plt.legend()
plt.grid(True)
plt.axis('equal') # 保证图形比例相等
plt.show()

# 检测当前时间点的矩形是否有重叠
has_overlap = False
for m in range(len(rectangles)):
for n in range(m + 2, len(rectangles)): # 跳过相邻的矩形
if is_overlapping(rectangles[m], rectangles[n]):
print(f"在时间 {time} 秒，矩形 {m} 和 矩形 {n} 存在重叠")
has_overlap = True

if not has_overlap:
print(f"在时间 {time} 秒，所有非相邻矩形都没有重叠")
```

附录 C　问题三代码

```
import numpy as np
from scipy.optimize import fsolve
from scipy.integrate import quad
import math
```

```
# 参数设置
head_speed = 1 # 龙头的速度，单位：米每秒
initial_circle = 16 # 龙头初始位置所在的圈数
time_step = 1 # 时间步长，单位：秒
head_length = 2.86 # 龙头两个把手的长度，单位：米
bench_length = 1.65 # 龙身两个把手的长度，单位：米（165 cm）
v = 1 # 龙头的速度m/s
target_radius = 4.5 # 结束条件，龙头的半径

# 初始位置计算（龙头在第16圈，A点处）
initial_angle = 2 * np.pi * initial_circle # 初始角度，单位：弧度


# 定义螺旋线弧长积分的被积函数
def integrand(theta, a, b):
return np.sqrt(b ** 2 + (a + b * theta) ** 2)

# 定义计算弧长的函数
def arc_length_function(theta, a, b):
return quad(integrand, 0, theta, args=(a, b))[0]

# 定义根据弧长L反解角度theta的函数
def inverse_arc_length(t, a, b, v):
arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))
L = arc_length - v * t
def equation(theta):
return arc_length_function(theta, a, b) - L
initial_guess = 2 * np.pi
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution

# 根据给定的角度theta计算螺旋线上的x和y坐标
def compute_coordinates(theta, a, b):
r = a + b * theta # 计算当前半径
x = r * np.cos(theta) # 计算x坐标
y = r * np.sin(theta) # 计算y坐标
return x, y

# 计算矩形的四个顶点的函数
def calculate_rectangle_corners(x1, y1, x2, y2, length, width):
center_x = (x1 + x2) / 2
center_y = (y1 + y2) / 2
angle = np.arctan2(y2 - y1, x2 - x1)

half_length = length / 2
half_width = width / 2
```

```
dx_l = half_length * np.cos(angle)
dy_l = half_length * np.sin(angle)

dx_w = half_width * np.sin(angle)
dy_w = half_width * np.cos(angle)

corner1 = (center_x - dx_l + dx_w, center_y - dy_l - dy_w)
corner2 = (center_x + dx_l + dx_w, center_y + dy_l - dy_w)
corner3 = (center_x + dx_l - dx_w, center_y + dy_l + dy_w)
corner4 = (center_x - dx_l - dx_w, center_y - dy_l + dy_w)

return [corner1, corner2, corner3, corner4]

# 分离轴定理（SAT）重叠检测
def is_overlapping(rect1, rect2):
def project_polygon(axis, polygon):
dots = [np.dot(axis, point) for point in polygon]
return min(dots), max(dots)

def normalize(vector):
length = np.linalg.norm(vector)
return vector / length if length > 0 else vector

for i in range(4):
edge1 = np.array(rect1[i]) - np.array(rect1[(i + 1) % 4])
edge2 = np.array(rect2[i]) - np.array(rect2[(i + 1) % 4])

axis1 = normalize(np.array([-edge1[1], edge1[0]]))
axis2 = normalize(np.array([-edge2[1], edge2[0]]))

min1, max1 = project_polygon(axis1, rect1)
min2, max2 = project_polygon(axis1, rect2)
if max1 < min2 or max2 < min1:
return False

min1, max1 = project_polygon(axis2, rect1)
min2, max2 = project_polygon(axis2, rect2)
if max1 < min2 or max2 < min1:
return False

return True

# 定义函数计算每个把手的位置，并检测碰撞
def compute_positions_and_check_collisions(spiral_pitch, v, target_radius, bench_length):
# 初始化参数
initial_angle = 2 * np.pi * initial_circle # 初始角度，单位：弧度
```

```
initial_radius = initial_circle * spiral_pitch # 初始半径计算

b = spiral_pitch / (2 * np.pi)
a = initial_radius - b * initial_angle

t = 0
while True:
# 计算龙头的角度
head_theta = inverse_arc_length(t, a, b, v)
head_x, head_y = compute_coordinates(head_theta, a, b)
head_radius = a + b * head_theta

if head_radius <= target_radius:
break

rectangles = [] # 存储矩形的顶点坐标
previous_theta = head_theta
previous_x, previous_y = head_x, head_y

# 计算同一时刻的所有 223 个矩形
for i in range(223):
length = head_length if i == 0 else bench_length
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - previous_x) ** 2 + (y - previous_y) ** 2)
return distance - length

next_theta = fsolve(find_next_position, previous_theta + 0.01)[0]
next_x, next_y = compute_coordinates(next_theta, a, b)

corners = calculate_rectangle_corners(previous_x, previous_y, next_x, next_y, length, 0.3)
rectangles.append(corners)

previous_theta = next_theta
previous_x, previous_y = next_x, next_y

# 碰撞检测：只检测同一时刻的矩形
has_collision = False
for m in range(len(rectangles)):
for n in range(m + 2, len(rectangles)): # 跳过相邻矩形
if is_overlapping(rectangles[m], rectangles[n]):
has_collision = True
break
if has_collision:
break

if has_collision:
```

```
return False # 发生碰撞
t += time_step

return True # 没有发生碰撞

# 主函数
def main():
min_pitch = None
for spiral_pitch in np.arange(0.55, 0.30, -0.01):
if compute_positions_and_check_collisions(spiral_pitch, v, target_radius, bench_length):
min_pitch = spiral_pitch
print(f"找到了不碰撞的最小螺距：{min_pitch:.2f}")
else:
print(f"碰撞螺距：{spiral_pitch:.2f}")


if min_pitch is None:
print("所有螺距下都发生了碰撞。")

if __name__ == "__main__":
main()
```

附录 D　问题四路径计算代码

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
import pandas as pd

# 参数设置
spiral_pitch = 1.7 # 螺距
R = 4.5 # 调头圆的半径

# 盘入螺线方程：r = b * ，从外向内盘绕
def inward_spiral(theta, b):
return b * theta

# 盘出螺线方程：r = -b * ，从内向外盘绕
def outward_spiral(theta, b):
return -b * theta

# 交点求解：找到螺线与调头圆的交点
def find_intersection_in(b, spiral_func):
def intersection_eq(theta):
return spiral_func(theta, b) - R # 当螺线半径等于调头圆半径时
```

```
theta_guess = 1.0 # 初始猜测
theta_solution = fsolve(intersection_eq, theta_guess)[0]
r_intersection = spiral_func(theta_solution, b)
x_intersection = r_intersection * np.cos(theta_solution)
y_intersection = r_intersection * np.sin(theta_solution)
return theta_solution, x_intersection, y_intersection

# 交点求解：找到螺线与调头圆的交点
def find_intersection_out(b, spiral_func):
def intersection_eq(theta):
return spiral_func(theta, b) - R

theta_guess = 1.0
theta_solution = fsolve(intersection_eq, theta_guess)[0]
r_intersection = spiral_func(theta_solution, b)
x_intersection = r_intersection * np.cos(theta_solution + np.pi)
y_intersection = r_intersection * np.sin(theta_solution)
return theta_solution, x_intersection, y_intersection

# 计算盘入螺线在交点处的切线角度
def calculate_tangent_angle(theta_inward, b):
r = inward_spiral(theta_inward, b)
dr_dtheta = b # dr/d  = b
dx_dtheta = dr_dtheta * np.cos(theta_inward) - r * np.sin(theta_inward)
dy_dtheta = dr_dtheta * np.sin(theta_inward) + r * np.cos(theta_inward)
tangent_angle = np.arctan2(dy_dtheta, dx_dtheta)
return tangent_angle

# 计算圆1和圆2的切点
def calculate_tangent_point(x_center_circle1, y_center_circle1, x_center_circle2,
    y_center_circle2, radius_circle1, radius_circle2):
# 切点的位置位于圆心连线上的一个比例点
ratio = radius_circle1 / (radius_circle1 + radius_circle2)
x_tangent = x_center_circle1 + ratio * (x_center_circle2 - x_center_circle1)
y_tangent = y_center_circle1 + ratio * (y_center_circle2 - y_center_circle1)
return x_tangent, y_tangent

# 计算弧长 (顺时针 or 逆时针)
def calculate_arc_length(x1, y1, x2, y2, radius, clockwise=True):
# 计算圆心角（弧度制）
angle1 = np.arctan2(y1, x1)
angle2 = np.arctan2(y2, x2)

# 计算角度差
delta_angle = angle2 - angle1
```

```
# 对于顺时针方向的圆1，角度差应为负值
if clockwise:
if delta_angle > 0:
delta_angle -= 2 * np.pi
# 对于逆时针方向的圆2，角度差应为正值
else:
if delta_angle < 0:
delta_angle += 2 * np.pi

# 计算弧长
arc_length = radius * abs(delta_angle)
return arc_length


# 通过盘出螺线方程，根据极径 r 计算盘出螺线的极坐标角度 theta
def calculate_spiral_angle(cut_x, cut_y, b):
# 计算极径 r
r = np.sqrt(cut_x ** 2 + cut_y ** 2)

# 通过 r = -b * theta，求解 theta
theta = -r / b
return theta

# 主函数
def main():
b = spiral_pitch / (2 * np.pi)
print(f"螺距参数 b = {b:.4f}")

# 盘入螺线与调头圆的交点
theta_inward, x_inward_intersection, y_inward_intersection = find_intersection_in(b,
    inward_spiral)

# 盘出螺线与调头圆的交点
theta_outward, x_outward_intersection, y_outward_intersection = find_intersection_out(b,
    outward_spiral)

# 圆2相切的范围：点B与点E
theta_outward_B = theta_outward + np.pi
theta_outward_E = theta_outward

# 自动调整 move_angle 的过程
results = [] # 用于保存数据
for move_angle in np.linspace(0.01, np.pi, 100):
print(f"尝试 move_angle: {move_angle:.4f} 弧度")

# 移动盘入螺线的切点
theta_inward_moved = theta_inward - move_angle
```

```
x_inward_moved = inward_spiral(theta_inward_moved, b) * np.cos(theta_inward_moved)
y_inward_moved = inward_spiral(theta_inward_moved, b) * np.sin(theta_inward_moved)

# 计算新的切线角度
tangent_angle = calculate_tangent_angle(theta_inward_moved, b)

# 盘出螺线 B 到 E 段
theta_vals_B_E = np.linspace(theta_outward_E, theta_outward_B, 500)
r_outward_B_E = outward_spiral(theta_vals_B_E, b)
x_outward_B_E = r_outward_B_E * np.cos(theta_vals_B_E + np.pi)
y_outward_B_E = r_outward_B_E * np.sin(theta_vals_B_E)

# 遍历圆1的半径
for radius_circle1 in np.linspace(2, 4.5, 1000):
radius_circle2 = radius_circle1 / 2
normal_angle = tangent_angle + np.pi / 2
x_center_circle1 = x_inward_moved + radius_circle1 * np.cos(normal_angle)
y_center_circle1 = y_inward_moved + radius_circle1 * np.sin(normal_angle)

circle2_centers = []
tangent_points = []

for angle_shift in np.linspace(0, 2 * np.pi, 1000):
x_center_circle2 = x_center_circle1 + (radius_circle1 + radius_circle2) * np.cos(angle_shift)
y_center_circle2 = y_center_circle1 + (radius_circle1 + radius_circle2) * np.sin(angle_shift)

if np.sqrt(x_center_circle2**2 + y_center_circle2**2) > R:
continue

distances = np.sqrt((x_outward_B_E - x_center_circle2)**2 + (y_outward_B_E -
    y_center_circle2)**2)
min_distance = np.min(distances)

cut_x, cut_y = x_outward_B_E[np.argmin(distances)], y_outward_B_E[np.argmin(distances)]
cut_distance_to_origin = np.sqrt(cut_x**2 + cut_y**2)
center_distance_to_origin = np.sqrt(x_center_circle2**2 + y_center_circle2**2)

if center_distance_to_origin < cut_distance_to_origin and np.abs(min_distance -
    radius_circle2) < 1e-6:
x_tangent, y_tangent = calculate_tangent_point(
x_center_circle1, y_center_circle1,
x_center_circle2, y_center_circle2,
radius_circle1, radius_circle2
)
circle2_centers.append((x_center_circle2, y_center_circle2))
tangent_points.append((x_tangent, y_tangent))
print(f"  找到符合条件的圆2 (move_angle = {move_angle:.4f}, 半径: {radius_circle1:.2f}),
```

```
    圆心: ({x_center_circle2:.4f}, {y_center_circle2:.4f})")
print(f"  圆1与圆2的切点坐标: ({x_tangent:.4f}, {y_tangent:.4f})")

# 计算圆1的弧长（顺时针）
arc_length_circle1 = calculate_arc_length(x_inward_moved, y_inward_moved, x_tangent,
    y_tangent,
radius_circle1, clockwise=True)

# 计算圆2的弧长（逆时针）
arc_length_circle2 = calculate_arc_length(x_tangent, y_tangent, cut_x, cut_y, radius_circle2,
clockwise=False)
# 总弧长
arc_length = arc_length_circle1 + arc_length_circle2

# 计算盘出螺线上的极坐标角度
spiral_angle = calculate_spiral_angle(cut_x, cut_y, b)

# 保存结果到字典中
results.append({
   "in1_theta": theta_inward_moved,#圆1与盘入螺线的切点角度
   "in1_x": x_inward_moved,# 圆1与盘入螺线的切点坐标
   "in1_y": y_inward_moved,
   "point_1_x": x_center_circle1,
   "point_1_y": y_center_circle1,# 圆1圆心坐标
   "r_1": radius_circle1,# 圆1半径
   "point_2_x": x_center_circle2,
   "point_2_y": y_center_circle2,# 圆2圆心坐标
   "r_2": radius_circle2,# 圆2半径
   "point_1_2_x": x_tangent,# 圆1与圆2的切点坐标
   "point_1_2_y": y_tangent,
   "out2_x": cut_x,# 圆2与盘出
   "out2_y": cut_y,
   "cut_point_angle": spiral_angle,
   "arc_length_circle1": arc_length_circle1,
   "arc_length_circle2": arc_length_circle2,
   "arc_length": arc_length
})

# 将结果保存到Excel
df = pd.DataFrame(results)
df.to_excel("output_new.xlsx", index=False)
print("结果已保存到output_new.xlsx文件中")

if __name__ == "__main__":
main()
```

附录 E　问题四位置及速度代码

```
import pandas as pd
import math
import numpy as np
from scipy.integrate import quad
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

# 参数设置
spiral_pitch = 1.7 # 螺距

time_step = 1 # 时间步长，单位：秒
v = 1 # 龙头的速度m/s
head_length = 2.86 # 龙头两个把手的长度，单位：米
bench_length = 1.65 # 龙身两个把手的长度，单位：米（165 cm）


###############速度####################
# 计算三角形的三个角度，给定三边 a，b，c。
def calculate_angles(a, b, c):

# 计算角度的余弦值
cos_alpha = (b ** 2 + c ** 2 - a ** 2) / (2 * b * c)
cos_beta = (a ** 2 + c ** 2 - b ** 2) / (2 * a * c)

# 计算角度（弧度）
alpha = np.arccos(cos_alpha)
beta = np.arccos(cos_beta)

return beta,alpha


def calculate_tangent_angle(x0, y0, x1, y1):
"""
计算圆在点 (x1, y1) 处的切线的倾斜角.

参数:
x0, y0: 圆心的坐标
x1, y1: 切线点的坐标

返回:
切线的倾斜角（弧度）
"""
# 计算切线的斜率
```

```
if y1 != y0:
slope = -(x1 - x0) / (y1 - y0)
else:
slope = float('inf') # 如果垂直于 y 轴

# 计算倾斜角
if slope != float('inf'):
angle = math.atan(slope)
else:
angle = math.pi / 2 # 垂直时的倾斜角为 /2

return angle

# 计算螺旋线在给定角度theta处的切线斜率tan 的角度
def compute_tangent_angle(theta, a, b):
# 计算当前半径
r = a + b * theta

# 计算 dx/d 和 dy/d
dx_dtheta = b * np.cos(theta) - r * np.sin(theta)
dy_dtheta = b * np.sin(theta) + r * np.cos(theta)

# 计算切线斜率
slope = dy_dtheta / dx_dtheta

# 计算角度
angle = np.arctan(slope)

return angle
########################
def distance(x1, y1, x2, y2):
"""计算两点之间的距离"""
return np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)


def circle_equation(x, y, x_center, y_center, r):
"""圆的方程 (x - x_center)^2 + (y - y_center)^2 = r^2"""
return (x - x_center) ** 2 + (y - y_center) ** 2 - r ** 2


def find_angle(x, y, x_center, y_center):
"""计算从圆心到点的角度"""
return np.arctan2(y - y_center, x - x_center)


def solve_point_on_arc(x4, y4, l, x3, y3, r, theta_start, theta_end):
"""求解点E (x5, y5), 使其在圆弧上并与已知点D距离为l"""
```

```
def equations(vars):
x5, y5 = vars
eq1 = distance(x4, y4, x5, y5) - l # DE距离约束
eq2 = circle_equation(x5, y5, x3, y3, r) # E点在圆上
return [eq1, eq2]

# 初始猜测点在弧中间的角度
theta_mid = (theta_start + theta_end) / 2
x_guess = x3 + r * np.cos(theta_mid)
y_guess = y3 + r * np.sin(theta_mid)

# 使用fsolve求解第一个解
solution_1 = fsolve(equations, [x_guess, y_guess])

# 使用不同的初始猜测来求解第二个解（在弧的另一边）
theta_guess_alternate = theta_mid + np.pi # 猜测另一个解
x_guess_alternate = x3 + r * np.cos(theta_guess_alternate)
y_guess_alternate = y3 + r * np.sin(theta_guess_alternate)

# 使用fsolve求解第二个解
solution_2 = fsolve(equations, [x_guess_alternate, y_guess_alternate])

# 返回x坐标较小的解
if solution_1[0] < solution_2[0]:
return solution_1
else:
return solution_2


def find_point_on_arc2(x1, y1, x2, y2, x3, y3, x4, y4, l):
"""
计算点E的坐标，使得E在圆弧上，且与D的距离为l.

参数:
- x1, y1: 圆弧起始点A
- x2, y2: 圆弧终止点B
- x3, y3: 圆心C
- x4, y4: 已知点D
- l: 已知点D到E的距离

返回:
- 点E的坐标 (x5, y5)
"""
# 计算半径
r = distance(x1, y1, x3, y3)
```

```
# 计算A和B的角度
theta_start = find_angle(x1, y1, x3, y3)
theta_end = find_angle(x2, y2, x3, y3)

# 确定解
result = solve_point_on_arc(x4, y4, l, x3, y3, r, theta_start, theta_end)

x5, y5 = result
return x5, y5
##################################################
def calculate_position1(x1, y1, x2, y2, r, v, t):
theta_0 = np.arctan2(y2 - y1, x2 - x1)
omega = v / r
theta = theta_0 - omega * t # 顺时针旋转角度减少
px = x1 + r * np.cos(theta)
py = y1 + r * np.sin(theta)
return px, py

def calculate_position2(x1, y1, x2, y2, r, v, t):
theta_0 = np.arctan2(y2 - y1, x2 - x1)
omega = v / r
theta = theta_0+ omega * t # 逆时针旋转角度增加
px = x1 + r * np.cos(theta)
py = y1 + r * np.sin(theta)
return px, py


def calculate_clockwise_arc(x1, y1, x2, y2, x3, y3):
"""
计算从点 A (x1, y1) 到点 B (x2, y2) 以圆心 (x3, y3) 为中心的顺时针弧度。

参数:
x1, y1: 点 A 的坐标（起始点）
x2, y2: 点 B 的坐标（终止点）
x3, y3: 圆心的坐标

返回:
顺时针弧度 theta_clockwise
"""
# 计算向量 OA 和 OB
OA_x, OA_y = x1 - x3, y1 - y3
OB_x, OB_y = x2 - x3, y2 - y3

# 计算向量长度
OA_length = math.sqrt(OA_x ** 2 + OA_y ** 2)
OB_length = math.sqrt(OB_x ** 2 + OB_y ** 2)
```

```
# 计算向量点积
dot_product = OA_x * OB_x + OA_y * OB_y

# 计算夹角弧度（无方向的较小弧度）
cos_theta = dot_product / (OA_length * OB_length)
theta = math.acos(cos_theta)

# 计算叉积
cross_product = OA_x * OB_y - OA_y * OB_x

# 如果叉积为负，则为顺时针方向，直接返回该弧度
if cross_product < 0:
theta_clockwise = theta
else:
# 否则为逆时针方向，顺时针弧度为 2 - 逆时针弧度
theta_clockwise = 2 * math.pi - theta

return theta_clockwise
############################################################
# 定义k1
def parameter_k1_circle1(v, t, R, x1, y1, x2, y2):
a = v * t / R
cos_value = math.cos(a)
k = ((cos_value * R * R) / (x1 - x2)) + x2 + y2 * ((y1 - y2) / (x1 - x2))
return k


def parameter_k1_circle2(v, t, R, x1, y1, x2, y2):
a = v * t / (R/2)
cos_value = math.cos(a)
k = ((cos_value * R * R) / (4 * (x1 - x2))) + x2 + y2 * ((y1 - y2) / (x1 - x2))
return k


# 定义k2
def parameter_k2(x1, y1, x2, y2):
k = ((y1 - y2) / (x1 - x2))
return k


# 定义a
def parameter_a(k2):
k = 1 + k2 * k2
return k


# 定义b
```

```
def parameter_b(k1, k2, x2, y2):
k = 2 * k2 * (k1 - x2) + 2 * y2
return -k


# 定义c
def parameter_c_circle1(k1, x2, R):
k = (k1 - x2) * (k1 - x2) - R * R
return k


def parameter_c_circle2(k1, x2, R):
k = (k1 - x2) * (k1 - x2) - R * R / 4
return k


# 解一元二次方程组
def solve_quadratic(a, b, c):
# 计算判别式 Δ = b^2 - 4ac
discriminant = b ** 2 - 4 * a * c

# 如果判别式为正或零，有实数解
if discriminant >= 0:
sqrt_discriminant = math.sqrt(discriminant)
x1 = (-b + sqrt_discriminant) / (2 * a)
x2 = (-b - sqrt_discriminant) / (2 * a)
return x1, x2
else:
return 0, 0
# # 判别式小于零时，返回复数解
# sqrt_discriminant = math.sqrt(-discriminant)
# real_part = -b / (2 * a)
# imaginary_part = sqrt_discriminant / (2 * a)
# return (real_part + imaginary_part * 1j, real_part - imaginary_part * 1j)



# 定义螺旋线弧长积分的被积函数
def integrand(theta, a, b):
return np.sqrt(b ** 2 + (a + b * theta) ** 2)


# 定义计算弧长的函数
def arc_length_function(theta, a, b):
arc_length, _ = quad(integrand, 0, theta, args=(a, b))
return arc_length
```

```
# 定义根据弧长L反解角度theta的函数

initial_circle = 16 # 龙头初始位置所在的圈数
# 初始位置计算（龙头在第16圈，A点处）
initial_angle = 2 * np.pi * initial_circle # 初始角度，单位：弧度


def inverse_arc_length(t, a, b, v,initial_angle):
# 计算16圈弧长（积分从0到16圈对应的角度范围）
arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))

L = arc_length - v * t

# 定义方程：弧长与L的差值
def equation(theta):
return arc_length_function(theta, a, b) - L

# 初始猜测值，可以根据实际情况调整
initial_guess = 2 * np.pi

# 使用fsolve求解方程，找到theta
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution


# 盘出曲线
# 定义根据弧长L反解角度theta的函数
def inverse_arc_length_out(t, a, b, v, initial_angle):
# 计算初始弧长（积分从0到初始角度）
initial_arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))

# 根据时间计算龙头的弧长（总弧长）
L = initial_arc_length + v * t

# 定义方程：弧长与L的差值
def equation(theta):
return arc_length_function(theta, a, b) - L

# 初始猜测值，可以根据实际情况调整
initial_guess = 2 * np.pi

# 使用fsolve求解方程，找到theta
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution
```

```
# 根据给定的角度theta计算螺旋线上的x和y坐标。
def compute_coordinates(theta, a, b):
r = a + b * theta # 计算当前半径
x = r * np.cos(theta) # 计算x坐标
y = r * np.sin(theta) # 计算y坐标
return x, y

def angle_from_center(x, y, x_center, y_center):
# 计算点相对于圆心的极角（弧度）
return math.atan2(y - y_center, x - x_center)

############################################################
def point_on_circle(x_center, y_center, radius, angle):
"""
根据圆心、半径和角度，计算圆上的点的坐标。
"""
x = x_center + radius * math.cos(angle)
y = y_center + radius * math.sin(angle)
return x, y


def find_point_on_arc(x_center1, y_center1, radius1,x_center2, y_center2, radius2, x_start1,
    y_start1,x_start2, y_start2, x_end1, y_end1,x_end2, y_end2, x1, y1,
    length,next_class,previous_theta,theta_inward_moved,cut_point_angle):

"""
找到圆弧上的点，该点到已知点1 (x1, y1) 的距离为 length。

参数:
x_center, y_center: 圆心坐标
radius: 圆的半径
x_start, y_start: 圆弧的起点坐标
x_end, y_end: 圆弧的终点坐标
x1, y1: 已知点1的坐标
length: 圆弧上的点到已知点的距离

(x_center1, y_center1, radius1,
x_center2, y_center2, radius2,
x_start1, y_start1,
x_start2, y_start2,
x_end1, y_end1,
x_end2, y_end2,
x1, y1, length)

返回:
圆弧上与点1的距离为 length 的点的坐标 (x_result, y_result)
"""
```

```
if next_class==2:
# 计算起点和终点的极角
start_angle = math.atan2(y_start1 - y_center1, x_start1 - x_center1)
end_angle = math.atan2(y_end1 - y_center1, x_end1 - x_center1)

# 计算顺时针方向的弧长
arc_length_total = calculate_clockwise_arc(x_start1, y_start1, x_end1, y_end1, x_center1,
    y_center1)

# 进行搜索，找到与点1距离为 length 的点
for i in range(1000):

x_circle, y_circle = find_point_on_arc2(x_start1, y_start1, x_end1, y_end1, x_center1,
    y_center1, x1, y1, length)
# # 迭代查找不同的角度位置
# angle = start_angle - (i / 1000.0) * arc_length_total # 按照顺时针方向划分弧度
#
# # 计算圆弧上的点
# x_circle, y_circle = point_on_circle(x_center1, y_center1, radius1, angle)
#
# # 计算圆弧上的点到已知点1的距离
# distance = math.sqrt((x_circle - x1) ** 2 + (y_circle - y1) ** 2)
#
# # 如果距离接近指定的长度，返回该点
# if abs(distance - length) < 5e-2:
#     print('next_class==2')
#     return x_circle, y_circle, theta_inward_moved

return x_circle, y_circle, theta_inward_moved
# return 0,0,0 # 如果没有找到合适的点，返回 None
elif next_class == 3:

x_circle, y_circle = find_point_on_arc2(x_start2, y_start2, x_end2, y_end2, x_center2,
    y_center2, x1, y1,
length)
return x_circle, y_circle, cut_point_angle

# # 计算起点和终点的极角
# start_angle = math.atan2(y_start2 - y_center2, x_start2 - x_center2)
# end_angle = math.atan2(y_end2 - y_center2, x_end2 - x_center2)
#
# # 计算ni时针方向的弧长
# arc_length_total = np.pi*2-calculate_clockwise_arc(x_start2, y_start2, x_end2, y_end2,
    x_center2, y_center2)
#
# # 进行搜索，找到与点1距离为 length 的点
# for i in range(1000):
```

```
#     # 迭代查找不同的角度位置
#     angle = start_angle - (i / 1000.0) * arc_length_total # 按照ni时针方向划分弧度
#
#     # 计算圆弧上的点
#     x_circle, y_circle = point_on_circle(x_center2, y_center2, radius2, angle)
#
#     # 计算圆弧上的点到已知点1的距离
#     distance = math.sqrt((x_circle - x1) ** 2 + (y_circle - y1) ** 2)
#
#     # 如果距离接近指定的长度，返回该点
#     if abs(distance - length) < 1e-2:
#         print('next_class==3')
#         return x_circle, y_circle,cut_point_angle
#
# return 0,0 ,0 # 如果没有找到合适的点，返回 None
elif next_class == 1:
# 计算b
b = spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = 0
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - x1) ** 2 + (y - y1) ** 2)

return distance - length

# 初始猜测值为前一个把手的角度加上一个小的增量
initial_guess = previous_theta + 0.01

# 求解下一个把手的位置，确保角度递增
next_theta = fsolve(find_next_position, initial_guess)[0]

# 确保角度差值不超过2
if next_theta - previous_theta > 2 * np.pi:
next_theta = previous_theta + 2 * np.pi
print('error')

# 确保角度递增
if next_theta < previous_theta:
print('error')

# 当前把手的xy和r
next_x, next_y = compute_coordinates(next_theta, a, b)
return next_x, next_y , next_theta# 如果没有找到合适的点，返回 None
elif next_class == 4:
# 计算b
```

```
b = -spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = 0
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - x1) ** 2 + (y - y1) ** 2)

return distance - length

# 初始猜测值为前一个把手的角度加上一个小的增量
initial_guess = previous_theta - 0.01

# 求解下一个把手的位置，确保角度递增
next_theta = fsolve(find_next_position, initial_guess)[0]

# 当前把手的xy和r
next_x, next_y = compute_coordinates(next_theta, a, b)
return next_x, next_y , next_theta# 如果没有找到合适的点，返回 None
else:
return 0,0,0


# 判断下一个点在哪一段
def judge(p1,p2,t1,t2,t3,t4,t5,t6,f,m):
#p1,p2为点坐标，t1-t2为第一个连接点 t3-t4为第二个连接点 t5-t6为第三个连接点,f为所在类型 分为
    1，2，3，4 m为判断距离
h1=(p1-t1)**2+(p2-t2)**2
h2=(p1-t3)**2+(p2-t4)**2
h3=(p1-t5)**2+(p2-t6)**2

h = min(h1,h2,h3)
if f == 1 :
return 1
elif h == h1 and f == 2 and h < m:
return 1
elif h == h1 and f == 2 and h >= m:
return 2
elif h == h2 and f == 2 :
return 2
elif h == h2 and f == 3 and h < m:
return 2
elif h == h2 and f == 3 and h >= m:
return 3
elif h == h3 and f == 3 :
return 3
elif f == 4 and h < m:
```

```
return 3
elif f == 4 and h >= m:
return 4
# else:
#    # print('not class')

#################################################################
# 主函数
def main():
# 读取 Excel 文件
file_path = "output_new.xlsx" # 替换为你的文件路径
df = pd.read_excel(file_path)

# 提取第一行数据
first_row = df.iloc[2] # 提取第一行

# 将第一行的各个数据赋值给变量
theta_inward_moved = first_row['in1_theta']
x_inward_moved = first_row['in1_x']
y_inward_moved = first_row['in1_y']
x_center_circle1 = first_row['point_1_x']
y_center_circle1 = first_row['point_1_y']
radius_circle1 = first_row['r_1']
x_center_circle2 = first_row['point_2_x']
y_center_circle2 = first_row['point_2_y']
radius_circle2 = first_row['r_2']
x_tangent = first_row['point_1_2_x']
y_tangent = first_row['point_1_2_y']
cut_x = first_row['out2_x']
cut_y = first_row['out2_y']
cut_point_angle=first_row['cut_point_angle']
arc_length_circle1 = calculate_clockwise_arc(x_inward_moved, y_inward_moved, x_tangent,
    y_tangent, x_center_circle1, y_center_circle1)*radius_circle1
arc_length_circle2 = (2*np.pi-calculate_clockwise_arc(x_tangent, y_tangent, cut_x, cut_y,
    x_center_circle2, y_center_circle2))*radius_circle2
arc_length = arc_length_circle1+arc_length_circle2

# 初始化数据存储
positions_inward_x = []
positions_inward_y = []
positions_outward_x = []
positions_outward_y = []


# 初始化数据存储
positions = []
```

```
# 初始化结果字典
results = {
   'Time (s)': [],
   'Head x (m)': [],
   'Head y (m)': []
}


results_v = {
   'Time (s)': [],
   'v (m/s)': []
}
# 计算 t 从 -100 到 100
for t in np.arange(-100, 101, 1):
# 在圆弧1
if 0 < v * t < arc_length_circle1:

solve_x, solve_y=calculate_position1(x_center_circle1, y_center_circle1, x_inward_moved,
    y_inward_moved, radius_circle1, v, t)

head_theta=theta_inward_moved

head_class=2


# 在圆弧2
elif arc_length_circle1 <= v * t <= arc_length:

t_2 = (t - arc_length_circle1)

solve_x, solve_y = calculate_position2(x_center_circle2, y_center_circle2, x_tangent,
    y_tangent, radius_circle2, v, t_2)

head_class = 3
head_theta = cut_point_angle



# 盘入螺线
elif -101 < v*t <= 0:

initial_radius = initial_circle * spiral_pitch # 初始半径计算

# 计算b
b = spiral_pitch / (2 * np.pi)

# 计算初始半径a
```

```
a = initial_radius - b * initial_angle

# 计算龙头的角度
head_theta = inverse_arc_length(t, a, b, v,theta_inward_moved)

# 计算龙头的xy坐标
solve_x, solve_y = compute_coordinates(head_theta, a, b)
head_class = 1

# 计算龙头的r
head_radius = a + b * head_theta

positions_inward_x.append(solve_x)
positions_inward_y.append(solve_y)

# 盘出螺线
elif arc_length <= v*t < 101:

a = 0 # 起始半径为0（盘出螺线从内向外）
b = -spiral_pitch / (2 * np.pi) # 螺距
head_theta = inverse_arc_length_out(t-arc_length, a, b, v,-cut_point_angle)
solve_x, solve_y = compute_coordinates(head_theta, a, b)
head_class = 4

positions_outward_x.append(solve_x)
positions_outward_y.append(solve_y)

# 计算龙身和龙尾各把手的位置

positions_for_t = []



previous_theta = head_theta
previous_x, previous_y = solve_x, solve_y
#previous_radius = head_radius
previous_v = v
previous_class=head_class
# print(head_class)

# 保存时间和龙头坐标
results['Time (s)'].append(t)
results['Head x (m)'].append(solve_x)
results['Head y (m)'].append(solve_y)

# 保存时间和龙头速度
results_v['Time (s)'].append(t)
```

```
results_v['v (m/s)'].append(v)

for i in range(1, 224): # 223 节: 1个龙头 + 221个龙身 + 1个龙尾

# 使用优化方法找到下一个把手的位置
if i == 1:
length = head_length
else:
length = bench_length

#下一个把手的类型
next_class = judge(previous_x, previous_y, x_inward_moved, y_inward_moved, x_tangent,
    y_tangent, cut_x, cut_y,
previous_class,length)
# 把手的xy
next_x, next_y ,next_theta= find_point_on_arc(x_center_circle1, y_center_circle1,
    radius_circle1,
x_center_circle2, y_center_circle2, radius_circle2,
x_inward_moved, y_inward_moved,
x_tangent, y_tangent,
x_tangent, y_tangent,
cut_x, cut_y,
previous_x, previous_y,length,next_class,
previous_theta,
theta_inward_moved,cut_point_angle)

# print(1, next_x, next_y)
# print(2, previous_x, previous_y)

# if next_x==0 and next_y==0 and next_theta==0:
#    print (t,i,'',previous_class,'finderror',next_class)
# else:
#    print(t, i,'',previous_class, 'find', next_class)

previous_distance=(previous_x*previous_x+previous_y**2)**0.5

next_distance = (next_x*next_x+next_y**2)**0.5
# print(3,((next_x-previous_x)**2+(next_y-previous_y)**2)**0.5)


if next_class==2 or next_class==3:
length_ab=distance(previous_x, previous_y,next_x, next_y)
else:
length_ab =length

alpha_degrees, beta_degrees = calculate_angles(previous_distance, next_distance, length_ab)
```

```
if previous_class==1 or previous_class==4 :
tan_angle_previous = compute_tangent_angle(previous_theta, a, b)
elif previous_class==2 :
tan_angle_previous = calculate_tangent_angle(x_center_circle1, y_center_circle1, previous_x,
    previous_y)
elif previous_class==3 :
tan_angle_previous = calculate_tangent_angle(x_center_circle2, y_center_circle2, previous_x,
    previous_y)

if previous_class == 1 or previous_class == 4:
tan_angle_next =compute_tangent_angle(next_theta, a, b)
elif previous_class == 2:
tan_angle_next = calculate_tangent_angle(x_center_circle1, y_center_circle1, next_x, next_y)
elif previous_class == 3:
tan_angle_next = calculate_tangent_angle(x_center_circle2, y_center_circle2, next_x, next_y)



angle_previous = angle_from_center(previous_x, previous_y, 0, 0) - tan_angle_previous -
    alpha_degrees
angle_next = angle_from_center(next_x, next_y, 0, 0) - tan_angle_next - beta_degrees
cos_previous = np.cos(angle_previous)
cos_next = np.cos(angle_next)
next_v = abs((previous_v * cos_previous) / cos_next)

# 保存每个把手的位置
if f'Segment {i} x (m)' not in results:
results[f'Segment {i} x (m)'] = []
results[f'Segment {i} y (m)'] = []

results[f'Segment {i} x (m)'].append(next_x)
results[f'Segment {i} y (m)'].append(next_y)

# 保存每个把手的速度
if f'Segment {i} v (m/s)' not in results_v:
results_v[f'Segment {i} v (m/s)'] = []

results_v[f'Segment {i} v (m/s)'].append(next_v)

# 更新前一个把手的位置和速度
# 存储每个把手的位置
positions_for_t.append([t, next_x, next_y])
previous_theta = next_theta
xx=next_x
xxx=next_y
previous_x= xx
previous_y= xxx
```

```
    # previous_radius = next_radius
    previous_v = next_v
    previous_class = next_class

    # 存储224个点的坐标
    positions.append(positions_for_t)

    filename = "Q4/test_positions.xlsx"
    filename_v = "Q4/test_v.xlsx"

    # 将结果保存到 Excel 文件
    df = pd.DataFrame(results)
    df.to_excel(filename, index=False)
    print(f"每个把手的位置已保存到 '{filename}'.")


    # 将v保存到另一个 Excel 文件
    df_v = pd.DataFrame(results_v)
    df_v.to_excel(filename_v, index=False)
    print(f"每个把手的v已保存到 '{filename_v}'.")

    if __name__ == "__main__":
    main()
```

附录 F　问题五代码

```
import pandas as pd
import math
import numpy as np
from scipy.integrate import quad
from scipy.optimize import fsolve
import matplotlib.pyplot as plt

# 参数设置
spiral_pitch = 1.7 # 螺距

time_step = 1 # 时间步长，单位：秒
head_length = 2.86 # 龙头两个把手的长度，单位：米
bench_length = 1.65 # 龙身两个把手的长度，单位：米（165 cm）


################速度####################
# 计算三角形的三个角度，给定三边 a，b，c。
def calculate_angles(a, b, c):
```

```
# 计算角度的余弦值
cos_alpha = (b ** 2 + c ** 2 - a ** 2) / (2 * b * c)
cos_beta = (a ** 2 + c ** 2 - b ** 2) / (2 * a * c)

# 计算角度（弧度）
alpha = np.arccos(cos_alpha)
beta = np.arccos(cos_beta)

return beta,alpha


def calculate_tangent_angle(x0, y0, x1, y1):
"""
计算圆在点 (x1, y1) 处的切线的倾斜角.

参数:
x0, y0: 圆心的坐标
x1, y1: 切线点的坐标

返回:
切线的倾斜角（弧度）
"""
# 计算切线的斜率
if y1 != y0:
slope = -(x1 - x0) / (y1 - y0)
else:
slope = float('inf') # 如果垂直于 y 轴

# 计算倾斜角
if slope != float('inf'):
angle = math.atan(slope)
else:
angle = math.pi / 2 # 垂直时的倾斜角为 /2

return angle

# 计算螺旋线在给定角度theta处的切线斜率tan 的角度
def compute_tangent_angle(theta, a, b):
# 计算当前半径
r = a + b * theta

# 计算 dx/d 和 dy/d
dx_dtheta = b * np.cos(theta) - r * np.sin(theta)
dy_dtheta = b * np.sin(theta) + r * np.cos(theta)

# 计算切线斜率
slope = dy_dtheta / dx_dtheta
```

```
# 计算角度
angle = np.arctan(slope)

return angle
######################
def distance(x1, y1, x2, y2):
"""计算两点之间的距离"""
return np.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)


def circle_equation(x, y, x_center, y_center, r):
"""圆的方程 (x - x_center)^2 + (y - y_center)^2 = r^2"""
return (x - x_center) ** 2 + (y - y_center) ** 2 - r ** 2


def find_angle(x, y, x_center, y_center):
"""计算从圆心到点的角度"""
return np.arctan2(y - y_center, x - x_center)


def solve_point_on_arc(x4, y4, l, x3, y3, r, theta_start, theta_end):
"""求解点E (x5, y5)，使其在圆弧上并与已知点D距离为l"""

def equations(vars):
x5, y5 = vars
eq1 = distance(x4, y4, x5, y5) - l # DE距离约束
eq2 = circle_equation(x5, y5, x3, y3, r) # E点在圆上
return [eq1, eq2]

# 初始猜测点在弧中间的角度
theta_mid = (theta_start + theta_end) / 2
x_guess = x3 + r * np.cos(theta_mid)
y_guess = y3 + r * np.sin(theta_mid)

# 使用fsolve求解第一个解
solution_1 = fsolve(equations, [x_guess, y_guess])

# 使用不同的初始猜测来求解第二个解（在弧的另一边）
theta_guess_alternate = theta_mid + np.pi # 猜测另一个解
x_guess_alternate = x3 + r * np.cos(theta_guess_alternate)
y_guess_alternate = y3 + r * np.sin(theta_guess_alternate)

# 使用fsolve求解第二个解
solution_2 = fsolve(equations, [x_guess_alternate, y_guess_alternate])

# 返回x坐标较小的解
```

```
if solution_1[0] < solution_2[0]:
return solution_1
else:
return solution_2


def find_point_on_arc2(x1, y1, x2, y2, x3, y3, x4, y4, l):
"""
计算点E的坐标，使得E在圆弧上，且与D的距离为l.

参数:
- x1, y1: 圆弧起始点A
- x2, y2: 圆弧终止点B
- x3, y3: 圆心C
- x4, y4: 已知点D
- l: 已知点D到E的距离

返回:
- 点E的坐标 (x5, y5)
"""
# 计算半径
r = distance(x1, y1, x3, y3)

# 计算A和B的角度
theta_start = find_angle(x1, y1, x3, y3)
theta_end = find_angle(x2, y2, x3, y3)

# 确定解
result = solve_point_on_arc(x4, y4, l, x3, y3, r, theta_start, theta_end)

x5, y5 = result
return x5, y5
#######################################################
def calculate_position1(x1, y1, x2, y2, r, v, t):
theta_0 = np.arctan2(y2 - y1, x2 - x1)
omega = v / r
theta = theta_0 - omega * t # 顺时针旋转角度减少
px = x1 + r * np.cos(theta)
py = y1 + r * np.sin(theta)
return px, py

def calculate_position2(x1, y1, x2, y2, r, v, t):
theta_0 = np.arctan2(y2 - y1, x2 - x1)
omega = v / r
theta = theta_0+ omega * t # 逆时针旋转角度增加
px = x1 + r * np.cos(theta)
py = y1 + r * np.sin(theta)
```

```
return px, py


def calculate_clockwise_arc(x1, y1, x2, y2, x3, y3):
"""
计算从点 A (x1, y1) 到点 B (x2, y2) 以圆心 (x3, y3) 为中心的顺时针弧度。

参数:
x1, y1: 点 A 的坐标（起始点）
x2, y2: 点 B 的坐标（终止点）
x3, y3: 圆心的坐标

返回:
顺时针弧度 theta_clockwise
"""
# 计算向量 OA 和 OB
OA_x, OA_y = x1 - x3, y1 - y3
OB_x, OB_y = x2 - x3, y2 - y3

# 计算向量长度
OA_length = math.sqrt(OA_x ** 2 + OA_y ** 2)
OB_length = math.sqrt(OB_x ** 2 + OB_y ** 2)

# 计算向量点积
dot_product = OA_x * OB_x + OA_y * OB_y

# 计算夹角弧度（无方向的较小弧度）
cos_theta = dot_product / (OA_length * OB_length)
theta = math.acos(cos_theta)

# 计算叉积
cross_product = OA_x * OB_y - OA_y * OB_x

# 如果叉积为负，则为顺时针方向，直接返回该弧度
if cross_product < 0:
theta_clockwise = theta
else:
# 否则为逆时针方向，顺时针弧度为 2 - 逆时针弧度
theta_clockwise = 2 * math.pi - theta

return theta_clockwise
############################################################
# 定义k1
def parameter_k1_circle1(v, t, R, x1, y1, x2, y2):
a = v * t / R
cos_value = math.cos(a)
k = ((cos_value * R * R) / (x1 - x2)) + x2 + y2 * ((y1 - y2) / (x1 - x2))
```

```
return k


def parameter_k1_circle2(v, t, R, x1, y1, x2, y2):
a = v * t / (R/2)
cos_value = math.cos(a)
k = ((cos_value * R * R) / (4 * (x1 - x2))) + x2 + y2 * ((y1 - y2) / (x1 - x2))
return k


# 定义k2
def parameter_k2(x1, y1, x2, y2):
k = ((y1 - y2) / (x1 - x2))
return k


# 定义a
def parameter_a(k2):
k = 1 + k2 * k2
return k


# 定义b
def parameter_b(k1, k2, x2, y2):
k = 2 * k2 * (k1 - x2) + 2 * y2
return -k


# 定义c
def parameter_c_circle1(k1, x2, R):
k = (k1 - x2) * (k1 - x2) - R * R
return k


def parameter_c_circle2(k1, x2, R):
k = (k1 - x2) * (k1 - x2) - R * R / 4
return k


# 解一元二次方程组
def solve_quadratic(a, b, c):
# 计算判别式 Δ = b^2 - 4ac
discriminant = b ** 2 - 4 * a * c

# 如果判别式为正或零，有实数解
if discriminant >= 0:
sqrt_discriminant = math.sqrt(discriminant)
```

```
x1 = (-b + sqrt_discriminant) / (2 * a)
x2 = (-b - sqrt_discriminant) / (2 * a)
return x1, x2
else:
return 0, 0
# # 判别式小于零时，返回复数解
# sqrt_discriminant = math.sqrt(-discriminant)
# real_part = -b / (2 * a)
# imaginary_part = sqrt_discriminant / (2 * a)
# return (real_part + imaginary_part * 1j, real_part - imaginary_part * 1j)



# 定义螺旋线弧长积分的被积函数
def integrand(theta, a, b):
return np.sqrt(b ** 2 + (a + b * theta) ** 2)


# 定义计算弧长的函数
def arc_length_function(theta, a, b):
arc_length, _ = quad(integrand, 0, theta, args=(a, b))
return arc_length


# 定义根据弧长L反解角度theta的函数

initial_circle = 16 # 龙头初始位置所在的圈数
# 初始位置计算（龙头在第16圈，A点处）
initial_angle = 2 * np.pi * initial_circle # 初始角度，单位：弧度


def inverse_arc_length(t, a, b, v,initial_angle):
# 计算16圈弧长（积分从0到16圈对应的角度范围）
arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))

L = arc_length - v * t

# 定义方程：弧长与L的差值
def equation(theta):
return arc_length_function(theta, a, b) - L

# 初始猜测值，可以根据实际情况调整
initial_guess = 2 * np.pi

# 使用fsolve求解方程，找到theta
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution
```

```
# 盘出曲线
# 定义根据弧长L反解角度theta的函数
def inverse_arc_length_out(t, a, b, v, initial_angle):
# 计算初始弧长（积分从0到初始角度）
initial_arc_length, _ = quad(integrand, 0, initial_angle, args=(a, b))

# 根据时间计算龙头的弧长（总弧长）
L = initial_arc_length + v * t

# 定义方程：弧长与L的差值
def equation(theta):
return arc_length_function(theta, a, b) - L

# 初始猜测值，可以根据实际情况调整
initial_guess = 2 * np.pi

# 使用fsolve求解方程，找到theta
theta_solution = fsolve(equation, initial_guess)[0]
return theta_solution


# 根据给定的角度theta计算螺旋线上的x和y坐标。
def compute_coordinates(theta, a, b):
r = a + b * theta # 计算当前半径
x = r * np.cos(theta) # 计算x坐标
y = r * np.sin(theta) # 计算y坐标
return x, y

def angle_from_center(x, y, x_center, y_center):
# 计算点相对于圆心的极角（弧度）
return math.atan2(y - y_center, x - x_center)

#############################################################
def point_on_circle(x_center, y_center, radius, angle):
"""
根据圆心、半径和角度，计算圆上的点的坐标。
"""
x = x_center + radius * math.cos(angle)
y = y_center + radius * math.sin(angle)
return x, y


def find_point_on_arc(x_center1, y_center1, radius1,x_center2, y_center2, radius2, x_start1,
    y_start1,x_start2, y_start2, x_end1, y_end1,x_end2, y_end2, x1, y1,
    length,next_class,previous_theta,theta_inward_moved,cut_point_angle):
```

```
"""
找到圆弧上的点，该点到已知点1 (x1, y1) 的距离为 length。

参数:
x_center, y_center: 圆心坐标
radius: 圆的半径
x_start, y_start: 圆弧的起点坐标
x_end, y_end: 圆弧的终点坐标
x1, y1: 已知点1的坐标
length: 圆弧上的点到已知点的距离

(x_center1, y_center1, radius1,
x_center2, y_center2, radius2,
x_start1, y_start1,
x_start2, y_start2,
x_end1, y_end1,
x_end2, y_end2,
x1, y1, length)

返回:
圆弧上与点1的距离为 length 的点的坐标 (x_result, y_result)
"""
if next_class==2:
# 计算起点和终点的极角
start_angle = math.atan2(y_start1 - y_center1, x_start1 - x_center1)
end_angle = math.atan2(y_end1 - y_center1, x_end1 - x_center1)

# 计算顺时针方向的弧长
arc_length_total = calculate_clockwise_arc(x_start1, y_start1, x_end1, y_end1, x_center1,
    y_center1)

# 进行搜索，找到与点1距离为 length 的点
for i in range(1000):

x_circle, y_circle = find_point_on_arc2(x_start1, y_start1, x_end1, y_end1, x_center1,
    y_center1, x1, y1, length)
# # 迭代查找不同的角度位置
# angle = start_angle - (i / 1000.0) * arc_length_total # 按照顺时针方向划分弧度
#
# # 计算圆弧上的点
# x_circle, y_circle = point_on_circle(x_center1, y_center1, radius1, angle)
#
# # 计算圆弧上的点到已知点1的距离
# distance = math.sqrt((x_circle - x1) ** 2 + (y_circle - y1) ** 2)
#
# # 如果距离接近指定的长度，返回该点
```

```
# if abs(distance - length) < 5e-2:
#     print('next_class==2')
#     return x_circle, y_circle, theta_inward_moved

return x_circle, y_circle, theta_inward_moved
# return 0,0,0 # 如果没有找到合适的点，返回 None
elif next_class == 3:

x_circle, y_circle = find_point_on_arc2(x_start2, y_start2, x_end2, y_end2, x_center2,
    y_center2, x1, y1,
length)
return x_circle, y_circle, cut_point_angle

# # 计算起点和终点的极角
# start_angle = math.atan2(y_start2 - y_center2, x_start2 - x_center2)
# end_angle = math.atan2(y_end2 - y_center2, x_end2 - x_center2)
#
# # 计算ni时针方向的弧长
# arc_length_total = np.pi*2-calculate_clockwise_arc(x_start2, y_start2, x_end2, y_end2,
    x_center2, y_center2)
#
# # 进行搜索，找到与点1距离为 length 的点
# for i in range(1000):
#     # 迭代查找不同的角度位置
#     angle = start_angle - (i / 1000.0) * arc_length_total # 按照ni时针方向划分弧度
#
#     # 计算圆弧上的点
#     x_circle, y_circle = point_on_circle(x_center2, y_center2, radius2, angle)
#
#     # 计算圆弧上的点到已知点1的距离
#     distance = math.sqrt((x_circle - x1) ** 2 + (y_circle - y1) ** 2)
#
#     # 如果距离接近指定的长度，返回该点
#     if abs(distance - length) < 1e-2:
#         print('next_class==3')
#         return x_circle, y_circle,cut_point_angle
#
# return 0,0 ,0 # 如果没有找到合适的点，返回 None
elif next_class == 1:
# 计算b
b = spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = 0
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - x1) ** 2 + (y - y1) ** 2)
```

```
return distance - length

# 初始猜测值为前一个把手的角度加上一个小的增量
initial_guess = previous_theta + 0.01

# 求解下一个把手的位置，确保角度递增
next_theta = fsolve(find_next_position, initial_guess)[0]

# 确保角度差值不超过2
if next_theta - previous_theta > 2 * np.pi:
next_theta = previous_theta + 2 * np.pi
print('error')

# 确保角度递增
if next_theta < previous_theta:
print('error')

# 当前把手的xy和r
next_x, next_y = compute_coordinates(next_theta, a, b)
return next_x, next_y , next_theta# 如果没有找到合适的点，返回 None
elif next_class == 4:
# 计算b
b = -spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = 0
def find_next_position(theta):
x, y = compute_coordinates(theta, a, b)
distance = np.sqrt((x - x1) ** 2 + (y - y1) ** 2)

return distance - length

# 初始猜测值为前一个把手的角度加上一个小的增量
initial_guess = previous_theta - 0.01

# 求解下一个把手的位置，确保角度递增
next_theta = fsolve(find_next_position, initial_guess)[0]

# 当前把手的xy和r
next_x, next_y = compute_coordinates(next_theta, a, b)
return next_x, next_y , next_theta# 如果没有找到合适的点，返回 None
else:
return 0,0,0


# 判断下一个点在哪一段
```

```
def judge(p1,p2,t1,t2,t3,t4,t5,t6,f,m):
#p1,p2为点坐标，t1-t2为第一个连接点 t3-t4为第二个连接点 t5-t6为第三个连接点,f为所在类型 分为
    1，2，3，4 m为判断距离
h1=(p1-t1)**2+(p2-t2)**2
h2=(p1-t3)**2+(p2-t4)**2
h3=(p1-t5)**2+(p2-t6)**2

h = min(h1,h2,h3)
if f == 1 :
return 1
elif h == h1 and f == 2 and h < m:
return 1
elif h == h1 and f == 2 and h >= m:
return 2
elif h == h2 and f == 2 :
return 2
elif h == h2 and f == 3 and h < m:
return 2
elif h == h2 and f == 3 and h >= m:
return 3
elif h == h3 and f == 3 :
return 3
elif f == 4 and h < m:
return 3
elif f == 4 and h >= m:
return 4
# else:
#    # print('not class')

#####################################################################
# 主函数
def main():
# 读取 Excel 文件
file_path = "output_new.xlsx" # 替换为你的文件路径
df = pd.read_excel(file_path)

# 提取第一行数据
first_row = df.iloc[2] # 提取第一行

# 将第一行的各个数据赋值给变量
theta_inward_moved = first_row['in1_theta']
x_inward_moved = first_row['in1_x']
y_inward_moved = first_row['in1_y']
x_center_circle1 = first_row['point_1_x']
y_center_circle1 = first_row['point_1_y']
radius_circle1 = first_row['r_1']
x_center_circle2 = first_row['point_2_x']
```

```
y_center_circle2 = first_row['point_2_y']
radius_circle2 = first_row['r_2']
x_tangent = first_row['point_1_2_x']
y_tangent = first_row['point_1_2_y']
cut_x = first_row['out2_x']
cut_y = first_row['out2_y']
cut_point_angle=first_row['cut_point_angle']
arc_length_circle1 = calculate_clockwise_arc(x_inward_moved, y_inward_moved, x_tangent,
    y_tangent, x_center_circle1, y_center_circle1)*radius_circle1
arc_length_circle2 = (2*np.pi-calculate_clockwise_arc(x_tangent, y_tangent, cut_x, cut_y,
    x_center_circle2, y_center_circle2))*radius_circle2
arc_length = arc_length_circle1+arc_length_circle2

# 初始化数据存储
positions_inward_x = []
positions_inward_y = []
positions_outward_x = []
positions_outward_y = []


# 初始化数据存储
positions = []

# 初始化结果字典
results = {
   'Time (s)': [],
   'Head x (m)': [],
   'Head y (m)': []
}


results_v = {
   'Time (s)': [],
   'v (m/s)': []
}
for v in np.arange(0, 2, 0.01):
# 计算 t 从 -100 到 100
for t in np.arange(-100, 101, 1):
# 在圆弧1
if 0 < v * t < arc_length_circle1:

solve_x, solve_y = calculate_position1(x_center_circle1, y_center_circle1, x_inward_moved,
y_inward_moved, radius_circle1, v, t)

head_theta = theta_inward_moved

head_class = 2
```

```
# 在圆弧2
elif arc_length_circle1 <= v * t <= arc_length:

t_2 = (t - arc_length_circle1)

solve_x, solve_y = calculate_position2(x_center_circle2, y_center_circle2, x_tangent,
    y_tangent,
radius_circle2, v, t_2)

head_class = 3
head_theta = cut_point_angle



# 盘入螺线
elif -101 < v * t <= 0:

initial_radius = initial_circle * spiral_pitch # 初始半径计算

# 计算b
b = spiral_pitch / (2 * np.pi)

# 计算初始半径a
a = initial_radius - b * initial_angle

# 计算龙头的角度
head_theta = inverse_arc_length(t, a, b, v, theta_inward_moved)

# 计算龙头的xy坐标
solve_x, solve_y = compute_coordinates(head_theta, a, b)
head_class = 1

# 计算龙头的r
head_radius = a + b * head_theta

positions_inward_x.append(solve_x)
positions_inward_y.append(solve_y)

# 盘出螺线
elif arc_length <= v * t < 101:

a = 0 # 起始半径为0（盘出螺线从内向外）
b = -spiral_pitch / (2 * np.pi) # 螺距
head_theta = inverse_arc_length_out(t - arc_length, a, b, v, -cut_point_angle)
solve_x, solve_y = compute_coordinates(head_theta, a, b)
```

```
head_class = 4

positions_outward_x.append(solve_x)
positions_outward_y.append(solve_y)

# 计算龙身和龙尾各把手的位置

positions_for_t = []

previous_theta = head_theta
previous_x, previous_y = solve_x, solve_y
# previous_radius = head_radius
previous_v = v
previous_class = head_class
# print(head_class)

# 保存时间和龙头坐标
results['Time (s)'].append(t)
results['Head x (m)'].append(solve_x)
results['Head y (m)'].append(solve_y)

# 保存时间和龙头速度
results_v['Time (s)'].append(t)
results_v['v (m/s)'].append(v)

for i in range(1, 224): # 223 节: 1个龙头 + 221个龙身 + 1个龙尾

# 使用优化方法找到下一个把手的位置
if i == 1:
length = head_length
else:
length = bench_length

# 下一个把手的类型
next_class = judge(previous_x, previous_y, x_inward_moved, y_inward_moved, x_tangent,
    y_tangent, cut_x,
cut_y,
previous_class, length)
# 把手的xy
next_x, next_y, next_theta = find_point_on_arc(x_center_circle1, y_center_circle1,
    radius_circle1,
x_center_circle2, y_center_circle2, radius_circle2,
x_inward_moved, y_inward_moved,
x_tangent, y_tangent,
x_tangent, y_tangent,
cut_x, cut_y,
previous_x, previous_y, length, next_class,
```

```
previous_theta,
theta_inward_moved, cut_point_angle)

# print(1, next_x, next_y)
# print(2, previous_x, previous_y)

# if next_x==0 and next_y==0 and next_theta==0:
#    print (t,i,'',previous_class,'finderror',next_class)
# else:
#    print(t, i,'',previous_class, 'find', next_class)

previous_distance = (previous_x * previous_x + previous_y ** 2) ** 0.5

next_distance = (next_x * next_x + next_y ** 2) ** 0.5
# print(3,((next_x-previous_x)**2+(next_y-previous_y)**2)**0.5)

if next_class == 2 or next_class == 3:
length_ab = distance(previous_x, previous_y, next_x, next_y)
else:
length_ab = length

alpha_degrees, beta_degrees = calculate_angles(previous_distance, next_distance, length_ab)

if previous_class == 1 or previous_class == 4:
tan_angle_previous = compute_tangent_angle(previous_theta, a, b)
elif previous_class == 2:
tan_angle_previous = calculate_tangent_angle(x_center_circle1, y_center_circle1, previous_x,
previous_y)
elif previous_class == 3:
tan_angle_previous = calculate_tangent_angle(x_center_circle2, y_center_circle2, previous_x,
previous_y)

if previous_class == 1 or previous_class == 4:
tan_angle_next = compute_tangent_angle(next_theta, a, b)
elif previous_class == 2:
tan_angle_next = calculate_tangent_angle(x_center_circle1, y_center_circle1, next_x, next_y)
elif previous_class == 3:
tan_angle_next = calculate_tangent_angle(x_center_circle2, y_center_circle2, next_x, next_y)

angle_previous = angle_from_center(previous_x, previous_y, 0, 0) - tan_angle_previous -
    alpha_degrees
angle_next = angle_from_center(next_x, next_y, 0, 0) - tan_angle_next - beta_degrees
cos_previous = np.cos(angle_previous)
cos_next = np.cos(angle_next)
next_v = abs((previous_v * cos_previous) / cos_next)

if next_v >2:
```

```
print('速度',v,'不合题意')



# 更新前一个把手的位置和速度
# 存储每个把手的位置
positions_for_t.append([t, next_x, next_y])
previous_theta = next_theta
xx = next_x
xxx = next_y
previous_x = xx
previous_y = xxx
# previous_radius = next_radius
previous_v = next_v
previous_class = next_class

# 存储224个点的坐标
positions.append(positions_for_t)



if __name__ == "__main__":
main()
```