等距螺线轨道上的链式刚体运动模型

摘要

对于问题一, 我们构建了**等距螺线轨道上的链式刚体运动模型**。首先，通过阿基米德螺线的极坐标方程描述把手的运动轨迹，并将其转换为直角坐标系。接着，基于刚体运动理论，简化了把手在两个点的运动情况，建立了相邻板之间的运动关系模型。在求解过程中，使用了两种方法计算把手位置和速度：1. 速度迭代法，优化了算法效率；2. 基于微分的微小步长法，通过逐渐减小增量计算瞬时速度，并设定了收敛阈值，当满足条件时认为计算已经收敛。最终，我们得到从初始时刻到 300 s 为止每秒整个舞龙队的位置和速度，结果如表1、表2以及附件 result1.xlsx 所示。

对于问题二, 我们构建了舞龙队盘入过程中的碰撞检测模型。首先，将板凳简化为二维平面中的矩形，使用中心点坐标和旋转角度描述每个矩形的位置和方向。通过分离轴定理（SAT）检测非相邻矩形之间的碰撞。模型求解中，首先用中心点坐标和旋转矩阵描述矩形的位置和方向，然后计算矩形边的法向量作为分离轴，检查顶点投影是否重叠。最后，通过建立运动方程和时间步进法，在离散时间步长内更新矩形的位置和角度，并使用 SAT 方法检测碰撞。最后，我们得到舞龙队盘入的终止时刻为 **412.47s**, 此时整个舞龙队的位置和速度见表3以及附件 result2.xlsx。

对于问题三, 我们构建了计算最小调头螺距的模型。首先证明了螺距与龙头前把手距离原点之间的反比关系，并确定了最小调头螺距的存在性。模型包括：使用等距螺线方程描述盘入路径，确保能进入直径 9 米的调头区域，并在 30 厘米到 55 厘米之间使用二分搜索算法确定最小螺距。在求解过程中，设定初始条件，模拟盘入过程，计算把手位置和速度，检测碰撞，并通过二分搜索算法优化螺距以满足调头空间要求。最终，我们得到满足条件的最小螺距约为 **0.450391** 米。

对于问题四, 我们建立了舞龙队调头路径优化模型。首先，分析了调整圆弧的可行性，并解释了调头区域与真实调头区域的关系。列举并讨论了各种约束条件，包括碰撞约束、最大深度碰撞约束和 S 型路径弯折约束。然后，基于这些约束对真实调度区间建模，求解了不同区间下调头曲线的长度，并找出最短调头路径。通过数值模拟验证了 1.7 米螺距下的碰撞情况，确认了无需考虑真实调度区间半径的约束。最后，基于 S 型路径弯折约束，求解了路径圆弧长度的最小值，得到优化后的调度路径几何信息。最终，我们得到最短调头曲线长度为 $l = 12.940776552$ 米，以调头开始时间为零时刻，从 −100 s 开始到 100 s 为止，每秒舞龙队的位置和速度详见表4、表5以及附件 result4.xlsx。

对于问题五, 借用第四题的模拟过程，我们可以求解出不同龙头前把手速度下，整个队伍中把手的最大速度，发现龙头前把手速度和整个队伍中把手的最大速度成**线性关系**，且整个队伍速度小于 2m/s，要求前把手速度小于 **1.7926s**。

关键字：**链式刚体运动模型　几何运动模拟　碰撞约束　二分优化　微分优化**

一、 问题重述

1.1 问题背景

随着全球文化多样性和传统文化保护意识的增强，如何将传统文化与现代技术相结合，促进其传播和发展，成为了一个亟待解决的问题。在中国，浙闽地区的“板凳龙”是一个历史悠久的民俗活动，极具地方特色。这种活动以蜿蜒曲折的“龙形”队伍为主体，由成百上千条板凳首尾相连，形成一个大规模的表演阵列。每年元宵节期间，当地居民会组织大规模的“板凳龙”活动，象征着驱邪纳福，祈求风调雨顺。

然而，如何提升板凳龙的表演效果，尤其是如何在有限的空间内，使舞龙队能够更加流畅地盘入和盘出，成为了一个技术性挑战。为了保证观赏效果，要求板凳龙的队伍占地面积尽可能小，同时队伍的行进速度尽可能快，这对板凳龙的设计、摆放以及路径规划提出了较高的要求。

针对这些问题，我们通过数学建模来对“板凳龙”的行进路线进行分析和优化，旨在为这项文化活动提供科学的依据。我们的目标是通过建立一个精准的模型，计算舞龙队伍的行进速度和位置，模拟其在螺线中的运动，并提出在空间最小化和避免碰撞的前提下，优化舞龙的路径设计，使其观赏效果最佳。

1.2 问题要求

问题 1：舞龙队在螺旋线中的运动仿真

舞龙队由 223 节板凳组成，其中第 1 节为龙头，后面 221 节为龙身，最后 1 节为龙尾，龙头的板长为 341 cm，龙身和龙尾的板长为 220 cm。龙头的行进速度保持为 1 m/s，队伍从螺旋线第 16 圈的 A 点顺时针盘入。要求模拟从初始时刻到 300 秒期间，龙头、龙身和龙尾的每节板凳把手的运动轨迹和速度，并将结果保存为 `result1.xlsx`。在论文中，还需要给出 0 秒、60 秒、120 秒、180 秒、240 秒、300 秒时刻，龙头、龙身（包括第 1、51、101、151、201 节）和龙尾的位置信息和速度。

问题 2：确定舞龙队停止盘入的时刻

在螺旋线盘入过程中，当队伍的板凳之间发生碰撞时，舞龙队必须停止继续盘入。要求确定舞龙队盘入的终止时刻，并给出此时所有关键点的位置和速度，同时将结果保存到 `result2.xlsx`。在论文中给出碰撞发生时，龙头、龙身（第 1、51、101、151、201 节）和龙尾的位置信息和速度。

问题 3：确定舞龙队的最小螺距

舞龙队在螺旋线盘入时需要在中心位置进行调头切换为逆时针盘出。调头空间是以螺旋线中心为圆心、直径为 9 米的圆形区域。要求确定舞龙队盘入至调头空间时的最小

螺距，确保龙头能够顺利到达调头空间的边界。

问题 4：优化调头路径

舞龙队的调头路径由两段圆弧相切组成的 S 形曲线，前一段圆弧的半径是后一段的 2 倍，并且该曲线与盘入、盘出螺线均相切。要求判断是否可以调整圆弧曲线的形状，仍保持相切的前提下，缩短调头曲线的总长度。以调头开始的时间为零点，模拟从-100 秒到 100 秒的运动，并记录每秒龙头、龙身和龙尾的位置信息和速度，结果保存为 `result4.xlsx`。

问题 5：确定舞龙队的最大行进速度

在调头过程中，龙头的行进速度保持不变。要求计算出龙头的最大行进速度，使得舞龙队的所有把手的速度不超过 2 m/s。

二、 问题分析

2.1 问题一分析

问题一要求我们求解从初始时刻到 300 s 为止，每秒整个舞龙队的位置和速度。首先，我们可以建立等距螺线轨道模型，使用阿基米德螺线的极坐标方程描述板凳把手的运动轨迹，并将其转换为直角坐标系表示。接着，我们可以基于刚体运动理论，简化分析把手所在的两个点的运动情况，建立了相邻板之间的运动关系模型。在模型求解方面，我们可以使用基于固定极角差的迭代法计算把手位置，通过已知的角差和前一个点的极坐标，推导出后一个点的极坐标，并将其转换为笛卡尔坐标。对于把手速度的计算，我们可以采用两种方法并相互验证：a) 基于相邻把手间的递推关系的速度迭代法：从龙头开始，使用递推公式计算每块板的速度和位置。同时，我们可以通过缓存技术、向量化计算、减少不必要的计算和动态步长调整等方式优化算法效率。b) 基于微分的微小步长法：通过逐渐减小时间或极角的增量，求解质点在螺旋线上运动时的瞬时速度。我们可以设定一个收敛阈值，当满足条件时认为计算已经收敛。

2.2 问题二分析

问题二要求我们确定舞龙队盘入的终止时刻，使得板凳之间不发生碰撞，并给出此时舞龙队的位置和速度，为此，我们可以以问题一中提出的模型为基础，为板凳加上长和宽以及增加碰撞检测。首先，我们可以将板凳简化为二维平面中的矩形，通过中心点坐标和旋转角度描述每个矩形的位置和方向。然后，我们可以采用分离轴定理（SAT）来检测非相邻矩形之间是否发生碰撞。在模型求解方面，我们可以采用以下步骤：

1. 矩形位置与旋转角度表示：使用中心点坐标和旋转矩阵来描述每个矩形的位置和方向。

2. 应用分离轴定理：计算矩形边的法向量作为分离轴，将顶点投影到这些轴上，检查投影是否重叠。
3. 建立运动方程：描述每个矩形在时间轴上的位置和角度变化。
4. 时间步进法检测碰撞：在离散的时间步长内，计算矩形的新位置和角度，并使用 SAT 方法检测碰撞。

2.3 问题三分析

问题三要求我们确定最小螺距，使得龙头前把手能够沿着相应的螺线盘入到调头空间的边界。首先，我们可以证明螺距与龙头前把手距离原点的距离之间存在反比关系，确定最小调头螺距的存在性。然后，我们可以建立以下模型：

螺线盘入模型：使用等距螺线方程描述盘入路径，螺距作为变量。调头空间约束：确保舞龙队能进入直径 9 米的圆形调头区域。二分搜索算法：在 30cm 到 55cm 之间搜索最小螺距。

在模型求解方面，我们可以采取以下步骤：

1. 初始条件设定：包括螺旋线参数、板凳参数和调头空间参数。
2. 盘入过程模拟：计算把手位置和速度，检测板凳的碰撞情况。

2.4 问题四分析

问题四要求我们调整圆弧，仍保持各部分相切，使得调头曲线变短，并给出以调头开始时间为零时刻，从 −100 s 开始到 100 s 为止，每秒舞龙队的位置和速度。首先，我们可以分析调整圆弧的可行性，阐明题给调头区域和真实调头区域之间的差异与联系。其次，我们可以列举并解释了可能存在的约束条件，包括龙头往返碰撞约束、龙头盘入最大深度碰撞约束和 S 型调头路径刚体弯折约束。然后，基于这些假设和约束，我们就可以对真实调度区间进行建模，求解不同真实调度区间下调头曲线的长度，并根据碰撞约束求解出最短调头路径。

2.5 问题五分析

问题五要求确定龙头的最大行进速度，使得舞龙队各把手的速度均不超过 2 m/s。首先，我们可以证明螺线与圆弧运动的等价性，通过比较螺线和圆弧运动的速度和加速度，找出了它们在特定条件下的等价关系。其次，我们可以证明龙头前把手与各把手速度之间存在线性关系，这是基于第一问中相邻把手间速度的递推关系式得出的结论。在模型求解过程中，我们可以采用数值方法。首先，通过解析几何计算确定不同部分的圆弧和螺线的交点及其相关信息。然后，对于头节点在不同圆弧段上的运动，我们使用数值积分计算头节点位置随时间的变化。我们还可以使用非线性方程求解器 fsolve 来确定

螺线或圆弧上点之间的角度关系。最后，通过数值差分方法计算了每个时间点下每个把手的速度。为了验证龙头前把手与各把手速度的线性关系，我们可以选取合适的步长，遍历 (0.6,2) 范围内所有龙头前把手速度下最快把手的速度，并进行线性拟合。

三、 模型假设

为简化问题，本文做出以下假设：

- 假设 1 进入题给调头空间时，并不要求立刻进行队伍调头。
- 假设 2 如果进入题给调头空间时队伍不立即调头，队伍将沿着原盘入螺线进入。
- 假设 3 调头空间内设定的盘入螺线轨道纳入全局的盘入螺线轨道定义中。
- 假设 4 调头空间内设定的盘出螺线轨道纳入全局的盘出螺线轨道定义中。
- 假设 5 根据假设 3.4，调头空间内的盘入螺线与调头空间内的盘出螺线关于圆心中心对称。
- 假设 6 舞龙队伍在行进过程中只前进不后退
- 假设 7 板凳在运动和碰撞过程中不会发生形变
- 假设 8 把手足够坚硬，两个板的连接处永远保持重合且不会断开

四、 符号说明

符号	说明	单位
θ_i	第 i 个把手此时所在的弧度值	rad
r	螺旋线上某点与螺旋线中心间的距离, 即极径	m
r_i	第 i 个把手的极径	m
r_0	螺旋线与极点间的初始距离，此题模型中值为 0	m
d	螺旋线的螺距的 $\frac{1}{2\pi}$ 倍	m
$v_i(t)$	t 时刻第 i 个把手处的速度	m/s
$v_{ij}(t)$	t 时刻第 i、j 个把手所在板的沿板中轴线的速度	m/s
$\dot{\theta}_i$	第 i 个把手的角速度	rad/s
$\Delta\theta$	两质点间的极角差	rad
x_i	第 i 个把手的横坐标	m
y_i	第 i 个把手的纵坐标	m
Δt	迭代间隔的时间步长	s

五、 问题一的模型的建立和求解

5.1 模型建立

5.1.1 等距螺线轨道建模

等距螺线，也称阿基米德螺线: 螺距（沿螺旋线方向量得的，相邻两螺纹之间的距离）固定不变的，即每一圈的半径以一个固定值增长的螺线，中心位于坐标系原点的阿基米德螺旋在极坐标系中的方程为:

$$r(\theta) = r_0 + d \cdot \theta$$

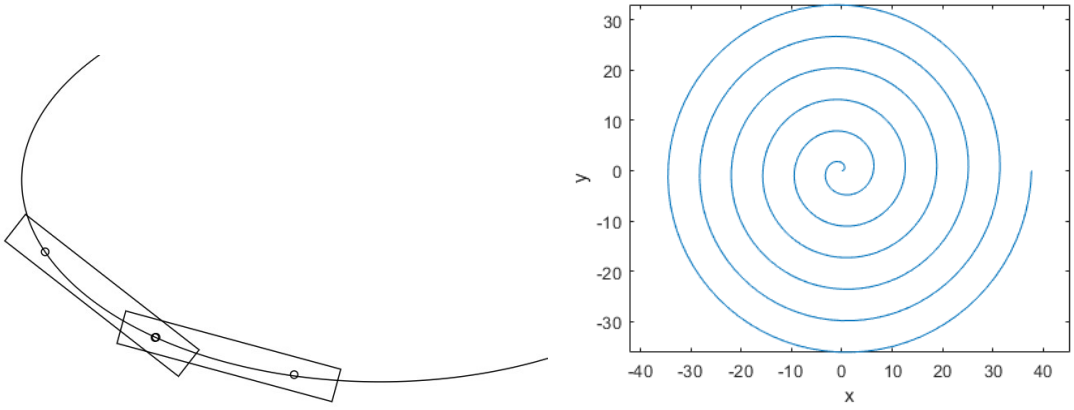
其中：r 是点到原点的距离（半径），θ 是极角（以弧度表示），d 是每一圈半径的增加量，即螺距。r_0 用于控制螺旋线与极点（原点）的初始距离，此问题中，原点与中心重合，故初始距离为 0。

为了表示和计算板凳龙把手的位置，将螺线在极坐标系中的方程转换为其在直角坐

标系中的表示

$$\begin{cases} x = (r_0 + d \cdot \theta) \cdot \cos\theta \\ y = (r_0 + d \cdot \theta) \cdot \sin\theta \end{cases}$$

由此便建立了类似下图的所有把手运动的轨道模型。



(a) 相邻板之间的运动关系　　**(b) 把手运动轨迹模型**

5.1.2 刚体运动建模

根据板凳不会发生形变的前提假设，可以将每个板凳的运动均抽象为刚体的运动，即其上任意两点间相对位置在运动过程中保持不变[1]，我们根据把手始终在路径上运动的条件，简化分析把手所在的两个点的运动情况，后续可以根据这两点的运动表征整块板的运动。

根据龙头的速度，可以计算出每一秒龙头的位置，然后根据固定的距离和轨道，算出板凳后把手的位置，在所有前后把手距离固定的情况下依次向下迭代，计算出所有把手在每一秒的位置。

单个质点运动的极坐标方程及其对应的直角坐标方程

假设质点在时间 t 时的极角为 $\theta(t)$，则质点的极坐标表示为：

$$r(t) = r_0 + d\theta(t)$$

将其转换为直角坐标：

$$\begin{cases} x(t) = (r_0 + d\theta(t))\cos(\theta(t)) \\ y(t) = (r_0 + d\theta(t))\sin(\theta(t)) \end{cases}$$

由相邻板之前牵连的位置关系递推所有速度

每对把手均由一块板相连，每个把手的速度均与螺线相切，知道把手在螺线上的位置即可求出该点的斜率，从而确定每一点的速度方向。第一个把手的大小已知，每对把手由木板相连且在每一点处速度方向和轨迹方程已知，可由关联体之间的速度关系不断

递推计算出所有把手处的速度大小。

前后质点的极径

前一个质点的极角为 $\theta_1(t)$，则其极径为：

$$r_i(t) = r_0 + d\theta_i(t)$$

后一个质点的极角为 $\theta_2(t)$，则其极径为：

$$r_{i+1}(t) = r_0 + d\theta_{i+1}(t)$$

两把手在板上的相对位置保持不变，故模型中两个质点间的距离不变，同时两个质点间的极角差 $\Delta\theta = \theta_i(t) - \theta_{i+1}(t)$ 也保持不变

龙头前把手的速度保持不变, 故模型中第一个质点的速度 v_1 恒定, 所有质点的速度方向均始终沿着螺线的切线方向

切线速度表达式

质点的速度沿着螺旋线的切线方向，速度大小可以用极径 r 和角速度 $\dot{\theta}$ 来表示：

$$v_\theta = r \cdot \dot{\theta}$$

前一个质点的速度大小为 v_i，且固定不变，因此：

$$v_i = (r_0 + d\theta_i(t)) \cdot \dot{\theta}_i(t)$$

解出 $\dot{\theta}_i(t)$：

$$\dot{\theta}_i(t) = \frac{v_i}{r_0 + d\theta_i(t)}$$

根据相对距离不变的条件得出极角和极径的递推式

两质点之间的极角差 $\Delta\theta = \theta_i(t) - \theta_{i+1}(t)$ 保持不变，因此后一个质点的极角为：

$$\theta_{i+1}(t) = \theta_i(t) - \Delta\theta$$

后一个质点的极径为：

$$r_{i+1}(t) = r_0 + d(\theta_i(t) - \Delta\theta)$$

推导后一个质点的速度

后一个质点的速度为：

$$v_{i+1} = r_{i+1}(t) \cdot \dot{\theta}_{i+1}(t)$$

由于 $\Delta\theta$ 保持不变，后一个质点的角速度 $\dot{\theta}_{i+1}(t)$ 与前一个质点的角速度相同，即：

$$\dot{\theta}_{i+1}(t) = \dot{\theta}_i(t) = \frac{v_i}{a + b\theta_i(t)}$$

因此，后一个质点的速度为：

$$v_{i+1} = (r_0 + d(\theta_i(t) - \Delta\theta)) \cdot \frac{v_i}{r_0 + d\theta_i(t)}$$

最终，后一个质点的速度递推公式为：

$$v_{i+1} = \frac{r_0 + d(\theta_i(t) - \Delta\theta)}{r_0 + d\theta_i(t)} \cdot v_i$$

微分法推导瞬时速度

由第一个把手固定的速度和每对把手间固定的距离可确定每一时刻所有把手的坐标位置，只要取到尽可能小的迭代时间间隔，即可求得一段极小距离上的平均速度，当时间间隔足够小，在误差允许的范围内，可由这段极小切片上的平均速度大小代替其在该点处的瞬时速度大小。

相邻点对的位置坐标递推式

假设前一个点的极坐标为 (r_i, θ_i)，后一个点的极坐标为 (r_{i+1}, θ_{i+1})。我们知道两点之间的极角差 $\Delta\theta = \theta_{i+1} - \theta_i$ 是保持不变的，这意味着两点之间的径向距离和角向距离都将保持不变。

在极坐标系中，两点 (r_i, θ_i) 和 (r_{i+1}, θ_{i+1}) 之间的距离 d 可以通过以下公式计算：

$$\acute{d} = \sqrt{r_i^2 + r_{i+1}^2 - 2r_i r_{i+1} \cos(\theta_{i+1} - \theta_i)}$$

距离 $\acute{d}$ 保持不变，那么这个方程可以作为一个约束条件，帮助我们推导出后一个点的位置。

推导后一个点的位置

已知前一个点的位置为 (r_1, θ_1)，并且两点之间的距离 d 保持不变。后一个点的极角可以表示为：

$$\theta_{i+1} = \theta_i + \Delta\theta$$

其中，$\Delta\theta$ 是常数。

由于螺旋线方程 $r(\theta) = a + b\theta$，后一个点的极径为：

$$r_{i+1} = r_0 + d(\theta_i + \Delta\theta)$$

因此，后一个点的极坐标为：

$$(r_{i+1}, \theta_{i+1}) = (r_0 + d(\theta_i + \Delta\theta), \theta_i + \Delta\theta)$$

转换为笛卡尔坐标 (x_2, y_2)，使用转换公式：

$$\begin{cases} x_{i+1} = (r_0 + d(\theta_i + \Delta\theta)) \cos(\theta_i + \Delta\theta) \\ y_{i+1} = (r_0 + d(\theta_i + \Delta\theta)) \sin(\theta_i + \Delta\theta) \end{cases}$$

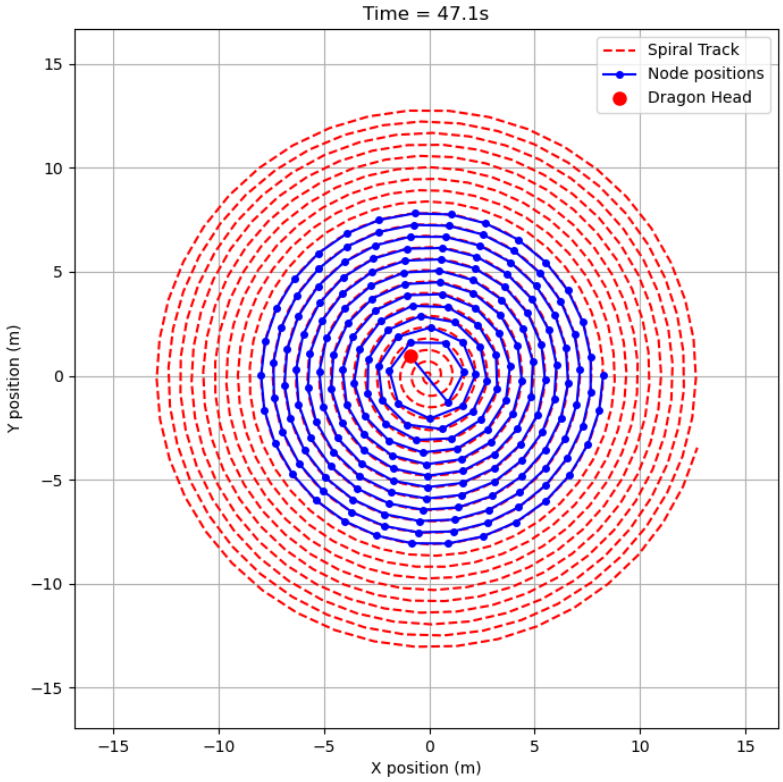


图 2　旋入过程中把手对位置的抽象图

微分处理运动过程，推导每个时刻的瞬时速度

质点的瞬时速度

质点的瞬时速度可以表示为速度向量的大小：

$$v(t) = \sqrt{\left(\frac{dr}{dt}\right)^2 + \left(r\frac{d\theta}{dt}\right)^2}$$

其中 $\frac{dr}{dt}$ 是径向速度，$r\frac{d\theta}{dt}$ 是切线速度。

对 $r(t) = r_0 + d\theta(t)$ 求导，我们有：

$$\frac{dr}{dt} = \frac{d}{dt}(r_0 + d\theta(t)) = d\frac{d\theta}{dt}$$

因此，瞬时速度的表达式为：

$$v(t) = \sqrt{\left(d\frac{d\theta}{dt}\right)^2 + \left(r(t)\frac{d\theta}{dt}\right)^2}$$

代入 $r(t) = r_0 + d\theta(t)$：

$$v(t) = \frac{d\theta}{dt} \cdot \sqrt{d^2 + (r_0 + d\theta(t))^2}$$

5.2 模型求解

5.2.1 把手位置的计算：基于固定极角差的迭代法

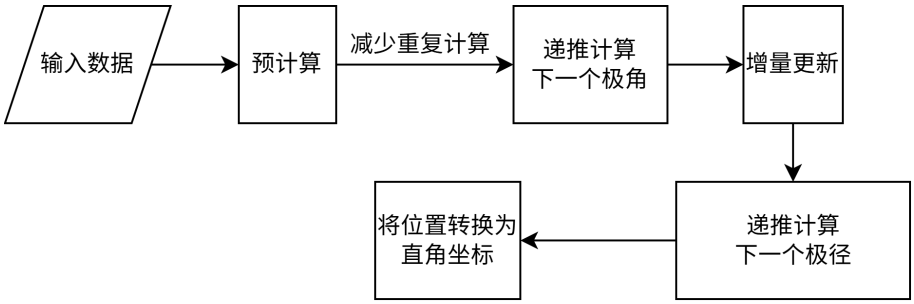


图 3　基于固定极角差的迭代法求解所有位置坐标流程

算法描述

这种方法假设两个质点沿着阿基米德螺旋线运动，且它们之间的极角差 $\Delta\theta$ 保持不变。通过前一个质点的极坐标 (r_i, θ_i) 和已知的角差 $\Delta\theta$，可以计算后一个质点的极坐标 (r_{i+1}, θ_{i+1})。

Step1：计算后一个点的极角 θ_{i+1}

使用已知的极角差 $\Delta\theta$，从前一个点的极角 θ_i 推导出后一个点的极角 θ_{i+1}。

$$\theta_{i+1} = \theta_i + \Delta\theta$$

Step2：计算后一个点的极径

根据螺旋线方程，后一个点的极径为：

$$r_{i+1} = r_0 + d\theta_{i+1} = r_0 + d(\theta_i + \Delta\theta)$$

Step3：转换为笛卡尔坐标

转换为笛卡尔坐标 (x_2, y_2)，可以使用以下公式：

$$\begin{cases} x_{i+1} = r_{i+1}\cos(\theta_{i+1}) = (r_0 + d(\theta_i + \Delta\theta))\cos(\theta_{i+1}) \\ y_{i+1} = r_{i+1}\sin(\theta_{i+1}) = (r_0 + d(\theta_i + \Delta\theta))\sin(\theta_{i+1}) \end{cases}$$

Step4：增量更新优化

为了减少计算量，可以使用增量更新法来计算后一个点的极径：

$$r_{i+1} = r_i + d\Delta\theta$$

这避免了每次重新计算螺旋线方程，只需要在前一个点的基础上进行加法操作即可得到新的极径 r_{i+1}。

Step5：预计算优化

极角差 $\Delta\theta$ 是固定值，可以预先计算 $\cos(\Delta\theta)$ 和 $\sin(\Delta\theta)$，减少实时计算的三角函数调用。

5.2.2 把手速度的计算

速度计算我们采取了两种计算速度的方式分别求出每个把手在每一秒的速度，两种方法的结果相互验证，以保证结果的准确性。

Method1: 基于相邻把手间的递推关系的速度迭代法

我们可以通过迭代递推公式从龙头开始推导出每块板的速度，并进一步优化算法的效率

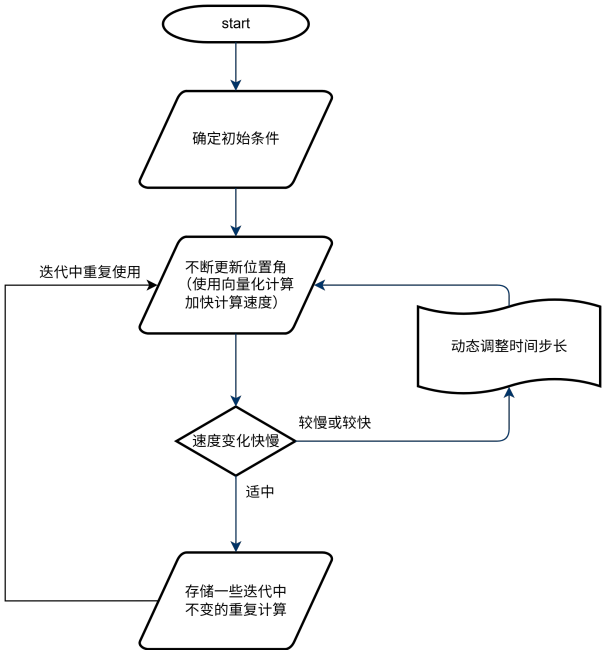


图 4　基于相邻把手间的递推关系的速度迭代法流程

Step1：确定初始条件

初始条件是迭代的起点，它直接影响整个系统的行为。对于问题中的舞龙队，我们有如下初始条件：

- 龙头初始速度为 $v_1(0) = 1\,\text{m/s}$，
- 龙头初始位置角 $\theta_1(0)$ 对应于螺线的第 16 圈，且位置为 A 点。

Step2：位置角的更新

为了计算随时间变化的位置角，我们使用下列公式更新每块板的位置角。假设第 i 块板在时间 t 时的位置角为 $\theta_i(t)$，那么在时间 $t + \Delta t$ 时它的位置角为：

$$\theta_i(t + \Delta t) = \theta_i(t) + \frac{v_i(t)}{r_i} \cdot \Delta t$$

其中：

$$r_i = r_0 + d \cdot \theta_i(t)$$

表示第 i 块板所在的螺旋线的半径。

Step3：迭代求解

- 从龙头开始，初始速度为 $v_1(0) = 1\,\text{m/s}$,
- 使用递推公式计算每块板的速度,
- 使用位置角更新公式计算每块板的位置。

迭代公式为：

$$v_{i+1}(t) = \frac{r_0 + d(\theta_i(t) - \Delta\theta)}{r_0 + d\theta_i(t)} \cdot v_i(t)$$

位置更新公式为：

$$\theta_i(t + \Delta t) = \theta_i(t) + \frac{v_i(t)}{r_i} \cdot \Delta t$$

通过不断迭代这些公式，我们可以推导出所有板的速度和位置。

Step4：算法优化

为了提升计算效率，我们进行了如下优化：

缓存技术

在每次迭代中，我们将每块板的速度和位置存储到数组中，以避免重复计算。例如，速度 $v_i(t)$ 和位置角 $\theta_i(t)$ 可以存储为向量：

$$\text{速度向量：} \quad \boldsymbol{v} = [v_1(t), v_2(t), \ldots, v_N(t)]$$

$$\text{位置角向量：} \quad \boldsymbol{\theta} = [\theta_1(t), \theta_2(t), \ldots, \theta_N(t)]$$

向量化计算

使用向量化操作代替循环操作，以加快计算速度。更新公式如下：

$$\boldsymbol{\theta}(t + \Delta t) = \boldsymbol{\theta}(t) + \frac{\boldsymbol{v}(t)}{\boldsymbol{r}} \cdot \Delta t$$

其中：

$$\boldsymbol{r} = r_0 + d \cdot \boldsymbol{\theta}(t)$$

减少不必要的计算

为了减少重复计算，我们将一些常量提前计算，例如相邻板之间的角度差 $\Delta\theta$，并在整个迭代过程中重复使用。递推公式更新为：

$$v_{i+1}(t) = \frac{r_0 + d(\theta_i(t) - \Delta\theta)}{r_0 + d \cdot \theta_i(t)} \cdot v_i(t)$$

动态步长调整

根据速度变化动态调整时间步长 Δt。当速度变化较小（即 $|v_{i+1}(t) - v_i(t)|$ 较小时），可以增大 Δt；而当速度变化较大时，缩小 Δt 以保证计算精度。

优化后迭代公式

优化后的速度迭代公式为：

$$v_{i+1}(t) = \frac{r_0 + d(\theta_i(t) - \Delta\theta)}{r_0 + d\theta_i(t)} \cdot v_i(t)$$

位置角更新公式为：

$$\theta_i(t + \Delta t) = \theta_i(t) + \frac{v_i(t)}{r_i} \cdot \Delta t$$

通过缓存和向量化操作，我们可以在每个时间步内同时更新所有板的位置和速度，从而提高计算效率。

Method2: 基于微分的微小步长法

算法描述

此算法基于减少迭代步长，通过逐渐减小时间或极角的增量，来求解质点在螺旋线上运动时的瞬时速度。此方法本质上是通过求解质点在极短时间内的位置变化，进而得到质点的速度。

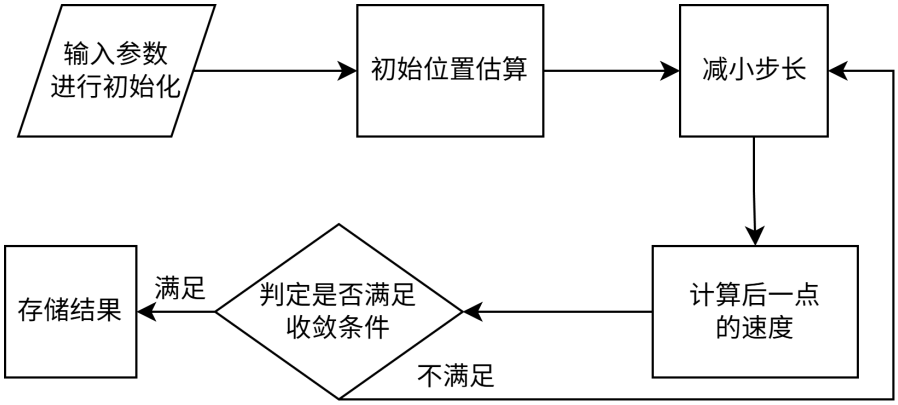


图 5　基于微分的微小步长法流程图

Step1: 输入参数初始化

输入参数：螺旋线常数参数 r_0 和 d，前一个点的位置 (r_i, θ_i)，极角差 $\Delta\theta_0$，时间步长 Δt_0，以及设定的收敛阈值 ϵ。

目标：计算质点在螺旋线上的瞬时速度 v_i，并利用该速度推导出后一个点的位置 (r_{i+1}, θ_{i+1})。

Step2: 初始位置估算

根据初始速度 v_1 和相对距离 b(板上两点间的距离)，初步估算后一个点的极角差：

$$\Delta\theta_0 = \frac{b}{r_i}$$

其中，$r_i = r_0 + d\theta_i$　初步估算后一个点的极角：

$$\theta_{i+1} = \theta_i + \Delta\theta_0$$

对应的极径：

$$r_{i+1} = r_0 + d(\theta_i + \Delta\theta_0)$$

Step3: 递归减小步长 $\Delta\theta$

在每一轮迭代中，减小极角差：

$$\Delta\theta_{n+1} = \frac{\Delta\theta_n}{2}$$

更新对应的极角和极径：

$$\theta_{n+1} = \theta_1 + \Delta\theta_{n+1}$$

$$r_{n+1} = r_0 + d\theta_{n+1}$$

重新计算新的平均速度：

$$v_{\text{avg},n+1} = \frac{\sqrt{(r_{n+1} - r_1)^2 + (r_{n+1}\Delta\theta_{n+1})^2}}{\Delta t_{n+1}}$$

同时减小时间步长：

$$\Delta t_{n+1} = \frac{\Delta t_n}{2}$$

Step4: 计算后一个点的速度

当 $\Delta\theta_n$ 足够小时，计算后一个点的瞬时速度：

$$v_{i+1} = \frac{\sqrt{(r_{n+1} - r_1)^2 + (r_{n+1}\Delta\theta_{n+1})^2}}{\Delta t_n}$$

Step5: 判定收敛

设定一个阈值 ϵ，当满足以下条件时，认为计算已经收敛：

$$|v_i^{(n+1)} - v_i^{(n)}| < \epsilon$$

或者：

$$|\Delta s_{n+1} - \Delta s_n| < \epsilon$$

此时，质点的速度 v_2 即为该点的瞬时速度。

5.3 求解结果

结果如下表所示：

表 1　问题 1-指定把手位置解

	时间 (s)					
	0 s	60 s	120 s	180 s	240 s	300 s
龙头 x (m)	8.8	5.7992	-4.0849	-2.9636	2.5945	4.4203
龙头 y (m)	0	-5.7711	-6.3045	6.0948	-5.3567	2.3204
第 1 节龙身 x (m)	8.3638	7.4568	-1.4452	-5.2372	4.8214	2.4591
第 1 节龙身 y (m)	2.8266	-3.4403	-7.4059	4.3596	-3.5617	4.4027
第 51 节龙身 x (m)	-9.5218	-8.6925	-5.5600	2.8744	5.9888	-6.3025
第 51 节龙身 y (m)	1.3203	2.5198	6.3636	7.2559	-3.8144	0.4530
第 101 节龙身 x (m)	2.9595	5.7221	5.3966	1.9340	-4.8897	-6.2532
第 101 节龙身 y (m)	-9.9052	-7.9768	-7.5334	-8.4640	-6.4016	3.9120
第 151 节龙身 x (m)	10.8488	6.6305	2.3222	0.9448	2.9151	7.0162
第 151 节龙身 y (m)	1.9076	8.1775	9.7441	9.4316	8.4179	4.4330
第 201 节龙身 x (m)	4.4576	-6.6938	-10.6401	-9.2523	-7.4088	-7.4195
第 201 节龙身 y (m)	10.7669	8.9716	1.2618	-4.3251	-6.2396	-5.3195
龙尾 (后) x (m)	-5.1991	7.4446	10.9662	7.3137	3.1586	1.7085
龙尾 (后) y (m)	-10.7297	-8.7314	0.9539	7.5620	9.4980	9.3163

表 2　问题 1-指定把手速度解

	时间 (s)					
	0 s	60 s	120 s	180 s	240 s	300 s
龙头 (m/s)	1.00000	1.00000	1.00000	1.00000	1.00000	1.00000
第 1 节龙身 (m/s)	0.95910	1.00445	1.00548	1.00711	0.98491	0.98491
第 51 节龙身 (m/s)	0.89870	1.08262	1.03450	1.01538	0.91805	0.91805
第 101 节龙身 (m/s)	0.86460	1.18722	1.10741	1.03110	0.96100	0.96100
第 151 节龙身 (m/s)	0.92124	1.08296	1.11421	0.96208	0.95678	0.95678
第 201 节龙身 (m/s)	0.91632	1.10481	1.20054	0.90079	1.05246	1.05246
龙尾 (后) (m/s)	0.87793	1.13501	1.13142	0.83074	0.99719	0.99719

六、 问题二的模型的建立和求解

6.1 模型建立

6.1.1 物理模型简化

在二维平面中，假设每个板凳为宽度 w 和长度 l 的矩形。每个矩形由其中心点坐标 (x, y) 和旋转角度 θ 描述。

我们将该问题中盘入时碰撞的检测抽象并简化为不相邻的任意矩形之间是否出现重叠(发生碰撞)。

对于每个矩形，其位置和方向可以通过以下两个变量描述：

- 中心点位置：(x_i, y_i)
- 旋转角度：γ

6.1.2 检测矩形是否碰撞

在检测非相邻矩形是否碰撞时，我们可以利用矩形的轴对齐边界盒（AABB）或分离轴定理（SAT）。由于每个矩形都可以通过其中心点和旋转角度定义，我们可以使用以下几何方法检测矩形之间的碰撞。

分离轴定理（SAT）：分离轴定理用于检测两个旋转矩形是否发生碰撞。

步骤如下：

1. 获取矩形的所有边的法线作为可能的分离轴。
2. 将矩形的四个顶点投影到每个轴上，计算投影的最小值和最大值。
3. 检查投影是否重叠。如果存在一个轴上没有重叠，则矩形不碰撞。否则，它们碰撞。

检测公式

设矩形 A 和 B 的四个顶点分别为 A_1, A_2, A_3, A_4 和 B_1, B_2, B_3, B_4，每个顶点投影在某条轴上的最小值和最大值分别为 $P_A^{\min}, P_A^{\max}$ 以及 $P_B^{\min}, P_B^{\max}$。如果满足以下条件之一，则矩形不发生碰撞：

$$P_A^{\max} < P_B^{\min} \quad 或 \quad P_B^{\max} < P_A^{\min}$$

6.1.3 根据运动方程步进模拟运动过程

两个矩形的运动方程如下：

$$\begin{cases} x_A(t) = x_{A0} + v_{A_x} t \\ y_A(t) = y_{A0} + v_{A_y} t \\ \theta_A(t) = \theta_{A0} + \omega_A t \end{cases}$$

$$\begin{cases} x_B(t) = x_{B0} + v_{B_x} t \\ y_B(t) = y_{B0} + v_{B_y} t \\ \theta_B(t) = \theta_{B0} + \omega_B t \end{cases}$$

6.1.4 碰撞检测与碰撞时刻计算

在时间步进模拟中，使用分离轴定理检测两个矩形是否发生碰撞。如果在某一时刻满足碰撞条件，则记录该时刻为最早碰撞时刻。

时间步进法检测碰撞

在每个时间步 $t_k = k \cdot \Delta t$ 处，计算矩形的位置和旋转角度，然后使用 SAT 方法检测碰撞。最早的碰撞时间为：

$$t_{\text{collision}} = k \cdot \Delta t$$

在时间步进模拟中，我们通过以下步骤检测最早的碰撞时刻：

1. 每个时间步长 Δt 计算矩形 A 和 B 的新位置和旋转角度。
2. 通过分离轴定理检测矩形是否碰撞。
3. 如果检测到碰撞，记录该时刻 t 为最早的碰撞时刻。

假设我们找到矩形 A 和 B 的碰撞时刻为 $t_{\text{collision}}$，此时满足以下条件：

$$P_A^{\max} \geq P_B^{\min} \quad \text{且} \quad P_B^{\max} \geq P_A^{\min}$$

6.2 模型求解

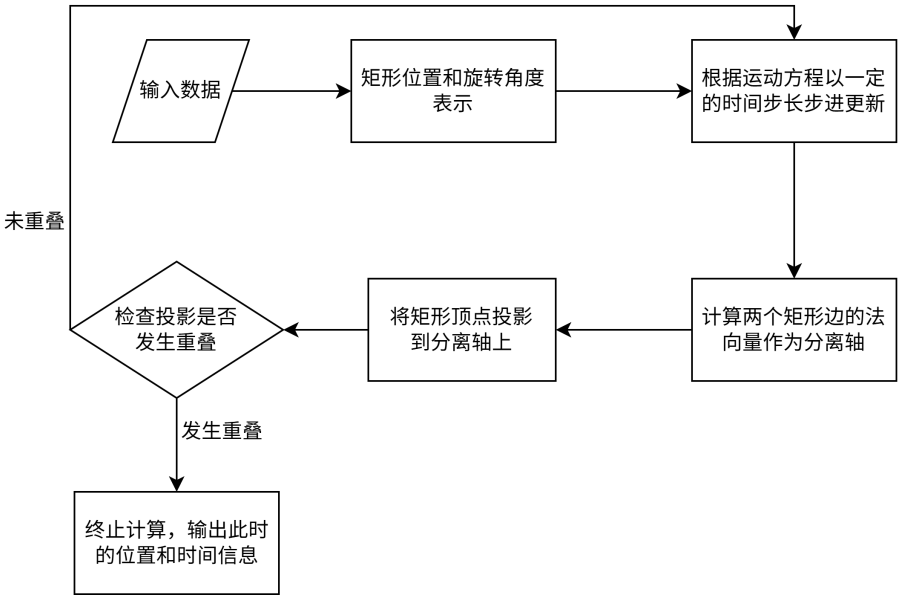


图 6　碰撞检测算法流程图

Step1：矩形位置与旋转角度表示

假设每个矩形 i 用其中心点 (x_i, y_i) 和旋转角度 θ_i 表示。对于一个宽为 w，长为 l 的矩形，其四个顶点相对于中心点的坐标为：

$$P_1 = \left(-\frac{l}{2}, -\frac{w}{2}\right), \quad P_2 = \left(\frac{l}{2}, -\frac{w}{2}\right), \quad P_3 = \left(\frac{l}{2}, \frac{w}{2}\right), \quad P_4 = \left(-\frac{l}{2}, \frac{w}{2}\right)$$

旋转矩阵为：

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

旋转后的顶点坐标为：

$$P_i' = R(\theta) \cdot P_i + \begin{pmatrix} x \\ y \end{pmatrix}$$

Step2：分离轴定理 (SAT)

对于两个矩形 A 和 B，通过以下步骤检测碰撞：

1. 计算矩形 A 和 B 的边的法向量作为分离轴。
2. 将矩形的顶点投影到每个分离轴上。
3. 检查投影是否重叠。如果存在一个轴上没有重叠，则矩形不碰撞；否则，它们碰撞。

顶点 $P = (x, y)$ 在轴 normal $= (n_x, n_y)$ 上的投影为：

$$\text{projection}(P) = x \cdot n_x + y \cdot n_y$$

Step3：运动方程

矩形的运动方程为：

$$\begin{cases} x_i(t) = x_{i0} + v_{i_x} t \\ y_i(t) = y_{i0} + v_{i_y} t \\ \theta_i(t) = \theta_{i0} + \omega_i t \end{cases}$$

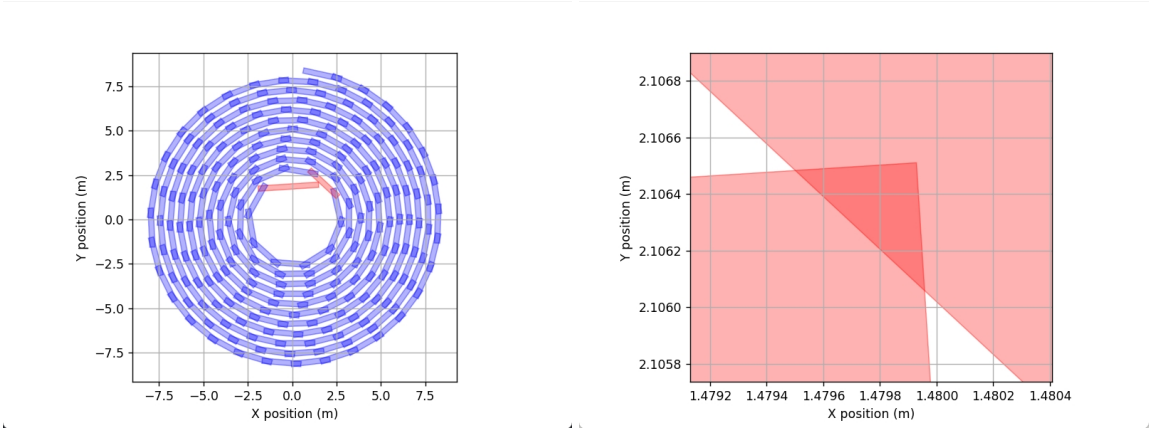
Step4：时间步进法检测碰撞

在每个时间步 $t_k = k \cdot \Delta t$ 处，计算矩形的位置和旋转角度，然后使用 SAT 方法检测碰撞。最早的碰撞时间为：

$$t_{\text{collision}} = k \cdot \Delta t$$

6.3 求解结果

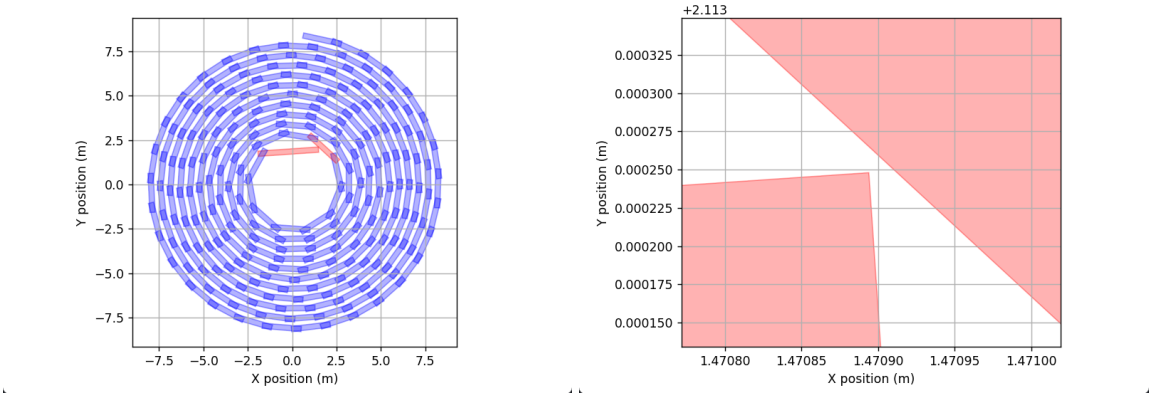
经过模型求解，我们得出，当时间为 412.48s 时，龙头与龙头后第 8 个龙身相碰撞，整个舞龙队的位置示意图如图7所示。



(a) 碰撞时刻整体示意图 **(b) 碰撞位置特写示意图**

图 7　碰撞时刻示意图

而当时间为 412.47s 及 412.47s 之前时，整个舞龙队没有发生碰撞。时间为 412.47s 时，整个舞龙队的位置示意图如图8所示。



(a) 整体示意图 **(b) 特写示意图**

图 8　终止时刻位置示意图

故终止时刻为 **412.47s**。

在终止时刻，龙头前把手、龙头后面第 1、51、101、151、201 条龙身前把手和龙尾后把手的位置和速度如表3所示。

七、 问题三的模型的建立和求解

7.1 模型建立

从盘入到盘出，舞龙队的整体运动包括盘入、掉头、盘出三个阶段，其中盘入过程在盘入螺线上进行，盘出过程在盘出螺线上进行，调头过程则在调头空间内的调头曲线

表 3　问题 2-412.48s 碰撞时刻各舞龙节点位置、速度信息

舞龙节点	横坐标 x (m)	纵坐标 y (m)	瞬时速度 (m/s)
龙头	1.215029	1.939322	0.999997
第 1 节龙身	-1.639184	1.757404	0.989926
第 50 节龙身	2.734151	3.548967	0.925963
第 100 节龙身	-2.122512	-5.483738	0.921213
第 150 节龙身	-0.650944	-6.974040	0.957198
第 200 节龙身	-7.962477	0.367672	1.049416
龙尾（后）	0.897511	8.329893	1.019227

上进行，本题要求找到最小螺距，使得龙头前把手能够沿着相应的螺线盘入到调头空间的边界，故只涉及盘入过程的计算。首先，可以证明，随着螺距的减小，在碰撞时刻，龙头前把手距离原点的距离会增大，故存在一个临界螺距：当螺距小于临界螺距，碰撞时，龙头前把手距离原点的距离大于调头空间的半径，即会在龙头前把手盘入到调头空间边界之前就发生碰撞；反之，则会在龙头前把手盘入到调头空间内部后发生碰撞。这个临界螺距即为我们需要的最小调头螺距，下面给出最小调头螺距的计算模型。

7.1.1 最小调头螺距的计算模型

螺线盘入模型

盘入螺线仍旧是等距螺线，但螺线的螺距是一个二分变化的量

$$\begin{cases} x(\theta) = r(\theta) \cdot \cos(\theta) \\ y(\theta) = r(\theta) \cdot \sin(\theta) \\ r(\theta) = r_0 + b \cdot \theta \end{cases}$$

调头空间的约束条件

舞龙队需要进入调头空间，该空间是一个直径为 9 米的圆形区域。我们需要保证舞龙队在盘入路径上能顺利进入该区域，即满足以下约束条件：

$$x^2 + y^2 \leq \left(\frac{9}{2}\right)^2 = 20.25\text{平方米}$$

二分搜索最小螺距

由于板凳的宽度为 30cm，故螺距不应该小于 30cm，而根据问题二的结果，我们可以计算出当螺距为 55cm，碰撞时龙头前把手与原点的距离小于给定调头空间的半径，故所求的最小螺距应该在 30cm 到 55cm 之间。为了找到最小螺距 b ，我们可以使用二分搜索方法：

1. 初始条件：

定义螺距的初始上下界为 $b_{\text{min}} = 0.3$ 和 $b_{\text{max}} = 0.55$ 米。

2. 迭代过程：

计算中间值 $b_{\text{mid}} = \frac{b_{\text{min}}+b_{\text{max}}}{2}$；使用 b_{mid} 计算螺旋线路径，并判断该路径是否进入调头区域；如果可以进入，则更新 $b_{\text{max}} = b_{\text{mid}}$，否则更新 $b_{\text{min}} = b_{\text{mid}}$；重复迭代，直到 $b_{\text{max}} - b_{\text{min}}$ 小于预设的容差 ϵ。

3. 终止条件：

当螺距的上下界差值收敛到给定精度(如 10^{-6})时,停止搜索并返回最优螺距 b_{optimal}。

7.2 模型求解

Step1：初始条件设定

设定螺旋线的螺距 p，初始位置 r_0，初始角度 θ_0。

板凳参数：板长 L，板宽 W，龙头速度 v，每秒时间步长 Δt。

调头空间：调头空间半径 R，中心坐标 (x_c, y_c)。

Step2：盘入过程模拟

根据螺旋线方程 $r(\theta) = r_0 + \frac{p}{2\pi} \cdot \theta$ 计算把手位置 $x_i(t)$，$y_i(t)$ 以及速度 $v_i(t)$。

使用递推关系 $v_{i+1}(t) = f(v_i(t), \theta_i(t), \Delta\theta)$ 逐步计算各个板凳速度。

检测相邻板凳的碰撞情况，直到进入调头空间。

Step3：二分搜索最小螺距

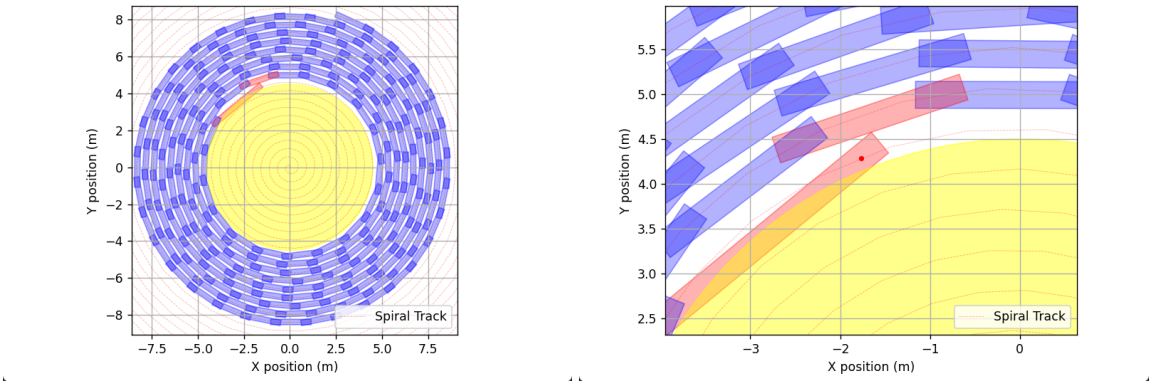
通过二分搜索调整螺距 p，模拟盘入，找到龙头能够进入调头空间的最小螺距。

算法优化

为了减少计算时间，我们可以对模拟范围进行优化，即将龙头前把手起始点定为调头空间之外一圈的盘入螺线轨道上，并将结束点定为调头空间之内一圈的盘入螺线轨道上。相较于从头开始进行模拟全程，这种方法可以大大减少计算时间。

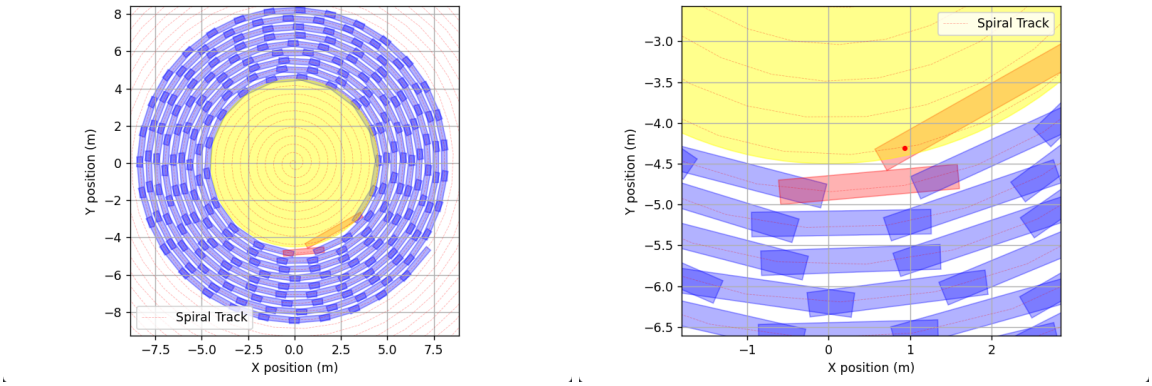
7.3 求解结果

经过模型求解，当螺距为 0.449414 米时，碰撞情况如图9所示,可以看到，此时龙头前把手并未进入调头空间边界，而当螺距为 0.450391 米时，碰撞情况如图10所示，此时龙头前把手已经进入调头空间边界，故我们取 0.450391 米为问题三的解，即最小调头螺距约为 0.450391 米。



(a) 整体示意图　　(b) 特写示意图

图 9　碰撞时刻示意图（边界外）



(a) 整体示意图　　(b) 特写示意图

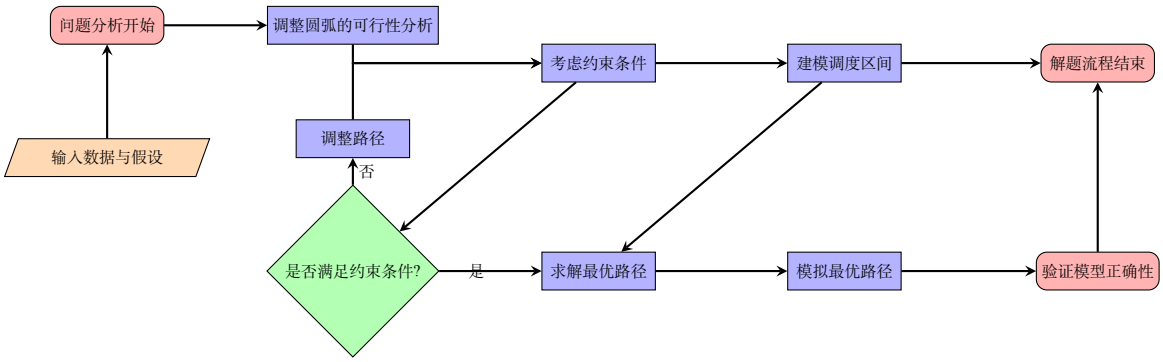
图 10　碰撞时刻示意图（边界内）

八、 问题四的模型的建立和求解

8.1 模型建立

在问题四中，舞龙队需要在以螺距为 1.7 米的螺线盘入，并在问题 3 中设定的调头空间内完成调头。调头路径为两段相切的 S 形曲线，其中前一段圆弧的半径是后一段的 2 倍，并且该路径与盘入、盘出螺线相切。我们探讨是否可以通过调整圆弧，保持各部分相切的前提下，缩短调头曲线的长度。首先，我们对于调整圆弧的可行性进行探讨，发现舞龙队进入题给调度区间后，并不一定立即开始调头，故在此我们又阐述了题给调头区域和真实调头区域之间的差异与联系。其次，不同潜在的约束限制着约舞龙队行进路线，约束的细节考虑都需要被重视，我们列举出可能存在的约束条件并加以解释说明。然后，基于以上问题规约假设，我们对真实调度区间进行建模，求出不同真实调度

区间下调头曲线的长度，并根据碰撞约束求解出最短调头路径。最后，我们模拟最优路径，获取各个时刻舞龙队的位置、速度信息，检验模型的正确性。



8.1.1 调整圆弧的可行性假设

考虑进入题给调头空间时，舞龙队伍立刻进入调头阶段，此时 $R = 4.5$, 情况如下图，

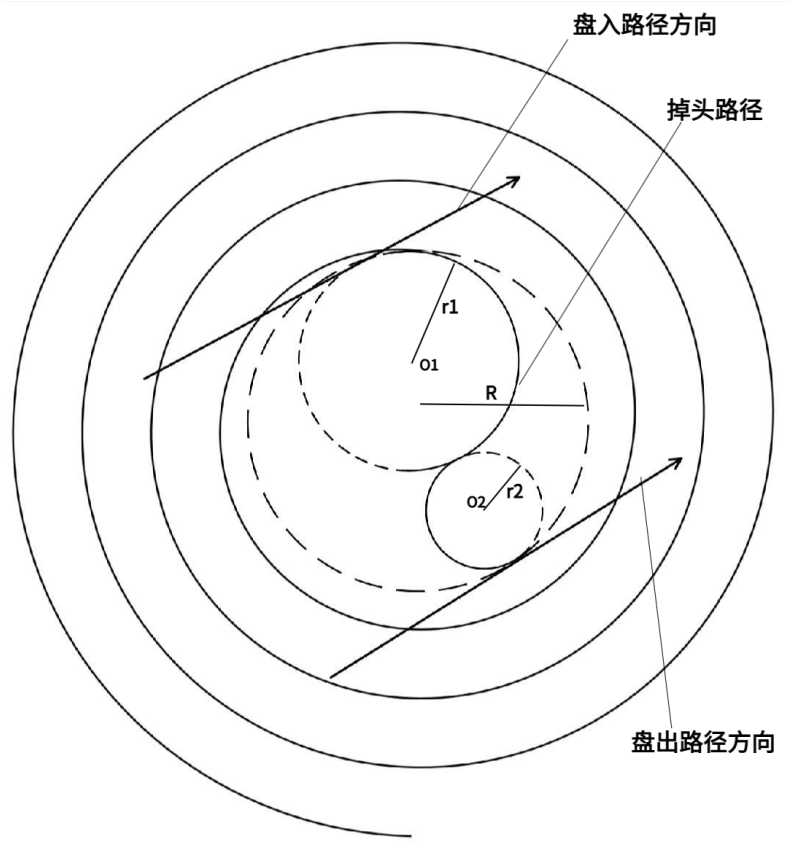


图 11　掉头路径示意图

根据调头路径的几何约束，两个圆的切点与螺线上的两个切点相重合，这决定了圆

的位置和大小。由于两个圆的半径比为 2:1，它们的大小也唯一确定。加之两圆相切且与螺线相切，这样的几何关系确保了这两个圆在图中的位置是唯一确定的。

由此我们可以知道，进入题给调头空间时，舞龙队伍立刻进入调头阶段，在这种情况下，舞龙队伍的调头路径只有唯一解，与题意寻找最短路径的要求不符合。故此，在本题的解题过程中，需要确定一些假设如下，

1. 进入题给调头空间时，并不要求立刻进行队伍调头。
2. 如果进入题给调头空间时队伍不立即调头，队伍将沿着原盘入螺线进入。
3. 调头空间内设定的盘入螺线轨道纳入全局的盘入螺线轨道定义中。
4. 调头空间内设定的盘出螺线轨道纳入全局的盘出螺线轨道定义中。
5. 根据假设 3.4，调头空间内的盘入螺线与调头空间内的盘出螺线关于圆心中心对称。
6. 舞龙队伍在行进过程中只前进不后退。

8.1.2 问题简化——真实调度区间与题给调度区间的关联与差异

(1) 真实调度区间的定义

在实际的调度问题中，给定的调度区域为以原点为圆心、直径为 9 米的圆形区域。然而，真实的调度区域的半径并非恒定不变。具体而言，真实的调度区域可以表示为以原点为圆心、直径为 R 的圆形区域，其中 R 的取值范围为从碰撞点到圆心的距离 R_c 到 9 米之间。因此，调度区域的半径 R 满足

$$R \in [R_c, 4.5]$$

需根据具体的碰撞点位置动态调整。

(2) 真实调度区间与题给调度区间的关联与差异

根据 8.1.1 小节的推导，我们可以知道在调头空间固定的情况下，舞龙队伍的调头路径只有唯一解，所以调头路径的长度解和调头空间的半径存在关联。再者，无论是否在进入题给调度空间时立刻调转，盘入螺线和盘出螺线在几何上中心对称的特点都表明，真实调度空间无非是和题给调度空间半径不同，其他的路径求解思路都是相同的，故此，我们只需要撰写求取特定半径调度区间的调度路径的函数，就可以通过遍历半径求出所有可能的调度路径解。

8.1.3 碰撞约束分析

(1) 龙头往返碰撞约束

龙头往返碰撞约束是符合直觉的，但是我们要定量的分析往返路径叠加情况下，到底对螺距的有效值造成什么样的影响，在此我们证明我们证明了盘入和盘出的路径相互重叠的现象相当于螺距的有效值减半。这意味着龙头在某个范围内运动时，更容易与调头边界或龙身碰撞。

为了证明往返两个路径重叠相当于螺距减半，我们首先需要明确一些关键的几何和数学概念。问题的本质是在螺旋线盘入和盘出时，由于两条路径关于中心对称，它们之间的最小距离等效于螺距的一半。下面我们逐步展开证明。

在极坐标系下，螺旋线可以用以下形式定义：

$$r(\theta) = r_0 + P \cdot \frac{\theta}{2\pi}$$

其中：

- $r(\theta)$ 表示螺旋线在角度 θ 时的径向距离；
- P 是螺距，表示螺旋线每旋转一圈时，径向距离的增加量；
- θ 是极角，表示点的角度。

假设螺旋线盘入时的螺距为 P_{in}，盘出时的螺距为 P_{out}，且这两条螺旋线关于中心 O 对称。

① 螺旋线的盘入和盘出

在盘入过程中，龙头沿顺时针方向移动，路径可以表示为：

$$r_{\text{in}}(\theta) = r_0 + P_{\text{in}} \cdot \frac{\theta}{2\pi}$$

当龙头进入调头空间并开始盘出时，路径改为沿逆时针方向移动，这时螺旋线的极角 θ_{out} 和盘入的极角 θ_{in} 之间存在如下关系：

$$\theta_{\text{out}} = -\theta_{\text{in}}$$

盘出的路径可以表示为：

$$r_{\text{out}}(\theta_{\text{out}}) = r_0 + P_{\text{out}} \cdot \frac{\theta_{\text{out}}}{2\pi}$$

② 往返路径的相对位置

螺旋线的盘入和盘出是关于中心 O 对称的。为了避免路径之间的重叠，必须确保盘入和盘出的径向距离有足够的分离。

设某个时间 t 时，龙头的位置为 $(r_{\text{in}}(t), \theta_{\text{in}}(t))$，盘入路径和盘出路径之间的最小径向距离是：

$$\Delta r = r_{\text{out}}(\theta_{\text{out}}) - r_{\text{in}}(\theta_{\text{in}})$$

由 $\theta_{\text{out}} = -\theta_{\text{in}}$ 并代入螺旋线的径向公式，可以得到：

$$\Delta r = \left(r_0 + P_{\text{out}} \cdot \frac{-\theta_{\text{in}}}{2\pi} \right) - \left(r_0 + P_{\text{in}} \cdot \frac{\theta_{\text{in}}}{2\pi} \right)$$

考虑到盘入与盘出的螺距是相等的，即 $P_{\text{in}} = P_{\text{out}} = P$，我们可以得到：

$$\Delta r = P \cdot \frac{-\theta_{\text{in}} - \theta_{\text{in}}}{2\pi} = -P \cdot \frac{2\theta_{\text{in}}}{2\pi} = -P \cdot \frac{\theta_{\text{in}}}{\pi}$$

③ 路径重叠与螺距减半的关系

由于盘入与盘出路径的角度是相对对称的，当两条路径足够接近时，它们之间的最小距离将会逐渐减小。如果我们观察路径的径向距离在 $\theta_{\text{in}} \approx 0$ 处，即在接近中心区域时，两条路径的径向距离近似为：

$$\Delta r \approx P \cdot \frac{0}{\pi} = 0$$

这表明，在中心区域附近，盘入与盘出的路径会重叠。而为了避免路径重叠，螺距必须被有效减半，以确保盘入和盘出路径之间有足够的距离。换句话说，当螺旋线的螺距减半时，盘入和盘出的径向距离不会收缩至零，确保路径不会相互碰撞。

因此，为了避免路径重叠，我们可以等效地认为，盘入和盘出的有效螺距为 $P_{\text{eff}} = \frac{P}{2}$。这意味着在中心对称的螺旋路径上，避免路径重叠的有效最小螺距等于原始螺距的一半。

(2) 龙头盘入最大深度碰撞约束

这个约束涉及龙头在系统中进入调度区间的最大深度，根据第二问的做法，我们可以轻松得到龙头盘入最大深度碰撞约束。

(3) S 型调头路径刚体弯折约束

① 系统概述

考虑板凳龙的几何构造：龙头和龙身是由多个刚性连接的板凳组成，每个板凳的长度是固定的，且相邻的板凳通过把手连接。由于板凳龙的龙身长度较长，在过弯或调头时，尤其是当转弯半径较小时，前后两端板凳的运动方向可能出现差异。

② 曲率与圆弧直径

设龙头和龙身处在一个圆弧上，而这个圆弧的半径 r 小于龙身的总长度 l。在这样的条件下，刚体的前后两部分不能简单地沿同一方向移动，因为龙身的整体几何约束要求它保持不可伸缩和刚性。

在这种情况下，弯曲路径的曲率 κ 与圆弧半径 r 之间的关系为：

$$\kappa = \frac{1}{r}$$

曲率越大，圆弧的半径 r 就越小，弯道越急。

③ 前后把手的运动方向，但后把手向后运动

假设龙头正处在弯道上，其中龙头的前把手位于圆弧的前端，而龙头的后把手以及后续的龙身板凳依次排列在同一圆弧上。由于圆弧的半径小于板凳的总长度，因此刚性板凳必须在弯曲过程中调整角度以适应这个曲率。

具体来说，前把手沿着圆弧的切线方向向前移动，而后把手的运动则受限于两方面：

- **刚性限制**：龙身各部分之间的距离是固定的，因此在曲率较大（即半径较小）的圆弧上，后把手可能无法沿同一方向向前移动。
- **几何关系**：在小半径的圆弧上，后把手需要适应前把手的运动和角度变化。为了保证龙身的整体刚性，后把手可能会向相反方向移动，即向后退，以避免龙身发生变形或折叠。

因此，当龙头进入小于自身长度的圆弧时，前把手沿着圆弧的切线方向继续前进，而为了维持刚性结构，后把手可能出现向后的运动，以调整角度，适应当前的曲率。

设龙头前后把手的位置分别为 $r_{\text{front}}(t)$ 和 $r_{\text{back}}(t)$，它们的相对位置满足刚性条件，即

$$|r_{\text{front}}(t) - r_{\text{back}}(t)| = l$$

其中 l 为两把手之间的距离（即板凳的长度）。在小半径的圆弧上，前把手的速度方向是沿着圆弧的切线方向，而后把手由于曲率的限制，其速度方向可能出现向后偏移的趋势。

设圆弧的半径为 r，龙头的前后把手运动速度分别为 $v_{\text{front}}(t)$ 和 $v_{\text{back}}(t)$，则可以通过几何关系得到：

$$v_{\text{front}}(t) = v_{\text{front}}\hat{t}, \quad v_{\text{back}}(t) = v_{\text{back}}\hat{n}$$

其中 $\hat{t}$ 表示前把手沿圆弧的切线方向，而 $\hat{n}$ 表示后把手在保持刚性时的运动方向，可能向后。

为了避免折叠，必须满足：

$$v_{\text{back}} \leq 0$$

即后把手的速度可以向后，这样才能保持整体刚体结构的完整性。

8.2 模型求解

Step1：1.7m 螺距行进路线的碰撞点求解

同第二问，本步骤的模型求解采用使用数值模拟方法，通过设定龙头前把手的速度、时间步长及总行进时间，计算了螺距为 1.7 米的盘入路线是否会发生碰撞。具体实验参数如下：

- **龙头前把手速度：** $v_{\text{head}} = 1.0$ m/s
- **时间步长：** $\Delta t = 0.01$ 秒

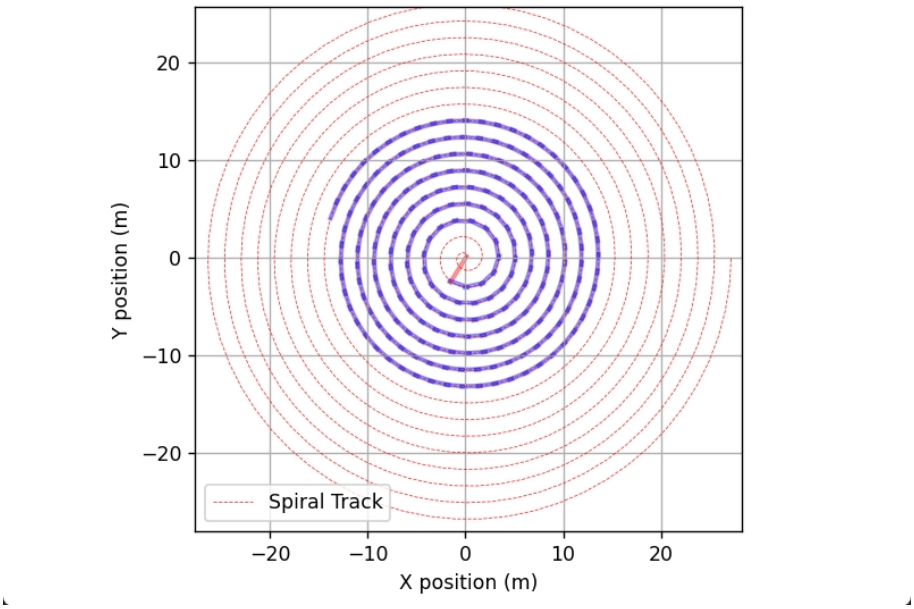


图 12　1.7m 螺距碰撞点测试结果 (抵达螺线中心)

- **总时间：** $T_{\text{total}} = 700$ 秒
- **螺距：** $P = \frac{170}{100} = 1.7$ 米
- **搜索步长：** $\Delta s = 0.0001$ 米

螺距 P 设定为 1.7 米，确保螺旋线盘入的每圈间隔固定。我们通过数值积分的方法，以每秒 1 米的速度模拟龙头前把手在等距螺旋线上的运动。在总时间为 700 秒的模拟中，如果龙头前把手未发生碰撞，则可得出龙头在 700 秒内完成的盘入深度。

通过对龙头前把手的 700 秒全程运动模拟，我们发现在螺距为 1.7 米的情况下，龙头前把手在螺旋线上的最深入点未发生碰撞。即龙头在整个盘入过程中，始终保持与路径的足够间隔，未发生重叠或碰撞现象，实验结果如图12。

通过本次模拟，我们成功验证了在螺距为 1.7 米的情况下，龙头前把手不会发生碰撞。**该结果表明我们不再需要考虑真实调度区间半径的约束**，即如下公式，

$$R \in [R_c, 4.5] = R \in [0, 4.5]$$

Step2：碰撞约束下，调度路径圆弧长度最小值求解

根据 8.1.3 中 (3) 的推导，我们可以明晰 S 型调头路径刚体弯折约束的本质就是龙头把手间距不大于调头路径的小圆弧直径，那么以小圆直径等于龙头把手间距作为最小情况，我们可以进行求解。

Step3：基于龙格-库塔法和有限差分方法模拟 step2 中最短路

龙格-库塔法（Runge-Kutta methods）是一种常用的数值解法，用于求解常微分方程的初值问题。这种方法通过一系列的中间步骤来逼近微分方程的解，提高了欧拉方法

的精度而不需要减小步长。

有限差分方法（Finite Difference Methods, FDM）是解决偏微分方程（PDEs）及其他数学问题的一种数值技术。这种方法通过在连续的定义域上创建网格点来近似微分算子，然后在这些网格点上使用差分方程来模拟连续的微分方程。

在代码中我们灵活运用了以上的方法使得代码性能和精度得到提升，不同于先前积分求前继节点的方法，这个方法更加高效。模拟方法和先前类似，需要注意的是分段的处理，因为路径呈现为盘入路径、调头路径大圆弧、调头路径小圆弧和盘出路径四大段。我们采用物理中的线坐标对点进行了记录，根据到起始点的距离而非位移，我们可以轻松的得到不同板凳所处的曲线区域。

8.3 求解结果

在调度路径的求解过程中，我们确定了两个相切的小圆以及它们在螺旋线上的切点信息，保证了两个小圆的半径比为 $r_1 : r_2 = 2 : 1$。调度区间的几何信息如下：

如下图，大圆的半径为 $r_1 = 2.8610$ 米，小圆的半径为 $r_2 = 1.4305$ 米。大圆的圆心坐标为 $(x_1, y_1) = (-1.3951, -0.3527)$，小圆的圆心坐标为 $(x_2, y_2) = (2.8231, 0.4369)$。调度区间的半径为 $R = 4.2830$ 米，最短调头路径的长度为 $l = 12.940776552$ 米。

这些几何信息描述了调度区间及圆在螺旋线中的位置和大小，确保路径规划的准确性。通过这些数据，可以有效验证调度路径中切线相切的几何条件及其合理性。

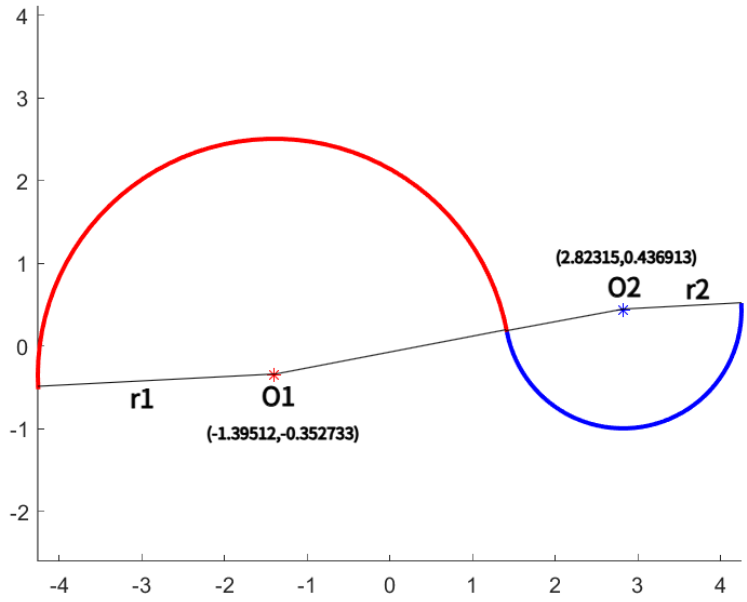


图 13　双圆弧调度路径平面图（米）

表 4　问题 4-指定把手位置解

	时间 (s)				
	-100 s	-50 s	0 s	50 s	100 s
龙头 x (m)	8.504281	6.545191	-4.251182	3.834124	-0.373518
龙头 y (m)	0.28939	-1.589901	-0.521094	4.865216	8.07887
第 1 节龙身 x (m)	8.018835	6.732593	-3.150662	5.507252	2.447255
第 1 节龙身 y (m)	3.10789	1.263953	-3.160877	2.545681	7.606811
第 51 节龙身 x (m)	-9.091736	-6.530688	-1.017323	-4.540841	3.93409
第 51 节龙身 y (m)	5.998373	-6.997873	-7.976059	-4.148095	1.907523
第 101 节龙身 x (m)	-12.704884	7.848779	6.102679	-5.052441	-5.483463
第 101 节龙身 y (m)	-1.376099	-8.64236	8.490779	7.549979	5.041533
第 151 节龙身 x (m)	-14.339188	11.491358	-3.830112	-7.488298	7.609457
第 151 节龙身 y (m)	1.535256	-6.99337	11.804052	-8.433924	-6.501556
第 201 节龙身 x (m)	-9.306841	7.698683	-3.604215	-3.119784	10.564111
第 201 节龙身 y (m)	12.884486	-12.897626	13.626357	-12.733373	5.76585
龙尾（后）x (m)	-4.46955	3.668479	-5.307366	8.774576	-12.327158
龙尾（后）y (m)	-15.883924	15.22531	-13.789489	10.701865	-3.541095

表 5　问题 4-指定把手速度解

	时间 (s)				
	-100 s	-50 s	0 s	50 s	100 s
龙头 (m/s)	1.0000	1.0000	1.0000	1.0000	1.0000
第 1 节龙身 (m/s)	0.999899	0.999741	0.998294	1.000393	1.00013
第 51 节龙身 (m/s)	0.999317	0.998551	0.994284	1.095513	1.004391
第 101 节龙身 (m/s)	0.999055	0.998143	0.993566	1.093713	1.012133
第 151 节龙身 (m/s)	0.998905	0.997938	0.993266	1.093181	1.011202
第 201 节龙身 (m/s)	0.998809	0.997813	0.993101	1.092927	1.010847
龙尾 (后) (m/s)	0.998776	0.997773	0.993049	1.092853	1.010752

九、 问题五的模型的建立和求解

9.1 模型建立

9.1.1 螺线与圆弧运动的等价性

速度和加速度的计算

1. 螺线的速度和加速度

速度（Tangential Velocity）

在极坐标系中，质点的速度 **v** 可以表示为：

$$v = \frac{dr}{dt}$$

其中

$$r = r(\theta)e_r$$

利用链式法则，速度分量为：

$$v = \frac{dr}{dt}e_r + r\frac{de_r}{dt}$$

在极坐标系中：

$$\frac{de_r}{dt} = \dot{\theta}e_\theta$$

因此：

$$v = \frac{dr}{dt}e_r + r\dot{\theta}e_\theta$$

对于螺线：

$$r(\theta) = a + b\theta$$

$$\frac{dr}{dt} = \frac{d(a+b\theta)}{dt} = b\dot{\theta}$$

因此：

$$v = b\dot{\theta}\mathbf{e}_r + (a+b\theta)\dot{\theta}e_\theta$$

加速度（Tangential and Radial Acceleration）

加速度分量为：

$$a = \frac{dv}{dt}$$

$$a = \frac{d}{dt}(b\dot{\theta}e_r + (a+b\theta)\dot{\theta}e_\theta)$$

注意到：

$$\frac{de_r}{dt} = \dot{\theta}e_\theta$$

$$\frac{de_\theta}{dt} = -\dot{\theta}e_r$$

所以：

$$a = \left(b\ddot{\theta}e_r + b\dot{\theta}^2 e_\theta\right) + \left((a+b\theta)\ddot{\theta}e_\theta - (a+b\theta)\dot{\theta}^2 e_r\right)$$

$$a = \left(b\ddot{\theta} - (a+b\theta)\dot{\theta}^2\right)e_r + \left(b\dot{\theta}^2 + (a+b\theta)\ddot{\theta}\right)e_\theta$$

2. 圆弧的速度和加速度

对于圆弧：

$$r = R$$

$$\frac{dr}{dt} = 0$$

速度分量为：

$$v = R\dot{\theta}e_\theta$$

加速度分量为：

$$a = \frac{d}{dt} = R\ddot{\theta}e_\theta + R\dot{\theta}^2 e_r$$

3. 等价性的证明

我们可以通过比较螺线和圆弧运动的速度和加速度来证明它们的等价性。具体来说，我们需要找出在什么条件下它们的速度和加速度相等。

速度等价条件

速度分量：对于螺线：

$$v = b\dot{\theta}e_r + (a+b\theta)\dot{\theta}e_\theta$$

对于圆弧：

$$v = R\dot{\theta}e_\theta$$

比较：

$$b\dot{\theta} = 0$$

$$(a+b\theta)\dot{\theta} = R\dot{\theta}$$

$$a + b\theta = R$$

$$\theta = \frac{R-a}{b}$$

加速度等价条件

加速度分量：对于螺线：

$$a = \left(b\ddot{\theta} - (a+b\theta)\dot{\theta}^2\right)e_r + \left(b\dot{\theta}^2 + (a+b\theta)\ddot{\theta}\right)e_\theta$$

对于圆弧：

$$a = R\ddot{\theta}e_\theta + R\dot{\theta}^2 e_r$$

比较：

$$b\ddot{\theta} - (a+b\theta)\dot{\theta}^2 = R\dot{\theta}^2$$

$$b\dot{\theta}^2 + (a + b\theta)\ddot{\theta} = R\ddot{\theta}$$

在上述等式中，如果 θ 取特定值，使得两个方程一致，那么螺线和圆弧的运动特性是等价的。

9.1.2 龙头前把手与各把手速度的线性关系

根据如下的第一问中的各把手间速度的递推关系式，我们可以认为相邻两个把手间的速度具有线性关系。

$$v_{i+1}(t) = \frac{r_0 + d(\theta_i(t) - \Delta\theta)}{r_0 + d\theta_i(t)} \cdot v_i(t)$$

使用数学归纳法递归可得: 所有把手的均与龙头前把手的速度具有线性关系。

9.1.3 速度线性关系的计算验证模型

我们可以对第四问中的运动模型在规定不同龙头前把手速度的情况下计算出各速度所对应的最快把手的速度，以此来验证所有把手与龙头前把手间速度的线性关系。

9.2 模型求解

此题我们选取合适的步长遍历 (0.6,2) 范围内所有龙头前把手速度下最快把手的速度，然后线性拟合两组速度以检查他们之间的线性关系。

Step1：几何计算

通过解析几何计算，代码确定了不同部分的圆弧和螺线的交点及其相关信息。计算了螺旋和圆弧的交点坐标以及相应的切线斜率。对于给定的圆 C_1 和 C_2，我们需要找到螺线与圆的交点角度。

圆C_1 的圆心坐标:

$$x_{C1} = R\cos(\theta_{ru}) + r_{C1}\cos(\theta_{max1} - \pi)$$
$$y_{C1} = R\sin(\theta_{ru}) + r_{C1}\sin(\theta_{max1} - \pi)$$

圆C_2 的圆心坐标:

$$x_{C2} = R\cos(\theta_{chu}) - r_{C2}\cos(\theta_{max2})$$
$$y_{C2} = R\sin(\theta_{chu}) - r_{C2}\sin(\theta_{max2})$$

Step2：数值积分

对于头节点在不同圆弧段（C1、C2）上的运动，代码通过数值积分来计算头节点的位置随时间变化的情况。这部分利用了时间步长 dt 来分段计算位置。

Step3：数值求解

使用了非线性方程求解器 fsolve 来确定在螺线或圆弧上的点之间的角度关系。具体来说：

对于圆弧 C_1 和 C_2，在每个时间点计算位置的公式如下：

圆弧C_1 上的位置信息：

$$\theta_{C1} = -\frac{v0 \cdot t_{c1}}{r_{C1}} + \theta_{max1}$$
$$X_{C1} = r_{C1}\cos(\theta_{C1}) + x_{C1}$$
$$Y_{C1} = r_{C1}\sin(\theta_{C1}) + y_{C1}$$

圆弧C_2 上的位置信息：

$$\theta_{C2} = \frac{v0 \cdot (t_{c2} - \frac{SC1}{v0})}{r_{C2}} + \theta_{min1} - \pi$$
$$X_{C2} = r_{C2}\cos(\theta_{C2}) + x_{C2}$$
$$Y_{C2} = r_{C2}\sin(\theta_{C2}) + y_{C2}$$

Step3：速度计算

通过数值差分方法计算每个时间点下每个把手的速度。使用了前向差分、后向差分和中心差分方法来计算速度分量（Vx 和 Vy），并通过速度合成公式计算总速度。

速度的计算是通过数值差分方法得到的：

$$V_x(i,j) = \frac{XX(i,j+1) - XX(i,j-1)}{2 \cdot dt}$$
$$V_y(i,j) = \frac{YY(i,j+1) - YY(i,j-1)}{2 \cdot dt}$$

速度的合成：

$$V = \sqrt{V_x^2 + V_y^2}$$

9.3 求解结果

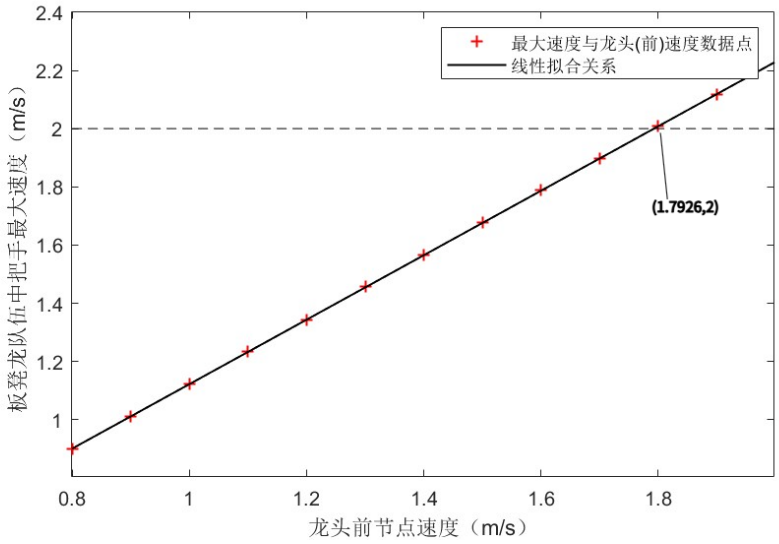


图 14　舞龙队各把手最大速度与龙头行进速度的各点对应关系

龙头最大行进速度	舞龙队各把手最大速度	龙头最大行进速度	舞龙队各把手最大速度
0.8	0.898629	1.5	1.676487
0.9	1.010200	1.6	1.787566
1.0	1.121610	1.7	1.898592
1.1	1.233439	1.8	2.007540
1.2	1.343513	1.9	2.118656
1.3	1.456195	2.0	2.226992
1.4	1.567411		

表 6　龙头最大行进速度与舞龙队各把手最大速度对应关系

十、 模型的分析与检验

图15展示了板凳龙在不同节点随时间的速度变化，螺距为 0.55 米。在整个过程中，龙头前把手的速度始终保持为 1 米/秒（如图中蓝色曲线所示），这表明龙头速度的稳定性极高，不受其他节点速度波动的影响，是整个模型的基准速度。相比之下，其他节点，尤其是位于 101 个板凳前把手和龙尾后的把手（分别以绿色和红色线表示）的速度表现出了较大的波动。

这些波动可以理解为模型对不同节点灵敏度较高的体现。在不同时间段，这些节点的速度由于相邻刚体运动的影响而产生了快速的变化，反映了模型对输入的微小变化非

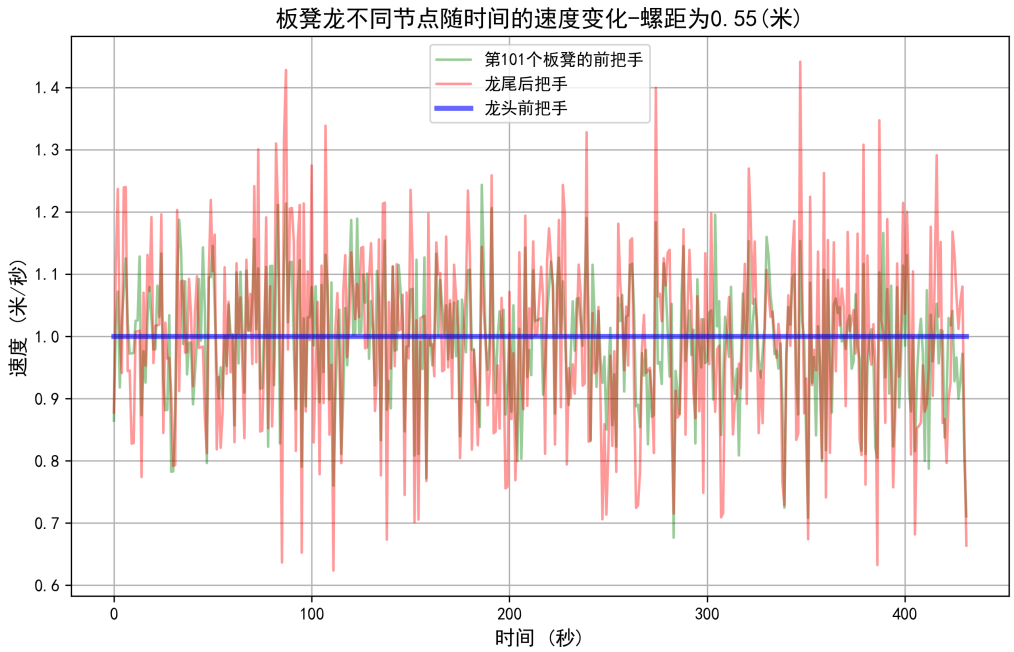


图 15　板凳龙不同节点随时间的速度变化

常敏感。尽管这些波动在视觉上表现为毛刺状，但可以看作是模型对环境变化的快速响应，即模型在保持整体稳定性的前提下，具备较强的灵敏性和对外部扰动的适应能力。

因此，可以认为模型具备较高的灵敏度，能够捕捉并反映细微的动力变化，同时保证了龙头节点的速度始终不变，从而维持了模型的整体稳定性。

十一、 模型的评价

11.1 模型的优点

1. 我们构建了等距螺线轨道上的链式刚体运动模型，采用阿基米德螺线极坐标方程描述把手的运动轨迹，并将其转换为直角坐标系。通过微小步长法和递推关系优化了计算效率，能够精准计算每秒的把手位置和速度。

2. 碰撞检测模型简化了板凳为二维矩形，利用分离轴定理（SAT）检测非相邻矩形的碰撞，并通过时间步进法逐步更新矩形的位置和角度，确保了结果的准确性和稳定性。

3. 模型验证了螺距与龙头距离原点之间的反比关系，使用二分搜索算法计算最小调头螺距，确保了调头区域的物理可行性和最优螺距解的稳定性。

4. 调头路径优化模型基于多种约束条件，包括碰撞约束、最大深度碰撞约束和 S 型路径弯折约束，优化了舞龙队调头路径长度，并通过数值模拟验证了不同区间下的调头路径，得到了较为精确的几何解。

11.2 模型的缺点

1. 虽然使用了高效的递推与微小步长方法，但由于多次迭代，求解时间较长，尤其在大规模计算时有一定瓶颈。

2. 碰撞检测模型依赖于 SAT 法，可能在非常复杂的几何形状或边界条件下遇到计算复杂度高的问题。

3. 二分法搜索螺距尽管有效，但存在收敛速度的局限性，可能在极端条件下需要更复杂的优化算法。

4. 调头路径优化过程中，对于不同半径的 S 型路径约束计算复杂度较高，可能在极端情况下导致算法的收敛性变差。

11.3 模型的推广

1. 可进一步推广为不等距螺线轨道上的刚体链式运动模型，以适应更复杂的舞龙队形变需求。

2. 可将二维矩形碰撞检测模型推广至三维场景下的碰撞检测，应用于更加复杂的三维物体交互环境中。

3. 螺距调头模型可以通过加入外部力或流体动力学模型，推广至研究更加复杂的运动行为及轨迹优化问题。

4. 调头路径优化模型可结合实际道路情况，应用于自动驾驶系统中的路径规划问题，进一步优化路径长度与行驶时间。

参考文献

[1] 金成梁. 刚体运动、几何变换中的 “平移” 与 “旋转” 是一回事吗?[J]. 教育视界, 2015 (20):78-79.

[2] 杨天宝. 深挖教材隐性知识促进典例深度学习——复杂连接体牵连速度的进阶式探究[J]. 中学物理教学参考, 2024, 53(12):36-37.

附录 A　文件列表

文件名	功能描述
simulate.py	计算第一题，模拟螺距为 0.55m 行进
calculateAngle.py	通过弧长计算角度
calculateLength.py	通过角度积分计算弧长
calculateSpeed.py	通过位置信息计算速度
code2	根据第一题给出的位置信息进行模拟并碰撞检测
code3	对调头区域附近的盘入过程进行模拟，并通过二分查找确定满足条件的最小螺距
code4.py	计算第四题，模拟四段弧线的舞龙队路径
code5.py	计算第五题，计算整个队伍的最大速度

附录 B　代码

code1_simulate.py

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from PIL import Image
from scipy.integrate import quad
from scipy.optimize import fsolve
import time as tim

start = tim.time()

# 初始化参数
num_segments = 224  # 总共224个把手
length_head = 286 / 100  # 龙头前把手距离（米）
length_body = 165 / 100  # 龙身和龙尾相邻把手距离（米）
velocity_head = 1.0  # 龙头前把手速度（m/s）
time_step = 0.01   # 时间步长（秒）
total_time = 100000  # 定义总时间为700秒(碰撞会终止)
```

```
spiral_spacing = 170 / 100  # 螺距，单位为米
search_step = 0.0001  # 搜索步长，可以调小以提高精度


# 创建用于保存图片的目录
save_dir = r'img'
os.makedirs(save_dir, exist_ok=True)

# 初始化位置数组
positions = np.zeros((num_segments, 2, int(total_time /
   time_step)))

# 起始点为A点，θ_A = 16圈
theta_start = (2.647 + 1) * 2 * np.pi


# 螺旋线方程（确保顺时针运动，从外圈向内圈，保持正的θ值）
def spiral_coordinates(theta):
    r = spiral_spacing * theta / (2 * np.pi)
    x = r * np.cos(theta)  # 使用正θ来确保方向正确
    y = r * np.sin(theta)  # 使用正θ来确保方向正确
    return x, y


# 计算两点之间的欧几里得距离
def distance(point1, point2):
    return np.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] -
    point2[1]) ** 2)


# 找到下一个满足距离的点（逆时针查找）
def find_next_position(prev_position, prev_theta,
   segment_length):
    # 沿着螺旋线搜索，增量步长可以根据精度需求调整
    theta = prev_theta + search_step # theta 递增确保逆时针查
```

```
    找
51
52      while True:
53          # 计算当前位置的螺旋线坐标
54          new_position = spiral_coordinates(theta)
55
56          # 计算当前点与上一个点的距离
57          if distance(prev_position, new_position) >=
    segment_length:
58              return new_position, theta
59
60          # 减小角度，沿螺旋线搜索
61          theta += search_step  # theta 递增确保逆时针查找
62
63
64  # 计算等距螺线的弧长
65  def arc_length(theta, b):
66      return b * np.sqrt(theta ** 2 + 1)
67
68
69  # 根据给定的起点角度A和弧长，求解终止角度B
70  def find_next_angle(A, arc_length_to_travel, b):
71      b /= 2 * np.pi
72      def length_integral(B):
73          result, _ = quad(arc_length, A, B, args=(b,))
74          return result + arc_length_to_travel
75
76      # 使用牛顿法或其他数值方法求解
77      B = fsolve(length_integral, A + arc_length_to_travel / b)
78      return B[0]
79
80
81  # 初始化龙头的位置，从A点开始 (theta_start)
82  current_position = spiral_coordinates(theta_start)
83  theta_head = theta_start
```

```
84
85  # 初始化龙头的位置，从A点开始 (theta_start)
86  current_position = spiral_coordinates(theta_start)
87  theta_head = theta_start
88
89  # 初始化 0 时刻的龙头位置
90  positions[0, :, 0] = current_position  # 记录龙头的初始位置
91
92  # 初始化 0 时刻的龙身位置
93  prev_theta = theta_start
94  for i in range(1, num_segments):  # 从第一个节点开始，逐个计算
95      # 第一个节点的距离是 length_head，其他节点是 length_body
96      segment_length = length_head if i == 1 else length_body
97      prev_position = positions[i - 1, :, 0]  # 获取前一个节点的
    位置
98
99      # 沿着螺旋线找到下一个符合距离要求的点
100     positions[i, :, 0], prev_theta = find_next_position(
    prev_position, prev_theta, segment_length)
101
102 print("Initial positions at t=0 saved.")
103
104
105 # 递归计算其余把手的位置
106 collision_detected = False
107 for t in range(1, int(total_time / time_step)):  # 从第一个时
    刻开始
108     if collision_detected:
109         total_time = (t - 1) * time_step
110         break
111     # 计算龙头应当移动的距离 (time_step * velocity_head)
112     distance_to_travel = velocity_head * time_step
113
114     # 根据当前theta_head和距离求解下一个theta_head
115     theta_head = find_next_angle(theta_head,
```

```
    distance_to_travel, spiral_spacing)
116     prev_theta = theta_head
117 
118     # 更新龙头的位置
119     positions[0, :, t] = spiral_coordinates(theta_head)
120     print(theta_head)
121 
122     # 碰撞条件1（如果龙头过于进入螺线）
123     if theta_head <= 0 * np.pi:  # 设定一个最小碰撞距离
124         print(f"Collision detected at time {t * time_step:.1f}
    s, segment {i}")
125         collision_detected = True
126 
127     for i in range(1, num_segments):  # 从第一个节点开始，逐个
    计算
128         # 第一个节点的距离是 length_head，其他节点是
    length_body
129         segment_length = length_head if i == 1 else
    length_body
130         prev_position = positions[i - 1, :, t]  # 获取上一个节
    点的位置
131 
132         # 沿着螺旋线找到下一个符合距离要求的点
133         positions[i, :, t], prev_theta = find_next_position(
    prev_position, prev_theta, segment_length)
134 
135 
136 # 计算两个相邻点的中点坐标和它们所在直线的斜率
137 def calculate_midpoint_and_slope(p1, p2):
138     midpoint = (p1 + p2) / 2
139     delta_y = p2[1] - p1[1]
140     delta_x = p2[0] - p1[0]
141     if delta_x == 0:  # 防止除以零
142         slope = np.inf  # 无穷大表示垂直线
143     else:
```

```
        slope = delta_y / delta_x
    return midpoint, slope


# 创建用于保存中点和斜率的列表
midpoints_and_slopes = []

for t in range(int(total_time / time_step)):  # 遍历每个时间节点
    for i in range(num_segments - 1):  # 遍历每一对相邻点
        point1 = positions[i, :, t]
        point2 = positions[i + 1, :, t]

        midpoint, slope = calculate_midpoint_and_slope(point1, point2)

        # 添加数据到列表中
        midpoints_and_slopes.append([t * time_step, midpoint[0], midpoint[1], slope])

# 转换为DataFrame保存
columns = ["Time (s)", "Midpoint_X", "Midpoint_Y", "Slope"]
df_midpoints_and_slopes = pd.DataFrame(midpoints_and_slopes, columns=columns)

# 保存为CSV文件
csv_path = "four/midpoints_and_slopes.csv"
df_midpoints_and_slopes.to_csv(csv_path, index=False)
print(f"Midpoints and slopes saved to '{csv_path}'")

# 构造 CSV 文件的表头，增加时间列
headers = ["Time (s)"]
for i in range(num_segments):
    headers.append(f"Node_{i}_x")
    headers.append(f"Node_{i}_y")
```

```
175
176 # 保存位置矩阵为 .csv 文件
177 csv_path = r"four/positions.csv"
178 positions_reshaped = positions.reshape(num_segments * 2, -1).T
        # 转置以让每一行表示一个时间点
179
180 # 确保 total_time / time_step 是整数
181 num_rows_to_extract = int(total_time / time_step)
182
183 # 从 positions_reshaped 中提取前 num_rows_to_extract 行
184 positions_reshaped = positions_reshaped[:num_rows_to_extract,
      :]
185
186 # 创建正确的时间列，确保与 positions 的时间步数匹配
187 time_column = np.arange(0, total_time, time_step).reshape(-1,
      1)
188
189
190
191 # 将时间列与位置数据组合在一起
192 data_to_save = np.hstack((time_column, positions_reshaped))
193
194 # 写入 CSV 文件，并添加表头
195 np.savetxt(csv_path, data_to_save, delimiter=",", header=",".
      join(headers), comments='')
196 print(f"Positions saved as {csv_path}")
197
198 end = tim.time()
199 print('程序运行时间为: %s Seconds'%(end-start))
```

code1_calculateAngle.py

```
1 import numpy as np
2 from scipy.integrate import quad
3 from scipy.optimize import fsolve
4
```

```
# 等距螺线的螺距
b = 0.55 / np.pi  # 你可以根据具体螺线的螺距进行调整

# 给定起始角度A，目标弧长C
A = 16 * 2 * np.pi  # 起始角度
C = 0.01  # 目标弧长
# 100.53096491487338
# 100.52982860103113

# 定义弧长函数
def arc_length(theta, A, b):
    return b * np.sqrt(theta**2 + 1)

# 计算从角度A到角度B之间的弧长（顺时针，B < A）
def length_from_A_to_B(B, A, b):
    result, _ = quad(arc_length, B, A, args=(A, b))  # 注意：积分上下限反转，保证顺时针
    return result

# 目标函数，用于找到合适的角度B，使得顺时针弧长为C
def objective(B):
    return length_from_A_to_B(B, A, b) - C

# 使用fsolve求解角度B（负方向）
B_initial_guess = A - C / b  # 初始猜测值，向负方向走
B_solution = fsolve(objective, B_initial_guess)

print(f"起始角度 A = {A}")
print(f"顺时针走弧长为 {C} 米的终止角度 B = {B_solution[0]}")
```

code1/code1_calculateSpeed.py

```
import pandas as pd
import numpy as np

# Load the positions data
```

```
file = r'data\positions.csv'
df = pd.read_csv(file)
time_step = 0.01

# Assume the data has columns: "Time (s)", "Node_0_x", "Node_0_y", ..., "Node_223_x", "Node_223_y"
time_col = df['Time (s)'].values
node_columns = df.columns[1:]  # All node positions

# Time intervals around integer seconds (e.g., for 1 second: 0.9975 to 1.0025)
integer_times = np.arange(0, int(max(time_col)) + 1)
velocity_results = []

# Loop through each node and calculate velocity for integer seconds
for t in integer_times:
    # Find rows around the target time (e.g., 0.9975 to 1.0025 for t=1)
    time_mask = (time_col >= t - time_step) & (time_col <= t + time_step)
    time_subset = df[time_mask]

    if len(time_subset) < 2:
        continue  # Not enough data to calculate velocity

    # Calculate velocity for each node
    velocities = {}
    for i in range(0, len(node_columns), 2):  # Loop through x, y pairs
        node_x_col = node_columns[i]
        node_y_col = node_columns[i + 1]

        # Calculate velocity: delta position / delta time
        delta_x = time_subset[node_x_col].iloc[-1] -
```

```
    time_subset[node_x_col].iloc[0]
34          delta_y = time_subset[node_y_col].iloc[-1] -
    time_subset[node_y_col].iloc[0]
35          delta_time = time_subset['Time (s)'].iloc[-1] -
    time_subset['Time (s)'].iloc[0]
36
37          velocity = np.sqrt(delta_x ** 2 + delta_y ** 2) /
    delta_time
38          velocities[f'Node_{i//2}'] = velocity
39
40      # Save the velocities for this time step
41      velocities['Time (s)'] = t
42      velocity_results.append(velocities)
43
44  # Convert results to DataFrame and save to CSV
45  df_velocities = pd.DataFrame(velocity_results)
46  df_velocities.to_csv(r'output/velocities.csv', index=False)
47
48  print(f"Velocity data saved to 'velocities.csv'")
```

code1_caculateLength.py

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   from scipy.integrate import quad
4   from matplotlib import font_manager
5
6   # 设置中文字体，SimHei为黑体字体
7   plt.rcParams['font.sans-serif'] = ['SimHei']  # 设置字体为黑体
8   plt.rcParams['axes.unicode_minus'] = False  # 解决坐标轴负号显
    示问题
9
10  # 等距螺线的螺距
11  b = 55 / 100 / (2 * np.pi)  # 假设螺距为55厘米
12
13  # 给定起始角度A和终止角度B
```

```
A = 100.53096491487338  # 起始角度
# B = 100.16861407038287  # 计算得到的终止角度
B = 100.53210122

# 定义弧长函数
def arc_length(theta, b):
    return b * np.sqrt(theta**2 + 1)

# 计算从角度A到角度B之间的弧长
def length_from_A_to_B(A, B, b):
    result, _ = quad(arc_length, A, B, args=(b,))
    return result

# 验证弧长
calculated_length = length_from_A_to_B(A, B, b)
print(f"从角度 A = {A} 到角度 B = {B} 的弧长为: {
   calculated_length}")

# 计算螺旋线的坐标
def spiral_coordinates(theta, b):
    r = b * theta
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y

# 生成整个螺旋线的角度值
theta_full = np.linspace(0, A, 1000)

# 生成从A到B之间的角度值
theta_values = np.linspace(B, A, 100)  # 从B到A生成100个角度值

# 计算对应的坐标（全螺旋线）
x_full = []
y_full = []
for theta in theta_full:
```

```
    x, y = spiral_coordinates(theta, b)
    x_full.append(x)
    y_full.append(y)

# 计算对应的坐标（弧段）
x_values = []
y_values = []
for theta in theta_values:
    x, y = spiral_coordinates(theta, b)
    x_values.append(x)
    y_values.append(y)

# 绘制整个螺旋线
plt.figure(figsize=(8, 8))
plt.plot(x_full, y_full, 'gray', label="完整螺旋线", alpha=0.5)

# 绘制螺旋弧
plt.plot(x_values, y_values, label=f'弧长: {calculated_length:.2f} 米')
plt.scatter([x_values[0]], [y_values[0]], color='red', label='起点 A')
plt.scatter([x_values[-1]], [y_values[-1]], color='blue', label='终点 B')

# 设置图形属性
plt.title('螺旋线上的弧段')
plt.xlabel('X 坐标')
plt.ylabel('Y 坐标')
plt.legend()
plt.grid(True)
plt.axis('equal')  # 保持比例
plt.show()
```

code2.py

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

# 计算点在分离轴上的投影
def project_polygon(vertices, axis):
    min_proj = max_proj = dot_product(vertices[0], axis)
    for v in vertices[1:]:
        proj = dot_product(v, axis)
        min_proj = min(min_proj, proj)
        max_proj = max(max_proj, proj)
    return min_proj, max_proj

# 点积计算
def dot_product(v, axis):
    return v[0] * axis[0] + v[1] * axis[1]

# 检查两个区间是否重叠
def overlap(min1, max1, min2, max2):
    return not (max1 <= min2 or max2 <= min1)

# 计算矩形四个角点的位置
def rotate_point(px, py, cx, cy, angle):
    s = math.sin(angle)
    c = math.cos(angle)
    px -= cx
    py -= cy
    xnew = px * c - py * s
    ynew = px * s + py * c
    return xnew + cx, ynew + cy

# 生成旋转矩形的四个角点
def get_rotated_corners(x, y, w, h, angle):
    half_w = w / 2
```

```
    half_h = h / 2
    corners = [
        (x - half_w, y - half_h),
        (x + half_w, y - half_h),
        (x + half_w, y + half_h),
        (x - half_w, y + half_h)
    ]
    return [rotate_point(px, py, x, y, angle) for px, py in
   corners]

# 矩形碰撞检测
def is_colliding(rect1, rect2):
    x1, y1, w1, h1, angle1 = rect1
    x2, y2, w2, h2, angle2 = rect2

    corners1 = get_rotated_corners(x1, y1, w1, h1, angle1)
    corners2 = get_rotated_corners(x2, y2, w2, h2, angle2)

    def get_edges(corners):
        edges = []
        for i in range(len(corners)):
            p1 = corners[i]
            p2 = corners[(i + 1) % len(corners)]
            edge = (p2[0] - p1[0], p2[1] - p1[1])
            edges.append(edge)
        return edges

    def get_axes(corners):
        edges = get_edges(corners)
        axes = [(-edge[1], edge[0]) for edge in edges]
        return axes

    def check_overlap(poly1, poly2):
        axes = get_axes(poly1) + get_axes(poly2)
        for axis in axes:
```

```
            min1, max1 = project_polygon(poly1, axis)
            min2, max2 = project_polygon(poly2, axis)
            if not overlap(min1, max1, min2, max2):
                return False
        return True

    return check_overlap(corners1, corners2)




def calculate_rotation_angle(slope):
    return math.atan(slope)

def load_data(file_path):
    df = pd.read_csv(file_path)
    return df

def get_rectangles(df):
    rectangles = []
    for index, row in df.iterrows():
        time = row['Time (s)']
        slope = row['Slope']
        angle = calculate_rotation_angle(slope)
        # print(index)
        # Create a list of rectangles for the current time step
        if index % 223 == 0:
            rects = (row['Midpoint_X'], row['Midpoint_Y'], 3.41, 0.3, angle)  # First rectangle
        else:
            rects = (row['Midpoint_X'], row['Midpoint_Y'], 2.20, 0.3, angle)


```

```
102
103         # Append the time step and its rectangles to the list
104         rectangles.append((time, rects))
105
106     return rectangles
107
108 # def detect_collisions(rectangles):
109 #     for time, rects in rectangles:
110 #         num_rects = len(rects)
111 #         for i in range(num_rects):
112 #             rect_i = rects[i]
113 #             # Define the indices of rectangles to check for collisions
114 #             indices_to_check = set(range(num_rects)) - {i}  # All indices except 'i'
115
116 #             # Exclude adjacent rectangles (i-1 and i+1)
117 #             if i > 0:
118 #                 indices_to_check.discard(i - 1)
119 #             if i < num_rects - 1:
120 #                 indices_to_check.discard(i + 1)
121
122 #             # Check collisions with non-adjacent rectangles
123 #             for j in indices_to_check:
124 #                 rect_j = rects[j]
125 #                 if is_colliding(rect_i, rect_j):
126 #                     plot_rotated_rectangles(rect_i, rect_j)
127 #                     print(f"Time: {time}s, Collision between rectangle {i+1} and rectangle {j+1}")
128 #                     return
129
130 def get_rotated(x, y, width, height, angle):
131     # 计算角度的弧度值
132     theta = angle
133
```

```
    # 计算矩形的四个角点
    dx = width / 2
    dy = height / 2
    corners = np.array([
        [-dx, -dy],
        [dx, -dy],
        [dx, dy],
        [-dx, dy]
    ])

    # 旋转矩阵
    rotation_matrix = np.array([
        [np.cos(theta), -np.sin(theta)],
        [np.sin(theta), np.cos(theta)]
    ])

    # 应用旋转矩阵
    rotated_corners = corners @ rotation_matrix.T

    # 移动到矩形中心
    rotated_corners += np.array([x, y])

    return rotated_corners

def plot_rotated_rectangles(rectangles):
    # 创建图形和坐标轴
    fig, ax = plt.subplots()

    # 绘制所有矩形
    for rect in rectangles:
        x, y, width, height, angle = rect
        corners = get_rotated(x, y, width, height, angle)

        # 闭合矩形
        corners = np.append(corners, [corners[0]], axis=0)
```

```
169         ax.plot(corners[:, 0], corners[:, 1], color='blue')
170         ax.fill(corners[:, 0], corners[:, 1], color='blue',
    alpha=0.3)
171 
172     # 设置图形属性
173     ax.set_aspect('equal')
174     plt.grid(True)
175     plt.savefig("output.png", bbox_inches='tight')
176     plt.show()
177 
178 
179 
180 
181 
182 
183 
184 def detect_collisions(rectangles):
185     for count in range(1, len(df)//223 + 1):
186         print(count)
187         #print(range((count - 1) * 223, count * 223))
188         for index in range((count - 1) * 223, count * 223):
189             time, rect_i = rectangles[index][0], rectangles[
    index][1]
190 
191             # Define the indices of rectangles to check for
    collisions
192             indices_to_check = set(range((count - 1) * 223,
    count * 223)) - {index}  # All indices except 'i'
193 
194 
195             # Exclude adjacent rectangles (i-1 and i+1)
196             if index > (count - 1) * 223:
197                 indices_to_check.discard(index - 1)
198             if index < count * 223 - 1:
199                 indices_to_check.discard(index + 1)
```

```
            if count == 2:
                with open('output.txt', 'a') as f:
                    print(index,file = f)
                    print(indices_to_check, file = f)

            # Check collisions with non-adjacent rectangles
            for j in indices_to_check:
                if j > index:
                    rect_j = rectangles[j][1]
                    if is_colliding(rect_i, rect_j):
                        # recs = [res[1] for res in rectangles[(count - 1) * 223:count * 223]]
                        # plot_rotated_rectangles(recs)
                        print(f"Time: {time}s, Collision between rectangle {index+2} and rectangle {j+2}")
                        return



# Example usage
file_path = '/home/yanjy/python/motion/three_backup/midpoints_and_slopes_7_0.45039062500000004.csv'
df = load_data(file_path)
rectangles = get_rectangles(df)

detect_collisions(rectangles)
```

code3.py

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
from PIL import Image
from scipy.integrate import quad
```

```python
from scipy.optimize import fsolve
import time as tim
import math
import sys
with open("output_log.txt", "w") as fil:
        # 保存当前标准输出和标准错误
    original_stdout = sys.stdout
    original_stderr = sys.stderr

    # 重定向标准输出和标准错误到文件
    sys.stdout = fil
    sys.stderr = fil
    # 计算点在分离轴上的投影
    def project_polygon(vertices, axis):
        min_proj = max_proj = dot_product(vertices[0], axis)
        for v in vertices[1:]:
            proj = dot_product(v, axis)
            min_proj = min(min_proj, proj)
            max_proj = max(max_proj, proj)
        return min_proj, max_proj

    # 点积计算
    def dot_product(v, axis):
        return v[0] * axis[0] + v[1] * axis[1]

    # 检查两个区间是否重叠
    def overlap(min1, max1, min2, max2):
        return not (max1 <= min2 or max2 <= min1)

    # 计算矩形四个角点的位置
    def rotate_point(px, py, cx, cy, angle):
        s = math.sin(angle)
        c = math.cos(angle)
        px -= cx
        py -= cy
```

```
        xnew = px * c - py * s
        ynew = px * s + py * c
        return xnew + cx, ynew + cy


    # 生成旋转矩形的四个角点
    def get_rotated_corners(x, y, w, h, angle):
        half_w = w / 2
        half_h = h / 2
        corners = [
            (x - half_w, y - half_h),
            (x + half_w, y - half_h),
            (x + half_w, y + half_h),
            (x - half_w, y + half_h)
        ]
        return [rotate_point(px, py, x, y, angle) for px, py
in corners]



    # 矩形碰撞检测
    def is_colliding(rect1, rect2):
        x1, y1, w1, h1, angle1 = rect1
        x2, y2, w2, h2, angle2 = rect2

        corners1 = get_rotated_corners(x1, y1, w1, h1, angle1)
        corners2 = get_rotated_corners(x2, y2, w2, h2, angle2)

        def get_edges(corners):
            edges = []
            for i in range(len(corners)):
                p1 = corners[i]
                p2 = corners[(i + 1) % len(corners)]
                edge = (p2[0] - p1[0], p2[1] - p1[1])
                edges.append(edge)
```

```
            return edges

        def get_axes(corners):
            edges = get_edges(corners)
            axes = [(-edge[1], edge[0]) for edge in edges]
            return axes

        def check_overlap(poly1, poly2):
            axes = get_axes(poly1) + get_axes(poly2)
            for axis in axes:
                min1, max1 = project_polygon(poly1, axis)
                min2, max2 = project_polygon(poly2, axis)
                if not overlap(min1, max1, min2, max2):
                    return False
            return True

        return check_overlap(corners1, corners2)









    def calculate_rotation_angle(slope):
        return math.atan(slope)

    def load_data(file_path):
        df = pd.read_csv(file_path)
        return df



    def get_rectangles(df):
        rectangles = []
```

```
        for index, row in df.iterrows():
            time = row['Time (s)']
            slope = row['Slope']
            angle = calculate_rotation_angle(slope)
            # print(index)
            # Create a list of rectangles for the current time step
            if index % 223 == 0:
                rects = (row['Midpoint_X'], row['Midpoint_Y'], 3.41, 0.3, angle)  # First rectangle
            else:
                rects = (row['Midpoint_X'], row['Midpoint_Y'], 2.20, 0.3, angle)



            # Append the time step and its rectangles to the list
            rectangles.append((time, rects))

        return rectangles


    def detect_collisions(rectangles):
        for count in range(1, len(rectangles)//223 + 1):
            print(count)
            #print(range((count - 1) * 223, count * 223))
            for index in range((count - 1) * 223, count * 223):
                time, rect_i = rectangles[index][0], rectangles[index][1]

                # Define the indices of rectangles to check for collisions
                indices_to_check = set(range((count - 1) *
```

```
223, count * 223)) - {index}  # All indices except 'i'
139
140
141                 # Exclude adjacent rectangles (i-1 and i+1)
142                 if index > (count - 1) * 223:
143                     indices_to_check.discard(index - 1)
144                 if index < count * 223 - 1:
145                     indices_to_check.discard(index + 1)
146                 if count == 2:
147                     with open('output.txt', 'a') as f:
148                         print(index,file = f)
149                         print(indices_to_check, file = f)
150
151                 # Check collisions with non-adjacent
    rectangles
152                 for j in indices_to_check:
153                     if j > index:
154                         rect_j = rectangles[j][1]
155                         if is_colliding(rect_i, rect_j):
156                             # recs = [res[1] for res in
    rectangles[(count - 1) * 223:count * 223]]
157                             # plot_rotated_rectangles(recs)
158                             print(f"Time: {time}s, Collision
    between rectangle {index+2} and rectangle {j+2}")
159                             return count
160         return 0
161
162
163
164
165     def iter(spiral_spacing, theta_start, iteration):
166         start = tim.time()
167
168         # 初始化参数
169         num_segments = 224  # 总共224个把手
```

```
    length_head = 286 / 100  # 龙头前把手距离（米）
    length_body = 165 / 100  # 龙身和龙尾相邻把手距离（米）
    velocity_head = 1.0  # 龙头前把手速度（m/s）
    time_step = 0.1   # 时间步长（秒）
    total_time = 10000  # 定义总时间为700秒(碰撞会终止)
    # spiral_spacing = 55 / 100  # 螺距，单位为米
    search_step = 0.001  # 搜索步长，可以调小以提高精度

    # 创建用于保存图片的目录
    # save_dir = r'img'
    # os.makedirs(save_dir, exist_ok=True)

    # 初始化位置数组
    positions = np.zeros((num_segments, 2, int(total_time / time_step)))

    # 起始点为A点，θ_A = 16圈
    # theta_start = 16 * 2 * np.pi


    # 螺旋线方程（确保顺时针运动，从外圈向内圈，保持正的θ值）
    def spiral_coordinates(theta):
        r = spiral_spacing * theta / (2 * np.pi)
        x = r * np.cos(theta)  # 使用正θ来确保方向正确
        y = r * np.sin(theta)  # 使用正θ来确保方向正确
        return x, y


    # 计算两点之间的欧几里得距离
    def distance(point1, point2):
        return np.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] - point2[1]) ** 2)


```

```
202         # 找到下一个满足距离的点（逆时针查找）
203         def find_next_position(prev_position, prev_theta,
    segment_length):
204             # 沿着螺旋线搜索，增量步长可以根据精度需求调整
205             theta = prev_theta + search_step # theta 递增确保
    逆时针查找
206 
207             while True:
208                 # 计算当前位置的螺旋线坐标
209                 new_position = spiral_coordinates(theta)
210 
211                 # 计算当前点与上一个点的距离
212                 if distance(prev_position, new_position) >=
    segment_length:
213                     return new_position, theta
214 
215                 # 减小角度，沿螺旋线搜索
216                 theta += search_step  # theta 递增确保逆时针查
    找
217 
218 
219         # 计算等距螺线的弧长
220         def arc_length(theta, b):
221             return b * np.sqrt(theta ** 2 + 1)
222 
223 
224         # 根据给定的起点角度A和弧长，求解终止角度B
225         def find_next_angle(A, arc_length_to_travel, b):
226             b /= 2 * np.pi
227             def length_integral(B):
228                 result, _ = quad(arc_length, A, B, args=(b,))
229                 return result + arc_length_to_travel
230 
231             # 使用牛顿法或其他数值方法求解
232             B = fsolve(length_integral, A +
```

```
    arc_length_to_travel / b)
233             return B[0]
234
235
236         # 初始化龙头的位置，从A点开始 (theta_start)
237         current_position = spiral_coordinates(theta_start)
238         theta_head = theta_start
239
240         # 初始化龙头的位置，从A点开始 (theta_start)
241         current_position = spiral_coordinates(theta_start)
242         theta_head = theta_start
243
244         # 初始化 0 时刻的龙头位置
245         positions[0, :, 0] = current_position  # 记录龙头的初
    始位置
246
247         # 初始化 0 时刻的龙身位置
248         prev_theta = theta_start
249         for i in range(1, num_segments):  # 从第一个节点开始，
    逐个计算
250             # 第一个节点的距离是 length_head，其他节点是
    length_body
251             segment_length = length_head if i == 1 else
    length_body
252             prev_position = positions[i - 1, :, 0]  # 获取前一
    个节点的位置
253
254             # 沿着螺旋线找到下一个符合距离要求的点
255             positions[i, :, 0], prev_theta =
    find_next_position(prev_position, prev_theta, segment_length
    )
256
257         print("Initial positions at t=0 saved.")
258
259
```

```
260         # 递归计算其余把手的位置
261         collision_detected = False
262         for t in range(1, int(total_time / time_step)):  # 从
    第一个时刻开始
263             if collision_detected:
264                 total_time = (t - 1) * time_step
265                 break
266             # 计算龙头应当移动的距离 (time_step *
    velocity_head)
267             distance_to_travel = velocity_head * time_step
268 
269             # 根据当前theta_head和距离求解下一个theta_head
270             theta_head = find_next_angle(theta_head,
    distance_to_travel, spiral_spacing)
271             prev_theta = theta_head
272 
273             # 更新龙头的位置
274             positions[0, :, t] = spiral_coordinates(theta_head
    )
275             print(theta_head)
276 
277             # 碰撞条件1（如果龙头过于进入螺线）
278             if theta_head <= theta_start - 4 * np.pi:  # 设定
    一个最小碰撞距离
279                 print(f"Collision detected at time {t *
    time_step:.1f}s, segment {i}")
280                 collision_detected = True
281 
282             for i in range(1, num_segments):  # 从第一个节点开
    始，逐个计算
283                 # 第一个节点的距离是 length_head，其他节点是
    length_body
284                 segment_length = length_head if i == 1 else
    length_body
285                 prev_position = positions[i - 1, :, t]  # 获取
```

```
上一个节点的位置
286
287                 # 沿着螺旋线找到下一个符合距离要求的点
288                 positions[i, :, t], prev_theta =
find_next_position(prev_position, prev_theta, segment_length
)
289
290
291         # 计算两个相邻点的中点坐标和它们所在直线的斜率
292         def calculate_midpoint_and_slope(p1, p2):
293             midpoint = (p1 + p2) / 2
294             delta_y = p2[1] - p1[1]
295             delta_x = p2[0] - p1[0]
296             if delta_x == 0:  # 防止除以零
297                 slope = np.inf  # 无穷大表示垂直线
298             else:
299                 slope = delta_y / delta_x
300             return midpoint, slope
301
302         # 创建用于保存中点和斜率的列表
303         midpoints_and_slopes = []
304
305         for t in range(int(total_time / time_step)):  # 遍历每
个时间节点
306             for i in range(num_segments - 1):  # 遍历每一对相
邻点
307                 point1 = positions[i, :, t]
308                 point2 = positions[i + 1, :, t]
309
310                 midpoint, slope = calculate_midpoint_and_slope
(point1, point2)
311
312                 # 添加数据到列表中
313                 midpoints_and_slopes.append([t * time_step,
midpoint[0], midpoint[1], slope])
```

```
314
315         # 转换为DataFrame保存
316         columns = ["Time (s)", "Midpoint_X", "Midpoint_Y", "
    Slope"]
317         df_midpoints_and_slopes = pd.DataFrame(
    midpoints_and_slopes, columns=columns)
318
319         # 保存为CSV文件
320         csv_path = f"three/midpoints_and_slopes_{iteration}_{
    spiral_spacing}.csv"
321         df_midpoints_and_slopes.to_csv(csv_path, index=False)
322         print(f"Midpoints and slopes saved to '{csv_path}'")
323
324         # 构造 CSV 文件的表头，增加时间列
325         headers = ["Time (s)"]
326         for i in range(num_segments):
327             headers.append(f"Node_{i}_x")
328             headers.append(f"Node_{i}_y")
329
330         # 保存位置矩阵为 .csv 文件
331         csv_path = f"three/positions_{iteration}_{
    spiral_spacing}.csv"
332         positions_reshaped = positions.reshape(num_segments *
    2, -1).T  # 转置以让每一行表示一个时间点
333
334         # 确保 total_time / time_step 是整数
335         num_rows_to_extract = int(total_time / time_step)
336
337         # 从 positions_reshaped 中提取前 num_rows_to_extract
    行
338         positions_reshaped = positions_reshaped[:
    num_rows_to_extract, :]
339
340         # 创建正确的时间列，确保与 positions 的时间步数匹配
341         time_column = np.arange(0, total_time, time_step).
```

```
    reshape(-1, 1)

        # # 打印数组形状
        # print(f"positions_reshaped shape: {positions_reshaped.shape}")
        # print(f"time_column shape: {time_column.shape}")
        #
        # # 打印修正后的时间列形状
        # print(f"Corrected time_column shape: {time_column.shape}")

        # 将时间列与位置数据组合在一起
        data_to_save = np.hstack((time_column, positions_reshaped))

        # 写入 CSV 文件，并添加表头
        np.savetxt(csv_path, data_to_save, delimiter=",", header=",".join(headers), comments='')
        print(f"Positions saved as {csv_path}")


        df = load_data(f"three/midpoints_and_slopes_{iteration}_{spiral_spacing}.csv")
        rectangles = get_rectangles(df)

        count = detect_collisions(rectangles)
        if count == 0:
            return True
        df = load_data(f"three/positions_{iteration}_{spiral_spacing}.csv")
        x = df.iloc[count - 1,1]
        y = df.iloc[count - 1,2]
        if x**2 + y**2 - 4.5 * 4.5 < 1e-6:
            return True
        else :
```

```
            return False
        end = tim.time()
        print('程序运行时间为: %s Seconds'%(end-start))

    iteration = 1
    max_spi = 0.55
    min_spi = 0.30
    current_spi = (max_spi + min_spi) / 2
    while max_spi - min_spi > 1e-3:
        print(f"[{min_spi},{max_spi}],current_spi is : {current_spi}")
        if iter(current_spi, (4.5 / current_spi + 1) * 2 * np.pi, iteration):
            max_spi = current_spi
            current_spi = (max_spi + min_spi) / 2
        else:
            min_spi = current_spi
            current_spi = (max_spi + min_spi) / 2
        iteration += 1


    print(current_spi)
    with open('result.txt','w') as f:
        print(max_spi, file = f)
```

code4.py

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve
from scipy.integrate import ode45

# 参数设置
pitch = 1.7  # 螺距
k = pitch / (2 * np.pi)  # 螺线方程系数
len1 = 3.41  # 第一段长度（米）
```

```
dist1 = len1 - 0.55  # 第一个凳子把手之间的距离
len2 = 2.20  # 其他凳子长度（米）
dist2 = len2 - 0.55  # 其他凳子把手之间的距离
v_head = 1  # 头节点速度

# 螺线参数
theta_in = np.linspace(5 * 2 * np.pi, 0, 500)
r_in = k * theta_in
x_in = r_in * np.cos(theta_in)
y_in = r_in * np.sin(theta_in)

# 旋转180度后螺线
theta_out = theta_in - np.pi
r_out = k * (theta_out + np.pi)
x_out = r_out * np.cos(theta_out)
y_out = r_out * np.sin(theta_out)

# 绘制螺线
plt.figure(figsize=(6, 6))
plt.plot(x_in, y_in, 'r-', label="盘入螺线")
plt.plot(x_out, y_out, 'b-', label="盘出螺线")
plt.grid(True)
plt.gca().set_aspect('equal', adjustable='box')

# 定义掉头区域
turn_radius = 4.283
x_turn = turn_radius * np.cos(theta_in)
y_turn = turn_radius * np.sin(theta_in)
plt.plot(x_turn, y_turn, 'g-', linewidth=2, label="掉头区域")

# 求解螺线与调头圆形区域的交点角度和切线斜率
theta_in_cross = turn_radius / k
theta_out_cross = theta_in_cross - np.pi
slope = (k * np.sin(theta_in_cross) + turn_radius * np.cos(
    theta_in_cross)) / \
```

```
44          (k * np.cos(theta_in_cross) - turn_radius * np.sin(
    theta_in_cross))
45
46  # 求解C1和C2两个圆的几何关系
47  theta_max_1 = np.arctan(-1 / slope) + np.pi
48  theta_turn = np.arctan(np.tan(theta_in_cross)) + np.pi -
    theta_max_1
49  rC1_C2 = turn_radius / np.cos(theta_turn)
50  rC2 = rC1_C2 / 3
51  rC1 = 2 * rC2
52
53  # 圆心坐标
54  xC1 = turn_radius * np.cos(theta_in_cross) + rC1 * np.cos(
    theta_max_1 - np.pi)
55  yC1 = turn_radius * np.sin(theta_in_cross) + rC1 * np.sin(
    theta_max_1 - np.pi)
56  xC2 = turn_radius * np.cos(theta_out_cross) - rC2 * np.cos(
    theta_max_1 - np.pi)
57  yC2 = turn_radius * np.sin(theta_out_cross) - rC2 * np.sin(
    theta_max_1 - np.pi)
58
59  # 绘制圆弧
60  theta_C1_range = np.linspace(theta_max_1 - np.pi, theta_max_1,
     50)
61  theta_C2_range = np.linspace(theta_max_1 - np.pi, theta_max_1,
     50)
62  plt.plot(xC1 + rC1 * np.cos(theta_C1_range), yC1 + rC1 * np.
    sin(theta_C1_range), 'r', linewidth=2)
63  plt.plot(xC2 + rC2 * np.cos(theta_C2_range), yC2 + rC2 * np.
    sin(theta_C2_range), 'b', linewidth=2)
64  plt.plot(xC1, yC1, 'r*')
65  plt.plot(xC2, yC2, 'b*')
66
67  plt.legend()
68  plt.show()
```

```
69
70  # 定义子函数，计算螺线上的下一个点的位置
71  def solve_next_theta(pitch, x1, y1, theta1, d):
72      k = pitch / (2 * np.pi)
73      fun = lambda theta: (k * theta * np.cos(theta) - x1) ** 2
        + (k * theta * np.sin(theta) - y1) ** 2 - d ** 2
74      theta = fsolve(fun, theta1 + 0.01)[0]
75      return theta
```

code5.py

```
1   import numpy as np
2   import matplotlib.pyplot as plt
3   from scipy.optimize import fsolve
4   from scipy.integrate import ode45
5
6   # 参数设置
7   pitch = 1.7  # 螺距
8   k = pitch / (2 * np.pi)  # 螺线方程系数
9   len1 = 3.41  # 第一段长度（米）
10  dist1 = len1 - 0.55  # 第一个凳子把手之间的距离
11  len2 = 2.20  # 其他凳子长度（米）
12  dist2 = len2 - 0.55  # 其他凳子把手之间的距离
13  v_head = 1  # 头节点速度
14
15  # 螺线参数
16  theta_in = np.linspace(5 * 2 * np.pi, 0, 500)
17  r_in = k * theta_in
18  x_in = r_in * np.cos(theta_in)
19  y_in = r_in * np.sin(theta_in)
20
21  # 旋转180度后螺线
22  theta_out = theta_in - np.pi
23  r_out = k * (theta_out + np.pi)
24  x_out = r_out * np.cos(theta_out)
25  y_out = r_out * np.sin(theta_out)
```

```
26
27  # 绘制螺线
28  plt.figure(figsize=(6, 6))
29  plt.plot(x_in, y_in, 'r-', label="盘入螺线")
30  plt.plot(x_out, y_out, 'b-', label="盘出螺线")
31  plt.grid(True)
32  plt.gca().set_aspect('equal', adjustable='box')
33
34  # 定义掉头区域
35  turn_radius = 4.283
36  x_turn = turn_radius * np.cos(theta_in)
37  y_turn = turn_radius * np.sin(theta_in)
38  plt.plot(x_turn, y_turn, 'g-', linewidth=2, label="掉头区域")
39
40  # 求解螺线与调头圆形区域的交点角度和切线斜率
41  theta_in_cross = turn_radius / k
42  theta_out_cross = theta_in_cross - np.pi
43  slope = (k * np.sin(theta_in_cross) + turn_radius * np.cos(
        theta_in_cross)) / \
44          (k * np.cos(theta_in_cross) - turn_radius * np.sin(
        theta_in_cross))
45
46  # 求解C1和C2两个圆的几何关系
47  theta_max_1 = np.arctan(-1 / slope) + np.pi
48  theta_turn = np.arctan(np.tan(theta_in_cross)) + np.pi -
        theta_max_1
49  rC1_C2 = turn_radius / np.cos(theta_turn)
50  rC2 = rC1_C2 / 3
51  rC1 = 2 * rC2
52
53  # 圆心坐标
54  xC1 = turn_radius * np.cos(theta_in_cross) + rC1 * np.cos(
        theta_max_1 - np.pi)
55  yC1 = turn_radius * np.sin(theta_in_cross) + rC1 * np.sin(
        theta_max_1 - np.pi)
```

```
xC2 = turn_radius * np.cos(theta_out_cross) - rC2 * np.cos(
    theta_max_1 - np.pi)
yC2 = turn_radius * np.sin(theta_out_cross) - rC2 * np.sin(
    theta_max_1 - np.pi)

# 绘制圆弧
theta_C1_range = np.linspace(theta_max_1 - np.pi, theta_max_1,
     50)
theta_C2_range = np.linspace(theta_max_1 - np.pi, theta_max_1,
     50)
plt.plot(xC1 + rC1 * np.cos(theta_C1_range), yC1 + rC1 * np.
    sin(theta_C1_range), 'r', linewidth=2)
plt.plot(xC2 + rC2 * np.cos(theta_C2_range), yC2 + rC2 * np.
    sin(theta_C2_range), 'b', linewidth=2)
plt.plot(xC1, yC1, 'r*')
plt.plot(xC2, yC2, 'b*')

plt.legend()
plt.show()

# 定义子函数，计算螺线上的下一个点的位置
def solve_next_theta(pitch, x1, y1, theta1, d):
    k = pitch / (2 * np.pi)
    fun = lambda theta: (k * theta * np.cos(theta) - x1) ** 2
    + (k * theta * np.sin(theta) - y1) ** 2 - d ** 2
    theta = fsolve(fun, theta1 + 0.01)[0]
    return theta
```