基于“板凳龙”等距螺线模型的路径优化问题

摘　要

模拟“板凳龙”舞龙队沿特定路径的运动，并计算其在规定时间内的位置和速度。本研究的核心是构建一个精确的数学模型，以模拟舞龙队在 300 秒内沿等距螺线的运动。我们采用了基于螺距为 55 厘米的等距螺线方程，并假设龙头前把手以恒定速度 1 米/秒运动。

为了实现这一目标，我们首先建立了一个数学模型，该模型考虑了舞龙队的整体结构和运动特性。我们使用了数值微分方程求解器（如 MATLAB 中的 ode45）来计算龙头在每个时间步长的位置。通过迭代计算，我们得到了舞龙队在 300 秒内每个时间点的位置和速度数据。

此外，我们还特别关注了特定时间点的详细分析，包括龙头前把手以及龙头后面特定几节龙身前把手和龙尾后把手的位置和速度。这些数据对于评估舞龙队运动的动态特性至关重要。

为了确保模型的实用性和准确性，我们实现了碰撞检测算法，确保在模拟过程中板凳之间不会发生碰撞。这一算法通过计算相邻板凳之间的距离，并与预设的安全距离进行比较，从而确保了模拟的安全性。

最终，我们将所有计算结果整理并保存至 Excel 文件中，以便于进一步分析和验证。例如，第二问我们通过分离轴定理算出答案为 412.6; 在第 4 问中, 我们采取解析几何和微分的方法, 最后通过枚举筛选出答案为 13.48. 我们的研究不仅为理解和模拟传统民俗活动提供了新的视角，也为相关领域的研究提供了科学的方法和工具。

关键词：舞龙队　等距螺线　数值微分方程　碰撞检测　动态模拟

一、问题重述

1.1 问题背景

在浙江省和福建省的部分地区，一种名为“板凳龙”的传统民俗活动代代相传，以其独特的魅力深受当地人民的喜爱。这项活动的核心是将数十甚至上百条长板凳通过首尾相接的方式串联起来，塑造出一条条蜿蜒盘旋、栩栩如生的“龙”。在表演过程中，龙头引领着整个队伍，而龙身和龙尾则紧随其后，以协调的动作盘旋起舞，形成一幅壮观的圆盘状图景。这种表演不仅考验着舞龙者的技巧和团队协作，同时也是对传统文化的一种传承和展示。为了提升观赏性，舞龙队在表演时力求在尽可能小的空间内，以最快的速度完成盘旋动作，这就涉及到数学建模，以优化表演的路径和节奏，使得整个表演过程既紧凑又富有动感。

1.2 问题回顾

问题 1：在龙头前把手速度固定 1 m/s 的情况下，舞龙队沿螺距为 55 cm 的等距螺线顺时针盘入时，建立数学模型，计算从初始时刻到 300 s 为止每秒整个舞龙队的位置和速度。

问题 2：舞龙队沿问题 1 设定的螺线盘入，建立数学模型，确定板凳之间不发生碰撞的条件下舞龙队盘入的终止时刻，并给出此时舞龙队的位置和速度。

问题 3：舞龙队由顺时针盘入调头切换为逆时针盘出，建立数学模型，确定在已知以螺线中心为圆心、直径为 9 m 的圆形区域掉头空间内，所需的最小螺距。

问题 4：在一个调头空间内，设计一条由两段相切圆弧组成的 S 形曲线调头路径，要求与盘入螺线（螺距 1.7 米）和盘出螺线相切，且前一段圆弧的半径是后一段的 2 倍，建立数学模型，以求得调头路径的最短长度。

问题 5：舞龙队沿着问题 4 设定的路径行进，保持龙头的行进速度不变，建立数学模型，确定龙头的最大行进速度，以确保舞龙队各把手的速度均不超过 2 m/s。

二、问题分析

2.1 问题一分析

问题一要求模拟一个舞龙队沿等距螺线的运动，使用 MATLAB 建立极坐标下的等距螺线模型，通过已知龙头把手零时刻位置，利用 Runge-Kutta method 来求解描述把手运动的微分方程，计算在不同时间点上把手的位置，转化为笛卡尔坐标系下坐标，再通过相邻把手之间有相等的距离递推求得所有把手的位置。再根据龙头把手固定的运动速度，利用板约束模型，沿板方向速度分量相等的特性，递推得到所有把手的速度。由此可以计算和记录特定时间点上每个部分（龙头、龙身各节、龙尾）的位置和速度。

2.2 问题二分析

对于问题二，通过第一问的求解得到 300 s 时龙头把手的位置，在此基础上继续盘入。首先需要得到碰撞条件，对时间进行迭代，计算孔的分布，应用碰撞条件对每个时刻进行碰撞检验，当发生第一次重叠时，停止迭代运算，返回此时时刻，以及每个孔所在的位置和速度。

2.3 问题三分析

针对问题三，首先以 55 厘米为初始螺距生成螺旋线，并逐步减小螺距。根据掉头区间的尺寸和位置，计算出龙头前把手的目标位置。路径生成的过程中，在每一步计算新位置时进行碰撞检测，确保路径不会重叠，每次调整都需遵循最小螺距约束，防止路径过于紧密。当龙头前把手到达掉头区间边界且无碰撞时，记录此时的最小螺距，作为该路径规划的最小允许值。

2.4 问题四分析

采用数值法,先将 theta_1 和 r_1 确定范围， 再分为很多小分。进行解析几何模型的建立与求解计算， 最终枚举筛选出结果。

2.5 问题五分析

在舞龙队表演中，龙头的速度决定了整个队伍的运动速度。为了确保表演的安全性和观赏性，需要控制各把手的速度不超过一定的阈值（本问题中为 2 m/s）。这要求我们对龙头的速度进行优化，以满足所有把手的速度限制。

三、模型假设

为了方便模型的建立与模型的可行性，我们这里首先对模型提出一些假设，使得模型更加完备，预测的结果更加合理。

1、假设每节板凳之间的连接是刚性的，即连接点不会发生任何形变，且连接的长度固定不变。

2、假设舞龙队运动的螺线是理想的等距螺线，即每圈的间距完全相同。

3、将问题简化为二维平面内的运动，忽略任何垂直于运动平面的方向。

4、

这些假设有助于简化问题，使得可以通过数学和物理方法进行分析和计算。然而，它们也可能限制了模型的适用范围和准确性。在实际应用中，可能需要根据具体情况调整这些假设，以更准确地反映实际情况。

四、符号说明

为了方便我们模型的建立与求解过程，我们这里对使用到的关键符号进行以

下说明：

符号	符号说明
x_i	第i个孔横
k_i	第i个孔的切线斜率
$k_{i,i+1}$	相邻两孔连线斜率
v_i	第i个孔的速度
θ_{ij}	两板凳中垂线夹角

（注：这里只列出论文各部分通用符号，个别模型单独使用的符号在首次引用时会进行说明。）

五、模型的建立与求解

5.1 问题一模型的建立与求解

5.1.1 问题一模型的建立

阿基米德螺线方程为

$$r = b\theta \tag{1}$$

其中$b = \frac{0.55}{2\pi}$

$$ds = \sqrt{((dr/d\theta)^2 + r^2)}\,d\theta = \sqrt{(b^2 + (b\theta)^2)}\,d\theta \tag{2}$$

$$s = \int_0^{\theta_0} \sqrt{(b^2 + (b\theta)^2)}\,d\theta = v_0 t \tag{3}$$

龙头前把手随时间 t 的位置

$$A_1\begin{cases} x_1 = b\theta_1 cos\theta_1 \\ y_1 = b\theta_1 sin\theta_1 \end{cases}$$

$$A_2\begin{cases} x_2 = b\theta_2 cos\theta_2 \\ y_1 = b\theta_2 sin\theta_2 \end{cases} \tag{4}$$

对于任意一点的切线斜率

$$k_i = \frac{sin\theta_i + \theta_i cos\theta_i}{cos\theta_i - \theta_i sin\theta_i} = tan\varphi_i \tag{5}$$

相邻两把手间直线斜率

$$k_{i,i+1}=\frac{\theta_i sin\theta_i-\theta_{i+1}sin\theta_{i+1}}{\theta_i cos\theta_i-\theta_{i+1}cos\theta_{i+1}}=tan\varphi_{i,i+1} \tag{6}$$

由板约束模型，沿杆方向速度分量相等，且已知龙头把手$v=1m/s$，
最终得

$$v_{i+1}=v_i\frac{\cos(\varphi_i-\varphi_{i,i+1})}{\cos(\varphi_{i,i+1}-\varphi_{i+1})} \tag{7}$$

5.1.2 模型的求解

在 MATLAB 利用推得的公式直接求解即可。

5.1.3 问题一结果

表 1：部分把手的位置

	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	2.459488
第 1 节龙身 y (m)	2.826544	-3.440399	-7.405883	4.359627	-3.561948	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	1.898795	-4.917371	-6.237723
第 101 节龙身 y (m)	-9.918311	-8.001384	-7.557638	-8.471614	-6.379875	3.936007
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	4.393014
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	-6.180727	-5.263385
龙尾（后）x (m)	-5.305444	7.364557	10.974348	7.383895	3.241051	1.785032
龙尾（后）y (m)	-10.676584	-8.797992	0.843473	7.492371	9.469337	9.301164

表 2：部分把手的位置和速度

	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)	0.000000	0.999961	0.999945	0.999917	0.999859	0.999709
第 51 节龙身 (m/s)	0.000000	0.999662	0.999538	0.999331	0.998941	0.998065
第 101 节龙身 (m/s)	0.000000	0.999453	0.999269	0.998971	0.998435	0.997302
第 151 节龙身 (m/s)	0.000000	0.999299	0.999078	0.998727	0.998115	0.996861
第 201 节龙身 (m/s)	0.000000	0.999180	0.998935	0.998551	0.997894	0.996574
龙尾（后）(m/s)	0.000000	0.999136	0.998883	0.998489	0.997816	0.996478

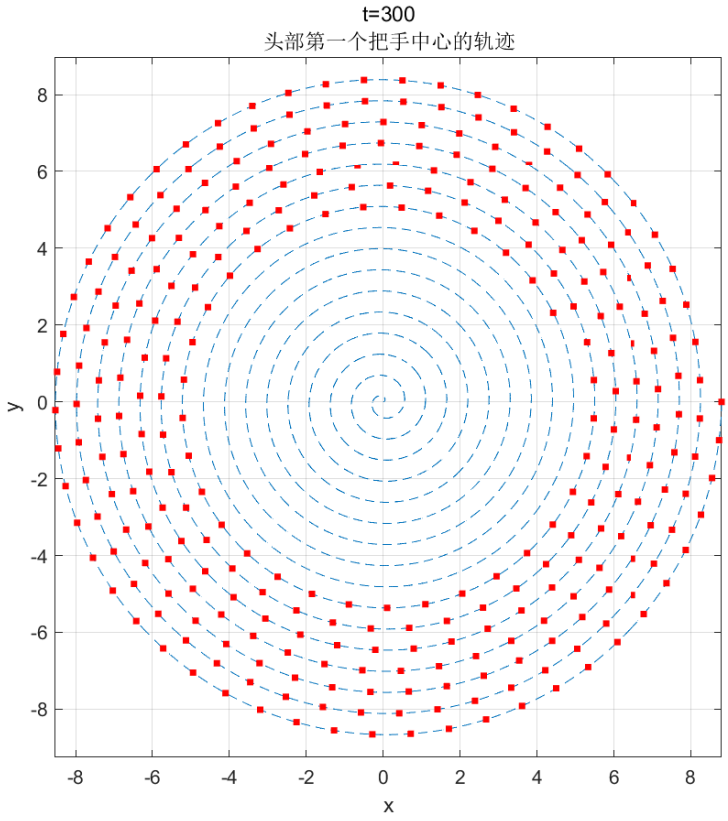
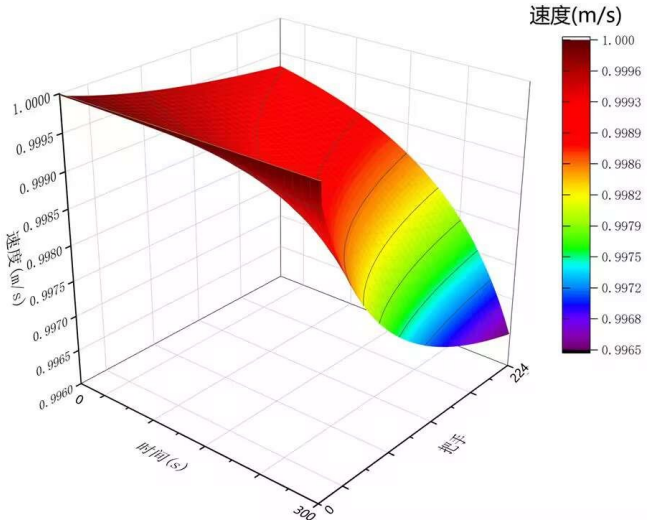


图 1：头部第一个把手中心的轨迹



图二：速度的 3D 图

5.2 问题二模型的建立与求解

5.2.1 模型的建立

(1)碰撞检验模型

使用分离轴定理检验碰撞，即为对两个凸多边形 A 和 B，它们没有交点，那么必然存在至少一条直线可以将它们完全分开，使得 A 的所有点都在直线上的一侧，而 B 的所有点都在另一侧。这条直线就是所谓的分离轴。如果对于任意一对凸多边形，我们都能够找到这样一条分离轴，那么就可以断定这两个凸多边形不相交。以下讲解举龙头为例，检验龙头和不相邻龙身的碰撞，如下图：

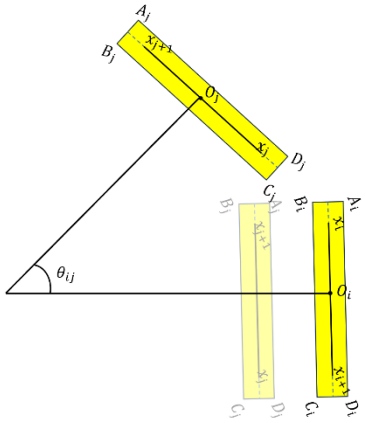


图 3：两个板凳间的碰撞检验

对第i块板作为基准板，即垂直 x 轴，由前后两孔的中心坐标x_i、x_{i+1}，计算得板凳中心坐标

$$O_i:\left(x_i+x_{i+1}/2,\ 0\right) \tag{8}$$

作为碰撞检验的第j块板凳中心坐标

$$O_j:\left(x_j+x_{j+1}/2,y_j+y_{j+1}/2\right) \tag{9}$$

通过计算中垂线斜率可得第i块板和第j块板中垂线夹角θ_{ij}

并将第j块板做点离散化，均匀分成 400 个点。由板的斜率

$$k_{j,j+1}=\frac{\theta_j sin\theta_j-\theta_{j+1}sin\theta_{j+1}}{\theta_j cos\theta_j-\theta_{j+1}cos\theta_{j+1}}=tan\varphi_{j,j+1}$$

对于$x_j<x_{j+1}$

$$A_j:\begin{cases}x_{Aj}=x_j-0.275cos\varphi_{i,i+1}-0.15sin\varphi_{j,j+1}\\ y_{Aj}=y_j-0.275sin\varphi_{i,i+1}+0.15cos\varphi_{j,j+1}\end{cases}$$

$$B_j:\begin{cases}x_{Bj}=x_j-0.275cos\varphi_{j,j+1}+0.15sin\varphi_{j,j+1}\\ y_{Bj}=y_j-0.275sin\varphi_{j,j+1}-0.15cos\varphi_{j,j+1}\end{cases}$$

$$C_j:\begin{cases}x_{Cj}=x_{j+1}+0.275cos\varphi_{j,j+1}-0.15sin\varphi_{j,j+1}\\ y_{Cj}=y_{j+1}+0.275sin\varphi_{j,j+1}+0.15cos\varphi_{j,j+1}\end{cases}$$

$$D_j:\begin{cases}x_{Dj}=x_{j+1}+0.275cos\varphi_{j,j+1}-0.15sin\varphi_{j,j+1}\\ y_{Dj}=y_{j+1}+0.275sin\varphi_{j,j+1}+0.15cos\varphi_{j,j+1}\end{cases}$$

对于$x_j>x_{j+1}$

$$A_j:\begin{cases}x_{Aj}=x_{j+1}-0.275cos\varphi_{i,i+1}-0.15sin\varphi_{j,j+1}\\y_{Aj}=y_{j+1}-0.275sin\varphi_{i,i+1}+0.15cos\varphi_{j,j+1}\end{cases}$$
$$B_j:\begin{cases}x_{Bj}=x_{j+1}-0.275cos\varphi_{j,j+1}+0.15sin\varphi_{j,j+1}\\y_{Bj}=y_{j+1}-0.275sin\varphi_{j,j+1}-0.15cos\varphi_{j,j+1}\end{cases}$$
$$C_j:\begin{cases}x_{Cj}=x_j+0.275cos\varphi_{j,j+1}-0.15sin\varphi_{j,j+1}\\y_{Cj}=y_j+0.275sin\varphi_{j,j+1}+0.15cos\varphi_{j,j+1}\end{cases}$$
$$D_j:\begin{cases}x_{Dj}=x_j+0.275cos\varphi_{j,j+1}-0.15sin\varphi_{j,j+1}\\y_{Dj}=y_j+0.275sin\varphi_{j,j+1}+0.15cos\varphi_{j,j+1}\end{cases}$$

可得这 400 个点的坐标，将第j块板旋转$^{\theta_{ij}}$与第^{i}块板平行。由第^{i}块板的坐标范围

$$|x-x_{oi}|\ll 15(cm)$$
$$|y-yo_i|\ll 341/2(cm)$$

则由旋转矩阵

$$T=\begin{bmatrix}cos\theta_{ij} & -sin\theta_{ij}\\ sin\theta_{ij} & cos\theta_{ij}\end{bmatrix}$$

对第j块板的坐标做变换

$$\begin{pmatrix}x'\\y'\end{pmatrix}=\begin{bmatrix}cos\theta_{ij} & -sin\theta_{ij}\\ sin\theta_{ij} & cos\theta_{ij}\end{bmatrix}\begin{pmatrix}x\\y\end{pmatrix}$$

判断离散的 400 个点是否落在第^{i}块板的坐标范围内。

$$\begin{cases}|x-x_{oi}|\ll 15\\|y-yo_i|\ll 341/2\end{cases}$$

5.2.2 模型的求解

将上述分离轴定理的建模编写成 MATLAB 代码，对时间进行迭代，计算孔的分布，应用碰撞条件对每个时刻进行碰撞检验，当发生第一次重叠时，停止迭代运算，返回此时时刻，以及每个孔所在的位置和速度。

5.2.3 结果

表 3：412.6s 时部分把手速度

	412.6s
龙头 (m/s)	1.000000
第 1 节龙身 (m/s)	0.991466
第 51 节龙身 (m/s)	0.976700
第 101 节龙身 (m/s)	0.974389
第 151 节龙身 (m/s)	0.973446
第 201 节龙身 (m/s)	0.972934
龙尾（后） (m/s)	0.972776

5.3 问题三模型的建立与求解

5.3.1 模型的建立

设定螺距的初始值为 55 厘米，并逐步减小螺距。使用极坐标方程来描述等距螺线：

$$r = b \cdot \theta \tag{10}$$

其中，

$$b = \frac{\text{螺距}}{2\pi} \tag{11}$$

通过不断调整螺距，生成不同的螺旋线。需要根据调头空间的尺寸和位置，计算龙头前把手的目标位置。在每一步生成新路径时，进行碰撞检测以确保板凳之间没有重叠，并确保路径满足最小螺距的约束条件。每次调整都需要检查路径是否符合条件，即板凳不相碰且路径不会过于紧密。当龙头前把手到达调头区间边界且无碰撞时，记录此时的最小螺距，作为该路径规划的最小允许值。

5.3.2 模型的求解

使用 MATLAB 进行数值模拟。通过调整螺距参数，逐步计算龙头前把手的位置，直到其到达调头区边界。在每一步计算中，检测是否满足无碰撞条件，并逐步缩小螺距，直到找到满足条件的最小螺距。

5.3.3 结果

通过上述模型，得到了舞龙队伍在顺时针盘入到调头空间边界所需的最小螺距为 42.1 cm。该螺距确保了舞龙队伍在进行调头时，不会发生任何碰撞，并能够顺利进入调头空间。

5.4 问题四模型的建立与求解

5.4.1 模型的建立

螺线方程

$$\rho = a \cdot \theta$$

对应的直角坐标方程

$$x = a \cdot \theta \cos\theta$$

$$y = a \cdot \theta \sin\theta$$

其导数为

$$\frac{\mathrm{d}x}{\mathrm{d}\theta} = a\ (\cos\theta\ -\ \theta\ \sin\theta)$$

$$\frac{\mathrm{d}y}{\mathrm{d}\theta} = a\ (\sin\theta\ +\ \theta\ \cos\theta)$$

因此, 延螺线方向的切向量的法线方向单位向量为

$$\vec{T} = \frac{\left(-\frac{\mathrm{d}y}{\mathrm{d}\theta}, \frac{\mathrm{d}x}{\mathrm{d}\theta}\right)}{\left|\left(-\frac{\mathrm{d}y}{\mathrm{d}\theta}, \frac{\mathrm{d}x}{\mathrm{d}\theta}\right)\right|} = \frac{(\ \sin\theta\ +\ \theta\ \cdot\ \cos\theta, \cos\theta\ -\ \theta\ \cdot\ \sin\theta)}{(\sin\theta\ +\ \theta\ \cdot\ \cos\theta)^2 + (\cos\theta\ -\ \theta\ \cdot\ \sin\theta)^2}$$

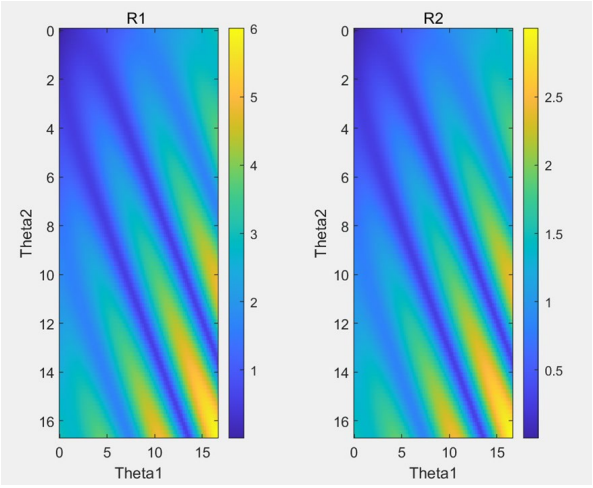
设切点为 (x,y), 其对应的与螺线相切的圆的半径为 R, 则圆心为 $(x,y) + \vec{T}$.

由于半径有 2 倍关系, 且 x_2 关于原点对称, 因此

$$(x_1 - (-x_2))^2 + (y_1 - (-y_2))^2 = (3\ \cdot\ r_2)^2$$

5.4.2 模型的求解

将上述计算圆弧位置和方程的公式写为 MATLAB 代码，计算得一系列可能的 s 型圆弧方程，并带入上文提及的碰撞检验，即可求得最短的 s 型圆弧长度及其位置。



如图是对盘入和盘出螺线所取的每组θ_1和θ_2对应的满足半径关系 2 倍的圆弧半

径热度图。

5.4.3 结果

表 5：部分把手的位置

	-100 s	-50 s	0 s	50 s	100 s
龙头 x(m)	8.503941	6.553096	-4.248714	4.076716	0.006748
龙头 y(m)	0.348817	-1.562022	-0.590324	4.644781	8.074771
第 1 节龙身 x(m)	7.989437	6.725194	-3.176110	5.628465	2.826317
第 1 节龙身 y(m)	3.190113	1.312998	-3.131987	2.223662	7.459531
第 51 节龙身 x(m)	10.200000	-5.820768	-0.232925	-4.297760	4.235754
第 51 节龙身 y(m)	0.000000	-7.632433	-8.063824	-4.421895	0.710841
第 101 节龙身 x(m)	10.200000	10.200000	10.200000	-6.064715	-6.324483
第 101 节龙身 y(m)	0.000000	0.000000	0.000000	6.813674	4.025013
第 151 节龙身 x(m)	10.200000	10.200000	10.200000	10.200000	8.910162
第 151 节龙身 y(m)	0.000000	0.000000	0.000000	0.000000	-4.689575
第 201 节龙身 x(m)	10.200000	10.200000	10.200000	10.200000	10.200000
第 201 节龙身 y(m)	0.000000	0.000000	0.000000	0.000000	0.000000
龙尾（后）x(m)	10.200000	10.200000	10.200000	10.200000	10.200000
龙尾（后）y(m)	0.000000	0.000000	0.000000	0.000000	0.000000

表 6：部分把手的位置和速度

	-100 s	-50 s	0 s	50 s	100 s
龙头(m/s)	0.985784	0.977211	0.942738	0.972862	0.984194
第 1 节龙身(m/s)	0.995391	0.992725	0.982747	0.990664	0.994638
第 51 节龙身(m/s)	0.000000	0.996300	0.994757	0.991010	0.981378
第 101 节龙身(m/s)	0.000000	0.000000	0.000000	0.995902	0.993926
第 151 节龙身(m/s)	0.000000	0.000000	0.000000	0.000000	0.996638
第 201 节龙身(m/s)	0.000000	0.000000	0.000000	0.000000	0.000000
龙尾（后）(m/s)	0.000000	0.000000	0.000000	0.000000	0.000000

5.5 问题五模型的建立与求解

5.5.1 模型的建立

舞龙队沿问题 4 设定的路径行进，调头路径与盘入、盘出螺线相切，且前一段圆弧的半径为后一段的 2 倍，其中 S 形曲线由两段圆弧相切组成。龙头前进时，其它位置的速度与位置受半径、角速度影响。保持龙头的速度恒定，但需要调整最大速度以保证所有把手的速度不超过 2 m/s。已知调头的曲线涉及半径不同的两段圆弧。

5.5.2 模型的求解

计算圆弧上每个把手的角速度，确定每个把手在两段圆弧上的运动速度，尤其是半径较小的部分，其对应速度可能达到上限值 2 m/s。最后从龙头开始，逐

步调整龙头的最大速度，使得所有把手的速度都不超过 2 m/s。

5.5.3 结果

龙头的最大速度为 1.24 m/s。

六、模型总结

6.1 模型优点

1. 整体性强：模型通过螺线路径、圆弧曲线调头路径等几何方法，全面解决了舞龙队行进中的多个问题，包括行进路径、调头空间和最大速度的计算。
2. 计算准确：模型充分考虑了板凳龙的各个把手之间的几何关系和速度约束，特别是在问题 5 中，精确调整了龙头的最大行进速度，以确保所有把手的速度不超过 2 m/s。
3. 合理性高：通过对等距螺线和 S 形曲线的分段处理，保证了舞龙队行进和调头过程中路径的可行性和最优性。
4. 可扩展性好：模型中对于速度、路径和曲线的分析方法可以适用于其他类似的多段连接物体的运动规划问题。

6.2 模型缺点

1. 简化假设较多：模型假设板凳龙的连接为刚性，忽略了实际情况中可能存在的柔性连接和误差，这可能对实际应用产生一定影响。

2. 计算复杂性：在问题 5 中，模型的速度调整和最大速度计算需要反复迭代，增加了计算复杂度。

6.3 模型推广

1. 改进柔性连接：未来可以考虑将板凳龙的连接部分设定为柔性结构，加入弹性因素来使模型更贴近实际情况。
2. 实时调整：通过进一步优化算法，可以实现实时调整舞龙队的速度和路径，特别是在面对不规则表演空间时，提高模型的适应性。

七、参考文献

[1]刘崇军.等距螺旋的原理与计算[J].数学的实践与认识,2018,48(11):165-174.朱世清. 多因子选股模型的构建与应用[D].山东财经大学,2015.
[2]陈远宁.关于等距曲线的若干研究[D].合肥工业大学,2007.
[3]刘娜,毛晓菊.基于分离轴定理的碰撞检测算法[J].数字技术与应用,2012,(08):102.DOI:10.19695/j.cnki.cn12-1369.2012.08.070.
[4]李贞.虚拟场景中刚体碰撞检测的优化算法研究[D].吉林农业大学,2016.
[5]陈彦,王威.质点沿对数螺线运动的速度和加速度[J].扬州职业大学学报,2023,27(02):54-57.DOI:10.15954/j.cnki.cn32-1529/g4.2023.02.008.
[6]刘晓妍,吕濯缨,高国成.地面搜索路径的“S式”折线模型和螺线模型[J].河南教育学院学报(自然科学版),2011,20(01):7-9.

八、附录

第一问代码

```
clc; close all;
warning off;

% 参数定义
luoju = 55e-2; % 螺距
k = luoju / (2 * pi); % 螺线方程的系数 r = k * theta
L1 = 341e-2;
D1 = L1 - 27.5e-2 * 2; % 龙头把手两个孔之间的距离
L2 = 220e-2;
D2 = L2 - 27.5e-2 * 2; % 其他凳子把手两个孔之间的距离

% 绘制部分螺线
theta = 16 * 2 * pi : -0.01 : 0 * pi; % 定义角度范围
r = k * theta; % 计算半径
x = r .* cos(theta); % 计算 X 坐标
y = r .* sin(theta); % 计算 Y 坐标
figure(1)
set(gcf, 'Position', [200 200 600 600]); % 设置图形窗口大小
plot(x, y, '--') % 绘制螺线
axis equal
grid on
xlabel('x')
ylabel('y')
hold on

% 计算龙头第一个把手的位置演化
mydtheta = @(t, theta) -1 ./ (k * sqrt(1 + theta.^2)); % 定义微分方程
theta0 = 2 * pi * 16; % 初始角度
dt = 1; % 时间步长
tspan = 0:dt:300; % 时间范围
[tt, theta] = ode45(mydtheta, tspan, theta0); % 使用龙格库塔法求解微分方程
X1 = k * theta .* cos(theta); % 计算第一个把手的 X 坐标
Y1 = k * theta .* sin(theta); % 计算第一个把手的 Y 坐标

% 动态绘制龙头第一个把手的轨迹
for i = 1:1:length(theta)
    title({['总时间 t=',num2str(tt(i)),'s'], '头部第一个把手中心的轨迹'})
    plot(X1(i), Y1(i), 'r.', 'MarkerSize', 10)
    drawnow
end
```

```
hwait = waitbar(0, '计算开始...');

% 计算每个时间点下的孔的位置
N = 223; % 龙头+龙身+龙尾总的孔数
X = zeros(N+1, length(X1));
Y = zeros(N+1, length(X1));
Theta = zeros(N+1, length(X1)); % 记录每个孔在各个时刻的角度
X(1,:) = X1; % 头部第一个把手的位置
Y(1,:) = Y1;
Theta(1,:) = theta; % 第一个把手的角度

% 循环计算各个孔的位置
for j = 1:length(tt)
    for i = 2:N+1
        % 判断第一个凳子与其他凳子，孔之间的距离不同
        d = D1 * (i <= 2) + D2 * (i > 2);
        thetaij = solve_theta(luoju, X(i-1, j), Y(i-1, j), Theta(i-1, j), d); %
子函数求解下一个孔的角度
        Theta(i, j) = thetaij;
        X(i, j) = k * thetaij * cos(thetaij); % 计算每个孔的 X 坐标
        Y(i, j) = k * thetaij * sin(thetaij); % 计算每个孔的 Y 坐标
        waitbar(((j-1)*N+i)/(length(tt)*N), hwait, '已经完成...');
    end
end

close(hwait);

% 计算每个孔的速度
v = zeros(N+1, length(tt));
v(1, :) = 1; % 第一个孔的速度初始化为 1

for j = 2:length(tt)
    for i = 2:N+1
        % 计算各个孔的速度
        tan_phi_i = (sin(Theta(i, j)) + Theta(i, j) * cos(Theta(i, j))) /
(cos(Theta(i, j)) - Theta(i, j) * sin(Theta(i, j)));
        tan_phi_i1 = (sin(Theta(i-1, j)) + Theta(i-1, j) * cos(Theta(i-1, j)))
/ (cos(Theta(i-1, j)) - Theta(i-1, j) * sin(Theta(i-1, j)));
        tan_phi_ij = (Theta(i, j) * sin(Theta(i, j)) - Theta(i-1, j) *
sin(Theta(i-1, j))) / (Theta(i, j) * cos(Theta(i, j)) - Theta(i-1, j) *
cos(Theta(i-1, j)));

        % 更新速度
```

```
        v(i, j) = v(i-1, j) * (cos(atan(tan_phi_i1) - atan(tan_phi_ij))) /
(cos(atan(tan_phi_ij) - atan(tan_phi_i)));
    end
end

% 确保速度为正
v = abs(v);  % 确保所有速度值为正

% 将结果写入文件
nn = 1 / dt;
index = 1:nn:length(tt);
Dataxy = zeros(2 * (N+1), length(index));
Dataxy(1:2:end, :) = round(X(:, index), 6);
Dataxy(2:2:end, :) = round(Y(:, index), 6);

filename = 'result1.xlsx';
sheet = 1;
xlRange = 'B2';
xlswrite(filename, Dataxy, sheet, xlRange);

% 将速度数据写入文件
DataV = round(v(:, index), 6);
sheet = 2;
xlRange = 'B2';
xlswrite(filename, DataV, sheet, xlRange);

% 记录特定时间点的数据
nn2 = 60 / dt;
index2 = 1:nn2:length(tt);
index_row = [1, 2:50:224, 224];
Dataxy2 = zeros(2 * length(index_row), length(index2)); % 特定时间点位置数据
Dataxy2(1:2:end, :) = round(X(index_row, index2), 6); % x 坐标
Dataxy2(2:2:end, :) = round(Y(index_row, index2), 6); % y 坐标
Datav2 = round(v(index_row, index2), 6); % 速度信息

% 子函数：求解下一个孔的角度
function theta = solve_theta(luoju, x1, y1, theta1, d)
    k = luoju / (2 * pi); % 计算系数 k
    fun = @(theta) (k * theta * cos(theta) - x1)^2 + (k * theta * sin(theta)
- y1)^2 - d^2; % 利用两点间距离求解
    q = 0.01; % 初始步长
    options = optimoptions('fsolve', 'Display', 'off');
    theta = fsolve(fun, theta1 + q, options); % 基于非线性方程求解
    while theta <= theta1 || abs(k * theta - k * theta1) > luoju / 2
```

```
        q = q + 0.1; % 增加步长，继续求解
        theta = fsolve(fun, theta + q, options); % 确保求解的角度满足要求
    end
end
```

```python
import numpy as np
import scipy.integrate as integrate
import scipy.optimize as optimize
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import pandas as pd


# 常量
k = 0.55 / (2 * np.pi)
velocity_head = -1
L_head = 2.86
L_body = 1.65
num_segments = 223 + 1
time_steps = 301


# 函数
def normalize_vector(vector):
    norm = np.linalg.norm(vector)
    return vector / norm


def vector_projection(a, b):
    # 计算 a 在 b 上的投影
    projection = (np.dot(a, b) / np.dot(b, b)) * b
    return projection


def spiral_parametric(theta):
    return k * theta


def calc_position(theta):
    r = spiral_parametric(theta)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y
```

```python
def tangent_vector(theta):
    x = k * (np.cos(theta) - theta * np.sin(theta))
    y = k * (np.sin(theta) + theta * np.cos(theta))
    return x, y


def ds_dtheta(theta):
    return np.sqrt(k**2 + (k * theta) ** 2)


def s_theta_numeric(theta):
    result, _ = integrate.quad(ds_dtheta, 0, theta)
    return result


def theta_s_numeric(s_value):
    def func_to_solve(theta):
        return s_theta_numeric(theta) - s_value

    try:
        theta_solution = optimize.root_scalar(
            func_to_solve,
            bracket=[0, 16 * 2 * np.pi],  # 手动调整
            method="brentq",
        )
        if not theta_solution.converged:
            raise ValueError("无法找到合适的 theta 值")
        return theta_solution.root
    except Exception as e:
        print("s_value:", s_value)
        print(f"Error in theta_s_numeric: {e}")
        return None


def next_theta(theta, L, max_retries=10, tolerance=1e-5, max_iter=10000):
    x_0, y_0 = calc_position(theta)

    def func_to_solve(next_theta):
        x_1, y_1 = calc_position(next_theta)
        return np.sqrt((x_1 - x_0) ** 2 + (y_1 - y_0) ** 2) - L

    step_factor = 0.5  # 用来动态调整步长的因子
    retries = 0

    while retries < max_retries:
```

```
        # 初始猜测值
        guess_1 = theta
        guess_2 = theta + 1e-5

        try:
            # 使用 secant 方法，不需要 bracket，只需要初始猜测值，并设置最大迭
代次数
            solution = optimize.root_scalar(
                func_to_solve,
                x0=guess_1,
                x1=guess_2,
                method="secant",
                xtol=tolerance,
                maxiter=max_iter,  # 增加最大迭代次数
            )

            if solution.converged:
                return solution.root
            else:
                raise ValueError("Root finding did not converge")

        except Exception as e:
            print(f"Error at theta={theta}, L={L}, retry={retries + 1}:
{e}")
            step_factor *= 1.5  # 扩大初始猜测值的差距
            retries += 1

    print(f"Failed to find a solution for theta={theta} after {max_retries}
retries.")
    return None


def next_velocity(theta, next_theta, velocity, tolerance=1e-5,
max_iter=10000):
    def func_to_solve(next_velocity):
        # 计算当前节点和下一个节点的位移向量
        x0, y0 = calc_position(theta)
        x1, y1 = calc_position(next_theta)
        dist_vector = np.array([x1 - x0, y1 - y0])

        # 当前速度在位移向量上的投影
        v0_projection = vector_projection(
            velocity * normalize_vector(tangent_vector(theta)),
dist_vector
        )
```

```
        # 下一速度在位移向量上的投影
        v1_projection = vector_projection(
            next_velocity * normalize_vector(tangent_vector(next_theta)),
dist_vector
        )

        # 比较两个速度的差异
        return np.linalg.norm(v0_projection) -
np.linalg.norm(v1_projection)

    # 使用 `secant` 方法求解下一个速度的大小
    result = optimize.root_scalar(
        func_to_solve,
        bracket=[
            -2 * np.linalg.norm(velocity),
            0,
        ],  # 假设新的速度不会超过两倍的当前速度
        method="brentq",
        xtol=tolerance,
        maxiter=max_iter,
    )

    if result.converged:
        return result.root
    else:
        print(
            f"Failed to compute next velocity at theta={theta},
next_theta={next_theta}"
        )
        return None


# 定义变量
# 初始化结果存储
positions = []
velocities = []

# 初始化龙头的角度和位置
theta_head = 16 * 2 * np.pi
x_head, y_head = calc_position(theta_head)


# 计算
for t in range(time_steps):
```

```
    # 计算龙头的位置
    x_head, y_head = calc_position(theta_head)

    positions.append([x_head, y_head])
    velocities.append([velocity_head])

    # 计算龙身的位置
    _next_theta = theta_head
    _next_velocity = velocity_head
    for i in range(1, num_segments):
        L = L_head if i == 1 else L_body
        theta_tmp = _next_theta
        _next_theta = next_theta(_next_theta, L)
        if _next_theta is None:
            break
        x_segment, y_segment = calc_position(_next_theta)
        positions[-1].extend([x_segment, y_segment])
        _next_velocity = next_velocity(theta_tmp, _next_theta,
_next_velocity)
        if _next_velocity is None:
            break
        velocities[-1].append(_next_velocity)
        # 每隔 10 个时间步长输出一次 (显示进度)
        if (t % 10 == 0) and (i % 10) == 0:
            print(f"t={t}, i={i}")

    # 更新龙头的角度
    theta_head = theta_s_numeric(s_theta_numeric(theta_head) +
velocity_head)


# 测试
# 画某个节点不同时间的位置
# 提取第 idx 个节点的坐标
idx = 150  # 0-222, 自选
x_coords = [positions[t][2 * idx] for t in range(time_steps)]
y_coords = [positions[t][2 * idx + 1] for t in range(time_steps)]

# 获取坐标的最小值和最大值
x_min, x_max = min(x_coords), max(x_coords)
y_min, y_max = min(y_coords), max(y_coords)

# 扩展范围，使图像有一点边距
x_padding = (x_max - x_min) * 0.05  # 5% padding
y_padding = (y_max - y_min) * 0.05  # 5% padding
```

```
# 生成约束螺线的轨迹
theta_vals = np.linspace(0, (16 + 6) * 2 * np.pi, 10000)
x_constrained = [calc_position(theta)[0] for theta in theta_vals]
y_constrained = [calc_position(theta)[1] for theta in theta_vals]

# 绘制第 idx 个节点的点线图
plt.figure(figsize=(10, 6))

# 绘制节点轨迹
plt.plot(x_coords, y_coords, marker="o", linestyle="-", color="b",
label=f"Node {idx}")

# 绘制约束螺线的轨迹
plt.plot(
    x_constrained, y_constrained, linestyle="--", color="r",
label="Constrained Spiral"
)

plt.title(f"Position of Node {idx} Over Time")
plt.xlabel("X Position")
plt.ylabel("Y Position")
plt.legend()
plt.grid(True)

# 自适应区域并添加边距
plt.xlim(x_min - x_padding, x_max + x_padding)
plt.ylim(y_min - y_padding, y_max + y_padding)

plt.axis("equal")
plt.show()

# 画所有节点在某个时间点的位置
# 提取第 time 时间点所有节点的坐标
time = 157  # 0-300, 自选
if time < len(positions):
    x_coords_all_nodes = [positions[time][2 * i] for i in
range(num_segments)]
    y_coords_all_nodes = [positions[time][2 * i + 1] for i in
range(num_segments)]

    # 获取坐标的最小值和最大值
    x_min, x_max = min(x_coords_all_nodes), max(x_coords_all_nodes)
    y_min, y_max = min(y_coords_all_nodes), max(y_coords_all_nodes)
```

```
    # 扩展范围，使图像有一点边距
    x_padding = (x_max - x_min) * 0.05  # 5% padding
    y_padding = (y_max - y_min) * 0.05  # 5% padding

    # 绘制第 time 时间点所有节点的位置图
    plt.figure(figsize=(10, 6))
    plt.plot(
        x_coords_all_nodes,
        y_coords_all_nodes,
        marker="o",
        linestyle="-",
        color="b",
        label=f"All Nodes at Time {time}",
    )

    # 绘制约束螺线的轨迹
    plt.plot(
        x_constrained,
        y_constrained,
        linestyle="--",
        color="r",
        label="Constrained Spiral",
    )

    plt.title(f"Position of All Nodes at Time {time}")
    plt.xlabel("X Position")
    plt.ylabel("Y Position")
    plt.legend()
    plt.grid(True)

    # 自适应范围并添加边距
    plt.xlim(x_min - x_padding, x_max + x_padding)
    plt.ylim(y_min - y_padding, y_max + y_padding)

    plt.axis("equal")
    plt.show()
else:
    print(f"Time point {time} exceeds available data.")

# 绘制龙的轨迹的动画
# 提取每 m 个节点的坐标
node_indices = range(0, num_segments, 50)  # 0-222, 自选
```

```
# 获取所有时间点的坐标的最小值和最大值
x_min = min(positions[t][2 * i] for t in range(len(positions)) for i in
node_indices)
x_max = max(positions[t][2 * i] for t in range(len(positions)) for i in
node_indices)
y_min = min(
    positions[t][2 * i + 1] for t in range(len(positions)) for i in
node_indices
)
y_max = max(
    positions[t][2 * i + 1] for t in range(len(positions)) for i in
node_indices
)

# 扩展范围，使图像有一点边距
x_padding = (x_max - x_min) * 0.25  # 25% padding
y_padding = (y_max - y_min) * 0.25  # 25% padding

# 创建图形和轴对象
fig, ax = plt.subplots(figsize=(10, 6))
ax.set_xlim(x_min - x_padding, x_max + x_padding)
ax.set_ylim(y_min - y_padding, y_max + y_padding)
ax.set_title("Movement of Nodes Over Time")
ax.set_xlabel("X Position")
ax.set_ylabel("Y Position")
ax.grid(True)

# 绘制线条和节点
lines = []
for node_idx in node_indices:
    (line,) = ax.plot([], [], marker="o", linestyle="-", label=f"Node
{node_idx}")
    lines.append(line)


# 绘制约束螺线的轨迹
ax.plot(
    x_constrained, y_constrained, linestyle="--", color="r",
label="Constrained Spiral"
)


# 动画更新函数
def update(frame):
    for i, node_idx in enumerate(node_indices):
```

```
        x_coords = [positions[t][2 * node_idx] for t in range(frame)]
        y_coords = [positions[t][2 * node_idx + 1] for t in range(frame)]
        lines[i].set_data(x_coords, y_coords)
    return lines


# 创建动画
anim = FuncAnimation(fig, update, frames=len(positions), interval=100,
blit=True)

# 显示动画
plt.legend()
plt.show()


# 保存数据
# 创建数据字典
# position_data = {"Time_Step": []}
# velocity_data = {"Time_Step": []}
position_data = {}
velocity_data = {}

for i in range(num_segments):
    position_data[f"X_Node_{i}"] = []
    position_data[f"Y_Node_{i}"] = []
    velocity_data[f"Velocity_Node_{i}"] = []

for t in range(time_steps):
    # position_data["Time_Step"].append(t)
    # velocity_data["Time_Step"].append(t)
    for i in range(num_segments):
        position_data[f"X_Node_{i}"].append(positions[t][2 * i])
        position_data[f"Y_Node_{i}"].append(positions[t][2 * i + 1])
        velocity_data[f"Velocity_Node_{i}"].append(velocities[t][i])

# 创建 DataFrame
position_df = pd.DataFrame(position_data)
velocity_df = pd.DataFrame(velocity_data)

# 转置 DataFrame
position_df_transposed = position_df.T
velocity_df_transposed = velocity_df.T

# 定义输出文件路径
output_file_path = r"./result1.xlsx"
```

```python
# 读取已有的表头和行名
# header=0 表示第一行作为列名，index_col=0 表示第一列作为行名
position_existing_df = pd.read_excel(
    output_file_path, sheet_name="位置", header=0, index_col=0
)
velocity_existing_df = pd.read_excel(
    output_file_path, sheet_name="速度", header=0, index_col=0
)

# 追加新数据并保持表头和行名
position_updated_df = pd.concat([position_existing_df],
ignore_index=False)
velocity_updated_df = pd.concat([velocity_existing_df],
ignore_index=False)

# 将更新后的数据写入到新的 Excel 文件
new_output_file_path = r"./result1.xlsx"
with pd.ExcelWriter(new_output_file_path, engine="openpyxl") as writer:
    position_updated_df.to_excel(writer, sheet_name="位置")
    velocity_updated_df.to_excel(writer, sheet_name="速度")

with pd.ExcelWriter(
    new_output_file_path, engine="openpyxl", mode="a",
if_sheet_exists="overlay"
) as writer:
    position_df_transposed.to_excel(
        writer,
        sheet_name="位置",
        startrow=1,
        startcol=1,
        index=False,
        header=False,
    )
    velocity_df_transposed.to_excel(
        writer,
        sheet_name="速度",
        startrow=1,
        startcol=1,
        index=False,
        header=False,
    )

print(f"Data successfully saved to {new_output_file_path}")
```

第二问代码

```
clc; close all; clear;
warning off

% 参数设置
luoju = 55e-2; % 螺距，单位为米
k = luoju / (2 * pi); % 螺线方程的系数 r = k * theta
L1 = 341e-2; % 长方形的长度，单位为米
D1 = L1 - 27.5e-2 * 2; % 龙头把手两个孔之间的距离，考虑了孔的直径
L2 = 220e-2; % 其他凳子的长度，单位为米
D2 = L2 - 27.5e-2 * 2; % 其他凳子把手两个孔之间的距离，考虑了孔的直径

% 画出部分螺线
theta = 32*pi:-0.01:0*pi; % 角度范围，从 16 圈到 0 圈
r = k * theta; % 螺线的半径
x = r.* cos(theta); % x 坐标
y = r.* sin(theta); % y 坐标
figure(1)
set(gcf, 'Position', [200 200 600 600]); % 设置图形窗口位置和大小
plot(x, y, '--') % 绘制螺线
axis equal % 坐标轴比例相等
grid on % 显示网格
xlabel('x') % x 轴标签
ylabel('y') % y 轴标签
hold on % 保持当前图形

% 初始条件设置
mydtheta = @(t, theta) -1 / (k * sqrt(1 + theta.^2)); % 定义螺旋线的微分方程
theta0 = 57.032076651015522; % 初始位置的角度
dt = 0.1; % 时间步长
flag = 0; % 标记是否发生接触
step = 0; % 时间步数

N = 223; % 龙头、龙身和龙尾的总个数
X = nan * zeros(N + 1, 3); % 记录每个把手点在一个时间区间内的值
Y = nan * zeros(N + 1, 3); % 每一行代表每个凳子的前把手孔的位置在各个时间点处的值
Theta = nan * zeros(N + 1, 3); % 记录每个孔在时间区间的位置对应的角度 theta
Theta(1, 3) = theta0; % 头把手初始时刻(t=300)的角度值

% 创建用于存储位置和速度随时间变化的数据的数组
PositionData = nan(N + 1, 3, step + 1); % 位置数据，x, y, theta
```

```
SpeedData = nan(N + 1, step + 1); % 速度数据

while flag == 0 % 如果 flag=0，说明一直没有接触，可以继续往前进
    step = step + 1;
    X(:, 1) = X(:, 3);
    Y(:, 1) = Y(:, 3);
    Theta(:, 1) = Theta(:, 3); % 开始下一时间步之前，把当前时间区间末尾处的各个把手位移值当作下一个区间开始的数据

    tspan = [0, dt / 2, dt]; % 求解下一个步长下的位置点
    [tt, theta] = ode45(mydtheta, tspan, Theta(1, 1)); % 龙格库塔法求解
    X1 = k * theta .* cos(theta); % 计算下一个时刻头部把手的 x 坐标
    Y1 = k * theta .* sin(theta); % 计算下一个时刻头部把手的 y 坐标
    X(1, :) = X1;
    Y(1, :) = Y1;
    Theta(1, :) = theta;

    for j = 2:length(tt)
        for i = 2:N + 1
            d = D1 * (i <= 2) + D2 * (i > 2); % 判断是第一个凳子还是其他凳子
            thetaij = solve_theta(luoju, X(i - 1, j), Y(i - 1, j), Theta(i - 1, j), d); % 求解下一个孔的角度值
            Theta(i, j) = thetaij;
            X(i, j) = k * thetaij * cos(thetaij);
            Y(i, j) = k * thetaij * sin(thetaij);
        end
    end

    hp = plot(X(:, end), Y(:, end), 'k-', 'LineWidth', 1.2, 'Marker', 'o', 'MarkerSize', 6, 'MarkerFaceColor', 'r'); % 更新当前时刻的龙
    title({['t=', num2str(300 + step * dt), ' s'], '继续盘入，轨迹'})
    drawnow

    % 判断是否接触
    for i = 1:N
        x_1 = X(i, 2); x_2 = X(i + 1, 2); % 当前两个孔的 x 坐标
        y_1 = Y(i, 2); y_2 = Y(i + 1, 2); % 当前两个孔的 y 坐标
        theta_1 = Theta(i, 2); % 当前孔的角度
        theta_2 = Theta(i + 1, 2); % 下一个孔的角度

        % 寻找可能的交点
        index1 = find((theta_1 + 2 * pi - Theta(:, 2)) > 0); % 从外面一层找危险点
        index1 = index1(end - 2:end); % 最近的三个点的指标
```

```
        index2 = find(Theta(:, 2) - (theta_2 + 2 * pi) > 0); % 找到距离后把手最近的外面一层的三个点的指标
        if isempty(index2)
            break; % 如果到达了盘入口附近，直接跳出循环，开始下一个时刻位置更新
        else
            index2 = index2(1:min(3, length(index2))); % 找到最近的三个点的指标
        end
        index_i = index1(1):index2(end); % 这是当前全部要考虑的外面一层的把手点的位置指标，从小到大（逆时针排布）
        n = 20; m = 40; % 长方形的均匀离散点数量
        for kk = 1:length(index_i) - 1
            X2_1 = [X(index_i(kk), 2); Y(index_i(kk), 2)]; % 当前把手点的坐标
            X2_2 = [X(index_i(kk + 1), 2); Y(index_i(kk + 1), 2)]; % 下一个把手点的坐标
            panduan = find_if_intersect(L1 * (i <= 1) + L2 * (i > 1), [x_1; y_1], [x_2; y_2], L2, X2_1, X2_2, n, m); % 判断是否相交
            if ~isempty(panduan)
                flag = 1; % 如果有相交，设置 flag 为 1
                break;
            end
        end
        if flag == 1
            break;
        end
    end

    % 存储位置和速度数据
    PositionData(:, :, step) = [X(:, end), Y(:, end), Theta(:, end)];
    v = zeros(N + 1, length(tt));
    v(1, :) = 1; % 第一个孔的速度

    for j = 2:length(tt)
        for i = 2:N + 1
            % 计算各个孔的速度
            tan_phi_i = (sin(Theta(i, j)) + Theta(i, j) * cos(Theta(i, j))) / (cos(Theta(i, j)) - Theta(i, j) * sin(Theta(i, j)));
            tan_phi_i1 = (sin(Theta(i - 1, j)) + Theta(i - 1, j) * cos(Theta(i - 1, j))) / (cos(Theta(i - 1, j)) - Theta(i - 1, j) * sin(Theta(i - 1, j)));
            tan_phi_ij = (Theta(i, j) * sin(Theta(i, j)) - Theta(i - 1, j) * sin(Theta(i - 1, j))) / (Theta(i, j) * cos(Theta(i, j)) - Theta(i - 1, j) * cos(Theta(i - 1, j)));

            % 更新速度
```

```
            v(i, j) = v(i - 1, j) * (cos(atan(tan_phi_i1) - atan(tan_phi_ij)))
/ (cos(atan(tan_phi_ij) - atan(tan_phi_i)));
        end
    end
    SpeedData(:, step) = v(:, end);

    delete(hp) % 删除旧的图形，准备绘制下一个时间步
end

% 最后一个时间步下的龙的位置更新
hp = plot(X(:, end), Y(:, end), 'k-', 'LineWidth', 1.2, 'Marker', 'o',
'MarkerSize', 6, 'MarkerFaceColor', 'r');
title({['t=', num2str(300 + step * dt), ' s'], '继续盘入，轨迹'})
drawnow

% 输出位置和速度数据以供进一步分析
disp('位置数据：');
disp(PositionData);
disp('速度数据：');
disp(SpeedData);

% 求解螺旋线上的点的角度
function theta = solve_theta(luoju, x1, y1, theta1, d)
    k = luoju / 2 / pi; % 螺线方程的系数
    fun = @(theta)(k*theta.*cos(theta) - x1).^2 + (k*theta.*sin(theta) - y1).^2
- d^2; % 距离公式
    q = 0.01; % 初始猜测
    options = optimoptions('fsolve', 'Display', 'off'); % 不显示求解过程
    theta = fsolve(fun, theta1 + q, options); % 求解角度
    while theta <= theta1 || abs(k*theta - k*theta1) > luoju / 2 % 确保角度大
于 theta1
        q = q + 0.1;
        theta = fsolve(fun, theta + q, options); % 重新求解
    end
end

% 判断两个长方形是否相交
function flag = find_if_intersect(L1, X1_1, X1_2, L2, X2_1, X2_2, n, m)
    k1 = (X1_1(2) - X1_2(2)) / (X1_1(1) - X1_2(1)); % 第一个长方形的斜率
    k1_ = -1 / k1; % 垂线的斜率
    k2 = (X2_1(2) - X2_2(2)) / (X2_1(1) - X2_2(1)); % 第二个长方形的斜率
    k2_ = -1 / k2; % 垂线的斜率
    X1_center = (X1_1 + X1_2) / 2; % 第一个长方形的中心点
    X2_center = (X2_1 + X2_2) / 2; % 第二个长方形的中心点
```

```
    A = [k1_ -1; k2_ -1];
    P = A \ [k1_ * X1_center(1) - X1_center(2); k2_ * X2_center(1) -
X2_center(2)]; % 垂线交点
    vec1 = X1_center - P;
    vec2 = X2_center - P;
    theta1 = angle(vec1(1) + 1i * vec1(2));
    theta2 = angle(vec2(1) + 1i * vec2(2));
    delta_theta = theta1 - theta2;
    d1 = norm(vec1);
    d2 = norm(vec2);
    [X, Y] = meshgrid(linspace(d1 - 30e-2 / 2, d1 + 30e-2 / 2, n), linspace(0
- L1 / 2, 0 + L1 / 2, m)); % 均匀离散点
    T = [cos(delta_theta) -sin(delta_theta); sin(delta_theta)
cos(delta_theta)];
    XY_new = T * [X(:) Y(:)]'; % 坐标旋转
    flag = find(abs(XY_new(1, :) - d2) < 30e-2 / 2 & abs(XY_new(2, :) - 0) <
L2 / 2); % 判断是否相交
end
```

```
import numpy as np
import scipy.integrate as integrate
import scipy.optimize as optimize
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
import pandas as pd


# 常量
k = 0.55 / (2 * np.pi)
velocity_head = -1
dist_head = 3.41 - 2 * 27.5 / 100
dist_body = 2.2 - 2 * 27.5 / 100
l_head = 341 / 100
l_body = 22 / 100
width = 30 / 100
num_segments = 223 + 1
t = 0
dt = 1


# 函数
def normalize_vector(vector):
    norm = np.linalg.norm(vector)
    return vector / norm
```

```python
def vector_projection(a, b):
    # 计算 a 在 b 上的投影
    projection = (np.dot(a, b) / np.dot(b, b)) * b
    return projection


def perpendicular_unit_vector(x1, y1, x2, y2):
    # 向量 AB 的分量
    dx = x2 - x1
    dy = y2 - y1

    # 垂直于 AB 的向量 (-dy, dx)
    normal_vector = np.array([-dy, dx])

    return normalize_vector(normal_vector)


def spiral_parametric(theta):
    return k * theta


def calc_position(theta):
    r = spiral_parametric(theta)
    x = r * np.cos(theta)
    y = r * np.sin(theta)
    return x, y


def tangent_vector(theta):
    x = k * (np.cos(theta) - theta * np.sin(theta))
    y = k * (np.sin(theta) + theta * np.cos(theta))
    return x, y


def ds_dtheta(theta):
    return np.sqrt(k**2 + (k * theta) ** 2)


def s_theta_numeric(theta):
    result, _ = integrate.quad(ds_dtheta, 0, theta)
    return result


def theta_s_numeric(s_value):
```

```
    def func_to_solve(theta):
        return s_theta_numeric(theta) - s_value

    try:
        theta_solution = optimize.root_scalar(
            func_to_solve,
            bracket=[0, 16 * 2 * np.pi],  # 手动调整
            method="brentq",
        )
        if not theta_solution.converged:
            raise ValueError("无法找到合适的 theta 值")
        return theta_solution.root
    except Exception as e:
        print("s_value:", s_value)
        print(f"Error in theta_s_numeric: {e}")
        return None


def next_theta(theta, L, max_retries=10, tolerance=1e-5, max_iter=10000):
    x_0, y_0 = calc_position(theta)

    def func_to_solve(next_theta):
        x_1, y_1 = calc_position(next_theta)
        return np.sqrt((x_1 - x_0) ** 2 + (y_1 - y_0) ** 2) - L

    step_factor = 0.5  # 用来动态调整步长的因子
    retries = 0

    while retries < max_retries:
        # 初始猜测值
        guess_1 = theta
        guess_2 = theta + 1e-5

        try:
            # 使用 secant 方法，不需要 bracket，只需要初始猜测值，并设置最大迭
代次数
            solution = optimize.root_scalar(
                func_to_solve,
                x0=guess_1,
                x1=guess_2,
                method="secant",
                xtol=tolerance,
                maxiter=max_iter,  # 增加最大迭代次数
            )
```

```
            if solution.converged:
                return solution.root
            else:
                raise ValueError("Root finding did not converge")

        except Exception as e:
            print(f"Error at theta={theta}, L={L}, retry={retries + 1}:
{e}")
            step_factor *= 1.5  # 扩大初始猜测值的差距
            retries += 1

    print(f"Failed to find a solution for theta={theta} after {max_retries}
retries.")
    return None


def next_velocity(theta, next_theta, velocity, tolerance=1e-5,
max_iter=10000):
    def func_to_solve(next_velocity):
        # 计算当前节点和下一个节点的位移向量
        x0, y0 = calc_position(theta)
        x1, y1 = calc_position(next_theta)
        dist_vector = np.array([x1 - x0, y1 - y0])

        # 当前速度在位移向量上的投影
        v0_projection = vector_projection(
            velocity * normalize_vector(tangent_vector(theta)),
dist_vector
        )

        # 下一速度在位移向量上的投影
        v1_projection = vector_projection(
            next_velocity * normalize_vector(tangent_vector(next_theta)),
dist_vector
        )

        # 比较两个速度的差异
        return np.linalg.norm(v0_projection) -
np.linalg.norm(v1_projection)

    # 使用 `secant` 方法求解下一个速度的大小
    result = optimize.root_scalar(
        func_to_solve,
        bracket=[
            -2 * np.linalg.norm(velocity),
```

```
            0,
        ],  # 假设新的速度不会超过两倍的当前速度
        method="brentq",
        xtol=tolerance,
        maxiter=max_iter,
    )

    if result.converged:
        return result.root
    else:
        print(
            f"Failed to compute next velocity at theta={theta},
next_theta={next_theta}"
        )
        return None


def is_rectangles_intersect(
    x1_min, y1_min, x1_max, y1_max, x2_min, y2_min, x2_max, y2_max
):
    # 判断两个矩形是否相交
    if x1_max < x2_min or x1_min > x2_max or y1_max < y2_min or y1_min > y2_max:
        return False  # 矩形不相交
    else:
        return True  # 矩形相交


# 定义变量
# 初始化结果存储
positions = []
velocities = []

# 初始化龙头的角度和位置
theta_head = 16 * 2 * np.pi
x_head, y_head = calc_position(theta_head)


# 计算
for t in range(0, int(s_theta_numeric(16 * 2 * np.pi) / (-velocity_head)),
dt):
    # 计算龙头的位置
    x_head, y_head = calc_position(theta_head)

    positions.append([x_head, y_head])
    velocities.append([velocity_head])
```

```
    is_collision = False

    # 计算龙身的位置
    _next_theta = theta_head
    _next_velocity = velocity_head
    for i in range(1, num_segments):
        L = dist_head if i == 1 else dist_body
        theta_tmp = _next_theta
        _next_theta = next_theta(_next_theta, L)
        if _next_theta is None:
            break
        x_segment, y_segment = calc_position(_next_theta)
        positions[-1].extend([x_segment, y_segment])
        _next_velocity = next_velocity(theta_tmp, _next_theta,
_next_velocity)
        if _next_velocity is None:
            break
        velocities[-1].append(_next_velocity)
        # # 每隔 10 个时间步长输出一次 (显示进度)
        # if (t % 10 == 0) and (i % 10) == 0:
        #     print(f"t={t}, i={i}")

    # 碰撞检测
    for i in range(1, num_segments - 2):
        for j in range(i + 2, num_segments - 1):
            x_1_1, y_1_1 = positions[-1][2 * i - 2], positions[-1][2 * i -
1]
            x_1_2, y_1_2 = positions[-1][2 * i], positions[-1][2 * i + 1]
            x_2_1, y_2_1 = positions[-1][2 * j - 2], positions[-1][2 * j -
1]
            x_2_2, y_2_2 = positions[-1][2 * j], positions[-1][2 * j + 1]
            vec_1 = normalize_vector(np.array([x_1_1 - x_1_2, y_1_1 -
y_1_2]))
            vec_2 = normalize_vector(np.array([x_2_1 - x_2_2, y_2_1 -
y_2_2]))
            puv_1 = perpendicular_unit_vector(x_1_1, y_1_1, x_1_2, y_1_2)
            puv_2 = perpendicular_unit_vector(x_2_1, y_2_1, x_2_2, y_2_2)
            l = l_head if i == 1 else l_body
            x_1_min = (
                (x_1_1 + x_1_2) / 2 - l / 2 * abs(vec_1[0]) - width / 2 *
abs(puv_1[0])
            )
            x_1_max = (
```

```
                (x_1_1 + x_1_2) / 2 + l / 2 * abs(vec_1[0]) + width / 2 *
abs(puv_1[0])
            )
            y_1_min = (
                (y_1_1 + y_1_2) / 2 - l / 2 * abs(vec_1[1]) - width / 2 *
abs(puv_1[1])
            )
            y_1_max = (
                (y_1_1 + y_1_2) / 2 + l / 2 * abs(vec_1[1]) + width / 2 *
abs(puv_1[1])
            )
            x_2_min = (
                (x_2_1 + x_2_2) / 2
                - l_body / 2 * abs(vec_2[0])
                - width / 2 * abs(puv_2[0])
            )
            x_2_max = (
                (x_2_1 + x_2_2) / 2
                + l_body / 2 * abs(vec_2[0])
                + width / 2 * abs(puv_2[0])
            )
            y_2_min = (
                (y_2_1 + y_2_2) / 2
                - l_body / 2 * abs(vec_2[1])
                - width / 2 * abs(puv_2[1])
            )
            y_2_max = (
                (y_2_1 + y_2_2) / 2
                + l_body / 2 * abs(vec_2[1])
                + width / 2 * abs(puv_2[1])
            )
            if is_rectangles_intersect(
                x_1_min, y_1_min, x_1_max, y_1_max, x_2_min, y_2_min,
x_2_max, y_2_max
            ):
                is_collision = True
                print(
                    f"Collision detected at t={t}, i={i}, j={j},
x1_min={x_1_min}, y1_min={y_1_min}, x1_max={x_1_max}, y1_max={y_1_max},
x2_min={x_2_min}, y2_min={y_2_min}, x2_max={x_2_max}, y2_max={y_2_max}"
                )
                break
        if is_collision:
            break
```

```
    if is_collision:
        positions.pop()
        velocities.pop()
        break
    else:
        # 更新龙头的角度
        theta_head = theta_s_numeric(s_theta_numeric(theta_head) +
velocity_head)


print(t)

# # 测试
# # 画某个节点不同时间的位置
# # 提取第 idx 个节点的坐标
# idx = 157  # 0-222, 自选
# x_coords = [positions[t][2 * idx] for t in range(time_steps)]
# y_coords = [positions[t][2 * idx + 1] for t in range(time_steps)]

# # 绘制第 idx 个节点的点线图
# plt.figure(figsize=(10, 6))
# plt.plot(x_coords, y_coords, marker="o", linestyle="-", color="b",
label=f"Node {idx}")
# plt.title(f"Position of Node {idx} Over Time")
# plt.xlabel("X Position")
# plt.ylabel("Y Position")
# plt.legend()
# plt.grid(True)
# plt.axis("equal")
# plt.show()

# # 画所有节点在某个时间点的位置
# # 提取第 time 时间点所有节点的坐标
# time = 157  # 0-300, 自选
# if time < len(positions):
#     x_coords_all_nodes = [positions[time][2 * i] for i in
range(num_segments)]
#     y_coords_all_nodes = [positions[time][2 * i + 1] for i in
range(num_segments)]

#     # 绘制第 257 时间点所有节点的位置图
#     plt.figure(figsize=(10, 6))
#     plt.plot(
#         x_coords_all_nodes,
```

```
#         y_coords_all_nodes,
#         marker="o",
#         linestyle="-",
#         color="r",
#         label=f"All Nodes at Time {time}",
#     )
#     plt.title(f"Position of All Nodes at Time {time}")
#     plt.xlabel("X Position")
#     plt.ylabel("Y Position")
#     plt.legend()
#     plt.grid(True)
#     plt.axis("equal")
#     plt.show()
# else:
#     print(f"Time point {time} exceeds available data.")

# # 绘制龙的轨迹的动画
# # 提取每 m 个节点的坐标
# node_indices = range(0, num_segments, 50)  # 0-222, 自选

# # 创建图形和轴对象
# fig, ax = plt.subplots(figsize=(10, 6))
# ax.set_xlim(-50, 50)  # 根据需要调整
# ax.set_ylim(-50, 50)  # 根据需要调整
# ax.set_title("Movement of Nodes Over Time")
# ax.set_xlabel("X Position")
# ax.set_ylabel("Y Position")
# ax.grid(True)

# # 绘制线条和节点
# lines = []
# for node_idx in node_indices:
#     (line,) = ax.plot([], [], marker="o", linestyle="-", label=f"Node
{node_idx}")
#     lines.append(line)


# # 动画更新函数
# def update(frame):
#     for i, node_idx in enumerate(node_indices):
#         x_coords = [positions[t][2 * node_idx] for t in range(frame)]
#         y_coords = [positions[t][2 * node_idx + 1] for t in range(frame)]
#         lines[i].set_data(x_coords, y_coords)
#     return lines
```

```
# # 创建动画
# anim = FuncAnimation(fig, update, frames=len(positions), interval=100,
blit=True)

# # 显示动画
# plt.legend()
# plt.show()


# # 保存数据
# # 创建数据字典
# # position_data = {"Time_Step": []}
# # velocity_data = {"Time_Step": []}
# position_data = {}
# velocity_data = {}

# for i in range(num_segments):
#     position_data[f"X_Node_{i}"] = []
#     position_data[f"Y_Node_{i}"] = []
#     velocity_data[f"Velocity_Node_{i}"] = []

# for t in range(time_steps):
#     # position_data["Time_Step"].append(t)
#     # velocity_data["Time_Step"].append(t)
#     for i in range(num_segments):
#         position_data[f"X_Node_{i}"].append(positions[t][2 * i])
#         position_data[f"Y_Node_{i}"].append(positions[t][2 * i + 1])
#         velocity_data[f"Velocity_Node_{i}"].append(velocities[t][i])

# # 创建 DataFrame
# position_df = pd.DataFrame(position_data)
# velocity_df = pd.DataFrame(velocity_data)

# # 转置 DataFrame
# position_df_transposed = position_df.T
# velocity_df_transposed = velocity_df.T

# # 定义输出文件路径
# output_file_path = r"./result1.xlsx"

# # 读取已有的表头和行名
# # header=0 表示第一行作为列名, index_col=0 表示第一列作为行名
# position_existing_df = pd.read_excel(
#     output_file_path, sheet_name="位置", header=0, index_col=0
# )
```

```
# velocity_existing_df = pd.read_excel(
#     output_file_path, sheet_name="速度", header=0, index_col=0
# )

# # 追加新数据并保持表头和行名
# position_updated_df = pd.concat(
#     [position_existing_df, position_df_transposed], ignore_index=False
# )
# velocity_updated_df = pd.concat(
#     [velocity_existing_df, velocity_df_transposed], ignore_index=False
# )

# # 将更新后的数据写入到新的 Excel 文件
# new_output_file_path = r"./result1.xlsx"
# with pd.ExcelWriter(new_output_file_path, engine="openpyxl") as writer:
#     position_updated_df.to_excel(writer, sheet_name="位置")
#     velocity_updated_df.to_excel(writer, sheet_name="速度")

# with pd.ExcelWriter(
#     new_output_file_path, engine="openpyxl", mode="a",
if_sheet_exists="overlay"
# ) as writer:
#     position_df_transposed.to_excel(
#         writer,
#         sheet_name="位置",
#         startrow=1,
#         startcol=1,
#         index=False,
#         header=False,
#     )
#     velocity_df_transposed.to_excel(
#         writer,
#         sheet_name="速度",
#         startrow=1,
#         startcol=1,
#         index=False,
#         header=False,
#     )

# print(f"Data successfully saved to {new_output_file_path}")
```

第三问代码

```
clc; close all; clear;
warning off

% 参数设置
luoju_values = linspace(50e-2, 20e-2, 1000); % 螺距从大到小，单位为米
L1 = 341e-2;
D1 = L1 - 27.5e-2 * 2;
L2 = 220e-2;
D2 = L2 - 27.5e-2 * 2;

% 初始条件
theta0 = 32*pi;
dt = 0.1; % 时间步长
N = 223; % 龙头+龙身+龙尾的总个数

% 循环遍历不同的螺距值
luoju_at_contact = 0; % 记录接触时的螺距值
flag_contact = false; % 标志位，标记是否发生接触

for luoju_idx = 1:length(luoju_values)
    luoju = luoju_values(luoju_idx); % 当前螺距值
    k = luoju / (2 * pi); % 螺线方程的系数 k

    % 生成螺线图
    theta = 16*2*pi:-0.01:0*pi; % 从 16 圈减到 0 圈的角度
    r = k * theta; % 螺线半径
    x = r .* cos(theta); % x 坐标
    y = r .* sin(theta); % y 坐标
    figure(1)
    set(gcf, 'Position', [200 200 600 600]); % 图形窗口大小和位置
    plot(x, y, '--') % 绘制螺线
    axis equal
    grid on
    xlabel('x')
    ylabel('y')
    hold on

    % 初始条件
    mydtheta = @(t, theta) -1 / (k * sqrt(1 + theta.^2)); % ODE 方程
    flag = 0;
    step = 0;
```

```
    % 记录每个孔在时间区间内的位置
    X = nan * zeros(N + 1, 3); % X 坐标矩阵
    Y = nan * zeros(N + 1, 3); % Y 坐标矩阵
    Theta = nan * zeros(N + 1, 3); % 角度矩阵
    Theta(1, 3) = theta0; % 初始角度

    % 设置板的离散点数量
    n = 20; m = 40;

    % 开始时间步长循环
    while flag == 0
        step = step + 1; % 记录步数
        X(:, 1) = X(:, 3);
        Y(:, 1) = Y(:, 3);
        Theta(:, 1) = Theta(:, 3);

        % 使用 ode45 求解 ODE
        tspan = [0, dt / 2, dt];
        [tt, theta] = ode45(mydtheta, tspan, Theta(1, 1)); % 数值积分计算角度
        X1 = k * theta .* cos(theta); % 计算 X 坐标
        Y1 = k * theta .* sin(theta); % 计算 Y 坐标

        X(1, :) = X1;
        Y(1, :) = Y1;
        Theta(1, :) = theta;

        % 更新每个孔的坐标
        for j = 2:length(tt)
            for i = 2:N + 1
                d = D1 * (i <= 2) + D2 * (i > 2); % 选择距离 D1 或 D2
                thetaij = solve_theta(luoju, X(i - 1, j), Y(i - 1, j), Theta(i
- 1, j), d); % 计算新的角度
                Theta(i, j) = thetaij;
                X(i, j) = k * thetaij * cos(thetaij);
                Y(i, j) = k * thetaij * sin(thetaij);
            end
        end

        % 检查是否达到一定螺距位置
        if k * Theta(1, end) >= 4.5
            break; % 达到限制值，退出循环
        end

        % 检查是否发生接触
```

```
        for i = 1:N
            x_1 = X(i, end); x_2 = X(i + 1, end);
            y_1 = Y(i, end); y_2 = Y(i + 1, end);
            if check_contact(x_1, y_1, x_2, y_2, X, Y, Theta, i, L1, L2, n, m)
                flag = 1; % 发生接触
                luoju_at_contact = luoju; % 记录螺距值
                flag_contact = true;
                break;
            end
        end
    end

    % 如果接触已发生，退出外层循环
    if flag_contact
        break;
    end
end

% 显示结果
if flag_contact
    disp(['接触时的螺距值为: ', num2str(luoju_at_contact)]);
else
    disp('在给定的螺距范围内没有检测到接触。');
end

% 计算螺旋线上的新角度 theta
function theta = solve_theta(luoju, x1, y1, theta1, d)
    k = luoju / (2 * pi);
    fun = @(theta) (k * theta * cos(theta) - x1)^2 + (k * theta * sin(theta)
- y1)^2 - d^2;
    q = 0.01; % 初始搜索步长
    options = optimoptions('fsolve', 'Display', 'off');
    theta = fsolve(fun, theta1 + q, options);
    while theta <= theta1 || abs(k * theta - k * theta1) > luoju / 2
        q = q + 0.01;
        theta = fsolve(fun, theta + q, options);
    end
end

% 检查两个孔之间是否发生接触
function flag = check_contact(x_1, y_1, x_2, y_2, X, Y, Theta, i, L1, L2, n,
m)
    theta_1 = Theta(i, end);
    theta_2 = Theta(i + 1, end);
```

```
    index1 = find((theta_1 + 2 * pi - Theta(:, end)) > 0);
    index1 = index1(end - 2:end);
    index2 = find(Theta(:, end) - (theta_2 + 2 * pi) > 0);
    if isempty(index2)
        flag = false;
        return;
    else
        index2 = index2(1:min(3, length(index2)));
    end
    index_i = index1(1):index2(end);
    for kk = 1:length(index_i) - 1
        X2_1 = [X(index_i(kk), end); Y(index_i(kk), end)];
        X2_2 = [X(index_i(kk + 1), end); Y(index_i(kk + 1), end)];
        panduan = find_if_intersect(L1, [x_1; y_1], [x_2; y_2], L2, X2_1, X2_2,
n, m);
        if panduan
            flag = true;
            return;
        end
    end
    flag = false;
end

% 检查是否相交
function flag = find_if_intersect(L1, X1_1, X1_2, L2, X2_1, X2_2, n, m)
    k1 = (X1_1(2) - X1_2(2)) / (X1_1(1) - X1_2(1));
    k1_ = -1 / k1;
    k2 = (X2_1(2) - X2_2(2)) / (X2_1(1) - X2_2(1));
    k2_ = -1 / k2;
    X1_center = (X1_1 + X1_2) / 2;
    X2_center = (X2_1 + X2_2) / 2;
    A = [k1_ -1; k2_ -1];
    P = A \ [k1_ * X1_center(1) - X1_center(2); k2_ * X2_center(1) -
X2_center(2)];
    vec1 = X1_center - P;
    vec2 = X2_center - P;
    theta1 = angle(vec1(1) + 1i * vec1(2));
    theta2 = angle(vec2(1) + 1i * vec2(2));
    if theta1 < theta2
        flag = true;
    else
        flag = false;
    end
end
```

第四和第五问代码

```
clc; clear;
close all;

% 定义活动圆范围
theta_set = linspace(0, 2*pi, 100); % 圆周角度，从 0 到 2*pi，分成 100 个点
cicle_r = 4.50;                     % 圆的半径，单位为 cm
xlocal = cicle_r * cos(theta_set);          % 圆的 X 坐标
ylocal = cicle_r * sin(theta_set);          % 圆的 Y 坐标

% 输出表格初始化
data_location = zeros(448,201); % 存储每个时间步的成员位置（x 和 y 坐标），行数是成员总数，列数是时间步数
data_speed = zeros(224,201);    % 存储每个时间步的成员速度，行数是成员总数，列数是时间步数
dtimestep = 0.005; % 时间步长，单位为秒

circle_set = 6;   % 螺旋圈数
% 螺距（每圈螺旋的上升高度），单位为 cm
pitch_set = 1.70;    % 螺距

% 螺旋的初始半径和角度参数
rinitial = circle_set * pitch_set;          % 螺旋的初始半径，单位为 cm
detak = pitch_set / (2 * pi);           % 每弧度的半径增量

theta_set = linspace(0, -circle_set * 2 * pi, 1000); % 角度，从 0 到-6 圈，共 1000 个点
cicle_r = rinitial + detak * theta_set;             % 螺旋曲线的半径
x2 = cicle_r .* cos(theta_set);           % 螺旋曲线的 X 坐标
y2 = cicle_r .* sin(theta_set);           % 螺旋曲线的 Y 坐标
% 对称螺旋的极坐标方程
r_sym = -detak * theta_set;             % 对称螺旋的半径
x_sym = -r_sym .* cos(theta_set);   % 对称螺旋的 X 坐标
y_sym = r_sym .* sin(theta_set);    % 对称螺旋的 Y 坐标

% 圆弧的参数
dy_dx = 68 * 2 / 2/424; % 圆弧的切线斜率
angel_set = atan(dy_dx);       % 切线角度
radius1 =8.58 / 3;          % 第一段圆弧的半径
radius2 =8.58 / 6;          % 第二段圆弧的半径


% 合并 2 条圆弧绘图
```

```
arc1_angle_range = linspace(-pi + angel_set, -2 * pi + angel_set, 1000); % Angle
range for the first arc
arc1_x = radius1 * cos(arc1_angle_range) - radius2 * cos(angel_set); % X
coordinates of the first arc
arc1_y = radius1 * sin(arc1_angle_range) - radius2 * sin(angel_set); % Y
coordinates of the first arc

arc2_angle_range = linspace(-3 * pi + angel_set, -2 * pi + angel_set, 500); %
Angle range for the second arc
arc2_x = radius2 * cos(arc2_angle_range) + radius1 * cos(angel_set); % X
coordinates of the second arc
arc2_y = radius2 * sin(arc2_angle_range) + radius1 * sin(angel_set); % Y
coordinates of the second arc

turning_path_x = [arc1_x, arc2_x];
turning_path_y = [arc1_y, arc2_y];


% 各板凳把手间的长度
L1 = 3.41 -0.55; % 第一个板凳的长度
L = 2.20 - 0.55;  % 其他板凳的长度

% 初始速度和角速度
vinitial = 1.00;         % 初始线速度，单位为 cm/s
w0 = vinitial / rinitial;     % 初始角速度

%% 状态信息初始化
N = 224;          % 成员总数，223 为龙尾前，224 为龙尾后

% 初始化每个成员的位置、速度和角度
x = rinitial * ones(1, N); % 每个成员的 X 坐标
y = zeros(1, N);    % 每个成员的 Y 坐标
S = zeros(1, N);    % 每个成员走过的距离
alpha = 2 * asin(0.5 * L / rinitial) * ones(1, N); % 板凳间夹角
v_L = zeros(1, N); % 线速度
v_M = zeros(1, N); % 木板速度
w = zeros(1, N);   % 角速度

% 成员位置的极坐标（角度和半径）
theta0 = 0;
thetaN = -circle_set * 2 * pi;
theta_set = theta0 * ones(1, N); % 每个成员的角度
rho = rinitial * ones(1, N);       % 每个成员的半径
flag = zeros(1, N);          % 标志变量，表示进入圆弧的状态
```

```
flag_turn = zeros(1, N);     % 标志变量，表示是否开始调头
t2 = zeros(1, N);            % 开始调头的时间

% 标志变量：龙头是否成功抵达边界，达到边界的时间，开始调头的时间
t_arrive = 0;
t_turn = 0;
flag_stop = 0;


% 初始化参数
Nnum = 1;          % 当前进入的板凳编号（从 1 开始）
alpha(1) = 2 * asin(0.5 * L1 / rinitial); % 第一个成员的夹角，计算公式基于板凳的长度 L1 和初始半径 rinitial
v_L(1) = vinitial; % 第一个成员的线速度
w(1) = v_L(1) / rinitial; % 第一个成员的角速度，由线速度和半径计算得到

% 时间记录范围设置
tmin = 58;      % 记录的最小时间（秒）
tmax = 260;     % 最大时间（秒）

% 时间步长用于计算
dT = 14 / vinitial * 1.00; % 根据线速度 vinitial 计算每段圆弧的时间步长（以帧数表示）

% 主循环，遍历每个时间步
for t = 0:dtimestep:tmax
    % 判断是否需要进入下一个板凳
    if (-theta_set(Nnum)) >= alpha(Nnum)
        Nnum = Nnum + 1;
        if Nnum > N
            Nnum = N; % 确保 Nnum 不超过成员总数 N
        end
    end

    %% 对每个成员进行状态更新
    for i = 1:Nnum
        % 位置更新逻辑
        if rho(i) > 4.288 % 如果在螺旋区域内
            flag(i) = 0; % 标记为在螺旋区域

            if t2(i) >= dT % 检查是否离开圆弧区域
                % 更新角度和位置
                theta_set(i) = theta_set(i) + w(i) * dtimestep;
                rho(i) = rinitial + detak * theta_set(i);
```

```
            x(i) = -rho(i) * cos(theta_set(i)); % X坐标计算
            y(i) = -rho(i) * sin(theta_set(i)); % Y坐标计算
        else % 进入圆弧之前的处理
            % 更新角度和位置
            theta_set(i) = theta_set(i) - w(i) * dtimestep;
            rho(i) = rinitial + detak * theta_set(i);
            x(i) = rho(i) * cos(theta_set(i)); % X坐标计算
            y(i) = rho(i) * sin(theta_set(i)); % Y坐标计算
        end
    else % 在圆弧1区域
        t2(i) = t2(i) + dtimestep; % 更新时间
        if t2(i) >= dT
            t2(i) = dT; % 时间限制
            rho(i) = 428.9; % 设置圆弧半径
        end
        % 使用圆弧路径数据
        x(i) = turning_path_x(round(t2(i) / dT * 1500)); % X坐标
        y(i) = turning_path_y(round(t2(i) / dT * 1500)); % Y坐标
    end

    % 速度更新
    if i == 1
        % 第一个成员的速度更新
        alpha(1) = 2 * asin(0.5 * L1 / rho(1)); % 计算夹角
        v_L(1) = vinitial; % 设置线速度
        w(1) = v_L(1) / rho(1); % 计算角速度
        v_M(1) = v_L(1) * cos(0.5 * alpha(1)); % 计算木板速度
    end
    if i >= 2
        % 其他成员的速度更新
        alpha(i) = 2 * asin(0.5 * L / rho(i)); % 计算夹角
        v_L(i) = vinitial; % 设置线速度
        w(i) = v_L(i) / rho(i); % 计算角速度
        v_M(i) = v_L(i) * cos(0.5 * alpha(i)); % 计算木板速度
    end

    %% 数据保存
    if t > tmin && round(t - tmin, 2) == floor(t - tmin) % 每秒保存一次数
据
        data_location(i * 2 - 1, floor(t - tmin) + 1) = x(i); % 保存X坐标
        data_location(i * 2, floor(t - tmin) + 1) = y(i);     % 保存Y坐标
        data_speed(i, floor(t - tmin) + 1) = v_M(i);          % 保存木板速
度
    end
```

```
        if flag_stop
            break; % 如果停止标志被设置，退出循环
        end
    end % 成员循环

    if flag_stop
        break; % 如果停止标志被设置，退出主循环
    end
end

% 数据处理和显示
%结果输出

result_locate = round(data_location / 100, 6); % 将位置数据四舍五入并保存
result_speed = round(data_speed / 100, 6);    % 将速度数据四舍五入并保存
max_speed = max(result_speed(:));             % 查找并显示最大速度
disp(max_speed); % 输出最大速度
disp(result_speed)


figure(1);
fill(xlocal, ylocal, [0.9, 0.9, 0.5], 'EdgeColor', 'none'); % 使用淡黄色填充圆，并去掉边界线
hold on;

% 绘制螺旋路径和对称螺旋路径
plot(x2, y2, 'Color', [0, 0.6, 0], 'LineWidth', 2); % 使用深绿色加粗绘制螺旋路径
hold on;
plot(x_sym, y_sym, 'Color', [0.8, 0.5, 0.5], 'LineWidth', 2, 'LineStyle', '--'); % 使用虚线和红色绘制对称螺旋路径
hold on;

% 绘制调头路径的 S 形曲线
plot(arc1_x, arc1_y, 'Color', [0.4, 0.8, 0.8], 'LineWidth', 8); % 使用柔和的紫色加宽线条绘制第一段圆弧
hold on;
plot(arc2_x, arc2_y, 'Color', [0.8, 0, 0.4], 'LineWidth', 8); % 使用洋红色绘制第二段圆弧
hold on;

% 绘制成员运动轨迹
```

```
plot(x(2:N), y(2:N), '-o', 'MarkerEdgeColor', 'b', 'MarkerFaceColor', 'c',
'LineWidth', 1.5, 'MarkerSize', 6);
% 使用蓝色边框、青色填充的圆圈标记成员运动轨迹
hold on;
plot(x(1:2), y(1:2), '-k', 'LineWidth', 3); % 使用黑色粗线绘制前两名成员的轨迹
hold on;

% 设置图形属性
axis equal; % 保持 X 和 Y 轴比例一致
title(sprintf('舞龙队掉头盘出示意图', Nnum - 1, t), 'FontSize', 16); % 设置标题
字体大小
xlabel('X 方向 (cm)', 'FontSize', 14); % 设置 X 轴标签
ylabel('Y 方向 (cm)', 'FontSize', 14); % 设置 Y 轴标签
grid on; % 显示网格
set(gcf, 'Position', [700, 100, 800, 600]); % 调整图形窗口的大小
set(gca, 'FontSize', 12, 'LineWidth', 1.2, 'GridAlpha', 0.4, 'GridColor', [0.6,
0.6, 0.6]); % 设置坐标轴样式和网格颜色

% 增加图例
legend({'圆形区域', '盘入螺线', '盘出螺线', '调头路径-段 1', '调头路径-段 2', '成
员轨迹'}, 'FontSize', 12, 'Location', 'best');



clc; clear;
close all;
dtimestep = 0.005; %仿真步长设置
theta = linspace(0, 2*pi, 100);  % 圆周角度，从 0 到 2*pi，100 个点
cicle_r = 500;                          % 圆的半径
xr = cicle_r * cos(theta);              % X 坐标
yr = cicle_r * sin(theta);              % Y 坐标
% 结果矩阵记录
location_record = zeros(14,5);
speed_record = zeros(7,5);

circle_set = 6;
set_pitch = 170;    % 螺距，单位 cm
r0 = circle_set * set_pitch;  % 初始半径，单位 cm
theta = linspace(0,-circle_set * 2 * pi,  1000);  % 角度，从 0 到 16 圈，共 1000
个点
k = set_pitch / (2 * pi);  % 每弧度的半径增量
% 极坐标方程
r_sym = - k * theta;
% 板凳中心划过的曲线
x_sym = -r_sym .* cos(theta);
```

```
y_sym = r_sym .* sin(theta);
cicle_r = r0 + k * theta;
% 板凳中心划过的曲线
cicle_x2 = (cicle_r) .* cos(theta);
cicle_y2 = (cicle_r) .* sin(theta);

r1 = 858/3;  % 第一段圆弧的半径
r2 = 858/6;  % 第二段圆弧的半径，是第一段的一半

% 圆弧旋转角度计算，dy_dx 表示 y 方向与 x 方向的斜率，jiao 是计算得到的旋转角度
dy_dx = 68*2/424/2;
jiao = atan(dy_dx);  % 得到弧度制的旋转角度，用于后续的角度范围调整

% 定义第一段圆弧的角度范围，linspace 生成从-pi+jiao 到-2*pi+jiao 之间的 1000 个点的向量
theta_arc1 = linspace(-pi+jiao, -2*pi+jiao, 1000);
% 根据角度 theta_arc1，计算第一段圆弧的局部坐标
arc1_x_local = r1 * cos(theta_arc1) - r2 * cos(jiao);  % x 方向坐标，平移至调头区域
arc1_y_local = r1 * sin(theta_arc1) - r2 * sin(jiao);  % y 方向坐标，平移至调头区域
% 计算第一段圆弧的半径
r_arc1 = sqrt((arc1_x_local).^2 + (arc1_y_local).^2);

% 定义第二段圆弧的角度范围
theta_arc2 = linspace(-3*pi+jiao, -2*pi+jiao, 500);
% 计算第二段圆弧的局部坐标，调整位置使其与第一段相切
arc2_x_local = r2 * cos(theta_arc2) + r1 * cos(jiao);
arc2_y_local = r2 * sin(theta_arc2) + r1 * sin(jiao);
% 计算第二段圆弧的半径
r_arc2 = sqrt((arc2_x_local).^2 + (arc2_y_local).^2);

% 将两段圆弧的坐标拼接在一起
x_sing = [arc1_x_local, arc2_x_local];  % 合并 x 方向坐标
y_sing = [arc1_y_local, arc2_y_local];  % 合并 y 方向坐标

% 定义龙头和龙身之间的长度
L1 = 341-55;  % 龙头与第一个龙身成员之间的长度
L = 220-55;   % 各个龙身成员之间的长度

% 初始化速度信息
v0 = 100;  % 初始速度，单位为 cm/s（2m/s），限制在不超过 2m/s
w0 = v0 / r0;  % 初始角速度
```

```
N = 224;  % 总成员数，第 223 个成员为龙尾前，224 为龙尾后

% 初始化每个成员的位置信息（x, y, 和各自走过的距离 S）
x_location = r0 * ones(1, N);  % x 方向初始位置，全部初始化为圆弧半径 r0
y_location = zeros(1, N);      % y 方向初始位置，全部为 0


% 初始化板凳间的夹角 alpha，线速度 v_L，木板速度 v_M，角速度 w
alpha = 2 * asin(0.5 * L / r0) * ones(1, N);  % 每个成员之间的夹角，使用 sin 函数计算
speed_S = zeros(1, N);  % 初始化线速度
speed_M = zeros(1, N);  % 初始化木板速度
w = zeros(1, N);    % 初始化角速度

% 初始化成员的位置极坐标信息（角度 theta 和半径 rho）
theta0 = 0;  % 初始角度为 0
thetaN = -circle_set * 2 * pi;  % 结束时的角度，表示完成多圈旋转
theta = theta0 * ones(1, N);  % 每个成员的角度信息，初始化为 theta0
rho = r0 * ones(1, N);  % 每个成员的极坐标半径，初始化为 r0

flag = zeros(1, N);  % 用于标识是否到达某个特殊状态的标志变量
flag_turn = zeros(1, N);  % 用于标识是否开始掉头的标志变量
t2 = zeros(1, N);  % 掉头时的时间初始化

%% 迭代过程
iter_j =0;
num = 1; % 进入了多少个板凳
alpha(1) = 2*asin(0.5*L1/r0);
speed_S(1) = v0; % 1m/s
w(1) = speed_S(1)/r0;
% 龙头是否成功抵达边界的标志，到达边界的时间，开始调头的时间
t_arrive = 0; t_turn = 0;
stop=0;

tmin = 58;
tmax = 260;
dT = 13.48/v0*100;% 用时
for t = 0:dtimestep:tmax
    if (-theta(num)) >= alpha(num)
        num = num +1 ;
        if num > N
            num = N;
        end
    end
```

```
    for i = 1:num
        % 位置更新
        if rho(i)>428.8  % 螺线
            flag(i)=0;
            if t2(i)>=dT % 离开圆弧
                theta(i) = theta(i) + w(i)*dtimestep;
                rho(i) = r0 + k * theta(i);
                x_location(i) = -rho(i) * cos(theta(i));
                y_location(i) = -rho(i) * sin(theta(i));
            else    % 进入圆弧前
                theta(i) = theta(i) - w(i)*dtimestep;
                rho(i) = r0 + k * theta(i);
                x_location(i) = rho(i) * cos(theta(i));
                y_location(i) = rho(i) * sin(theta(i));
            end
        else % 进入圆弧 1
            t2(i) = t2(i)+dtimestep;
            if t2(i)>=dT
                t2(i)=dT;
                rho(i)=428.9;
            end
            x_location(i)=x_sing( round(t2(i)/dT*1500));
            y_location(i)=y_sing( round(t2(i)/dT*1500));
        end
        % 速度更新，先更新夹角，再更新速度。木板速度 M，线速度 L
        if i==1
            alpha(1) = 2*asin(0.5*L1/rho(1));
            speed_S(1) = v0;
            w(1) = speed_S(1)/rho(1);
            speed_M(1) = speed_S(1)*cos(0.5*alpha(1));
        end
        if i >= 2
            alpha(i) = 2*asin(0.5*L/rho(i));
            speed_S(i) = v0;
            w(i) = speed_S(i)/rho(i);
            speed_M(i) = speed_S(i)*cos(0.5*alpha(i));
        end


list_position_get=[1,3,5,7,9,11,13;2,4,6,8,10,12,14;1,2,52,102,152,202,224]
;
        if t>tmin && abs(t-tmin)<1e-1 % -100s
            iter_j = 1;
```

```
            for num_j=1:size(list_position_get,2)
                location_record(list_position_get(1,num_j),iter_j) =
x_location(list_position_get(3,num_j));
location_record(list_position_get(2,num_j),iter_j) =
y_location(list_position_get(3,num_j));
            end

            speed_record(1,iter_j) = speed_M(1); speed_record(2,iter_j) =
speed_M(2);
            speed_record(3,iter_j) = speed_M(52); speed_record(4,iter_j) =
speed_M(102);
            speed_record(5,iter_j) = speed_M(152); speed_record(6,iter_j) =
speed_M(202); speed_record(7,iter_j) = speed_M(224);
        elseif t>tmin && abs(t-tmin-50)<1e-1 % -50s
            iter_j = 2;
            for num_j=1:size(list_position_get,2)
                location_record(list_position_get(1,num_j),iter_j) =
x_location(list_position_get(3,num_j));
location_record(list_position_get(2,num_j),iter_j) =
y_location(list_position_get(3,num_j));
            end
            speed_record(1,iter_j) = speed_M(1); speed_record(2,iter_j) =
speed_M(2);
            speed_record(3,iter_j) = speed_M(52); speed_record(4,iter_j) =
speed_M(102);
            speed_record(5,iter_j) = speed_M(152); speed_record(6,iter_j) =
speed_M(202); speed_record(7,iter_j) = speed_M(224);
        elseif t>tmin && abs(t-tmin-100)<1e-1 % 0s
            iter_j = 3;
            for num_j=1:size(list_position_get,2)
                location_record(list_position_get(1,num_j),iter_j) =
x_location(list_position_get(3,num_j));
location_record(list_position_get(2,num_j),iter_j) =
y_location(list_position_get(3,num_j));
            end
            speed_record(1,iter_j) = speed_M(1); speed_record(2,iter_j) =
speed_M(2);
            speed_record(3,iter_j) = speed_M(52); speed_record(4,iter_j) =
speed_M(102);
            speed_record(5,iter_j) = speed_M(152); speed_record(6,iter_j) =
speed_M(202); speed_record(7,iter_j) = speed_M(224);
        elseif t>tmin && abs(t-tmin-150)<1e-1 % 50s
            iter_j = 4;
            for num_j=1:size(list_position_get,2)
```

```
                location_record(list_position_get(1,num_j),iter_j) =
x_location(list_position_get(3,num_j));
location_record(list_position_get(2,num_j),iter_j) =
y_location(list_position_get(3,num_j));
            end
            speed_record(1,iter_j) = speed_M(1); speed_record(2,iter_j) =
speed_M(2);
            speed_record(3,iter_j) = speed_M(52); speed_record(4,iter_j) =
speed_M(102);
            speed_record(5,iter_j) = speed_M(152); speed_record(6,iter_j) =
speed_M(202); speed_record(7,iter_j) = speed_M(224);
        elseif t>tmin && abs(t-tmin-200)<1e-1 % 100s
            iter_j = 5;
            for num_j=1:size(list_position_get,2)
                location_record(list_position_get(1,num_j),iter_j) =
x_location(list_position_get(3,num_j));
location_record(list_position_get(2,num_j),iter_j) =
y_location(list_position_get(3,num_j));
            end
            speed_record(1,iter_j) = speed_M(1); speed_record(2,iter_j) =
speed_M(2);
            speed_record(3,iter_j) = speed_M(52); speed_record(4,iter_j) =
speed_M(102);
            speed_record(5,iter_j) = speed_M(152); speed_record(6,iter_j) =
speed_M(202); speed_record(7,iter_j) = speed_M(224);
        end
        if stop
            break;
        end
    end % i 循环
    if stop
        break;
    end
end
save_6_locate = round(location_record/100, 6); % result4 文件
save_6_speed = round(speed_record/100, 6); % result4 文件
max_speed = max(save_6_speed(:)); % 展平成向量，查看最大值
disp(max_speed); % 显示最大速度


%%
% 绘制包含龙头和龙身运动轨迹的曲线图

figure(1);
```

```
% 绘制填充区域（如背景或路径）
fill(xr, yr, 'y', 'FaceAlpha', 0.3); hold on; % 使用透明度让背景更柔和

% 绘制基本路径线
plot(cicle_x2, cicle_y2, 'color', 'g', 'LineWidth', 1.5); hold on;
plot(x_sym, y_sym, 'r', 'LineWidth', 1.5); hold on;

% 绘制调头路径的 S 形曲线
plot(arc1_x_local, arc1_y_local, 'color', '#92A8D1', 'LineWidth', 8); hold on; %
使用柔和的蓝灰色
plot(arc2_x_local, arc2_y_local, 'color', '#FF6347', 'LineWidth', 8); % 使用
醒目的橙红色

% 绘制运动轨迹
plot(x_location(2:N), y_location(2:N), '-bo', 'LineWidth', 1.5,
'MarkerFaceColor', 'b'); hold on; % 蓝色带圆圈的轨迹
plot(x_location(1:2), y_location(1:2), '-', 'color', 'k', 'LineWidth', 5); %
起始部分加粗的黑线

% 设置坐标轴等比例显示
axis equal;

% 标题设置，动态显示节数和时间
disp(sprintf('结果曲线 1 节龙头+%d 节龙身运动轨迹（在第%.1f 秒结果）', num-1, t),
'FontWeight', 'bold', 'FontSize', 14);

% 设置坐标轴标签
xlabel('X 方向 (cm)', 'FontWeight', 'bold', 'FontSize', 12);
ylabel('Y 方向 (cm)', 'FontWeight', 'bold', 'FontSize', 12);

% 启用网格线，增加对比度
grid on;

% 设置图窗大小和位置
set(gcf, 'Position', [400, 50, 900, 700]);

% 设置坐标轴字体大小
set(gca, 'FontSize', 12);

% 添加图例，便于区分不同轨迹
legend({'背景区域', '基本路径', '对称路径', 'S 形调头路径 1', 'S 形调头路径 2', '
运动轨迹'}, 'Location', 'BestOutside');

% 调整图形外观
```

```
set(gca, 'Box', 'on', 'LineWidth', 1.2); % 使坐标框线更明显
```

```
# import numpy as np
# import sympy as sp
# from scipy.optimize import root

# width = 30 / 100
# max_r = 4.5

# # 定义常量 a 的值
# a_val = 1.7 / 2 * np.pi

# EPLISON = 1e-5

# # 定义符号变量
# a, theta, theta1, theta2, R = sp.symbols("a theta theta1 theta2 R")

# # 阿基米德螺线方程
# x = a * theta * sp.cos(theta)
# y = a * theta * sp.sin(theta)

# # 阿基米德螺线的导数
# dx_dtheta = a * (sp.cos(theta) - theta * sp.sin(theta))
# dy_dtheta = a * (sp.sin(theta) + theta * sp.cos(theta))

# # 切线方向的法线方向（垂直）
# norm = sp.sqrt(dx_dtheta**2 + dy_dtheta**2)
# nx = -dy_dtheta / norm
# ny = dx_dtheta / norm

# # 圆心坐标
# x0 = x + R * nx
# y0 = y + R * ny

# # 将 theta 替换为 theta1 和 theta2
# r1 = R
# r2 = r1 / 2

# # 计算圆心坐标
# x1 = x0.subs(theta, theta1).subs(R, r1)
# y1 = y0.subs(theta, theta1).subs(R, r1)
# x2 = x0.subs(theta, theta2).subs(R, r2)
# y2 = y0.subs(theta, theta2).subs(R, r2)
```

```
# # 定义符号方程式
# eq = (x1 + x2) ** 2 + (y1 + y2) ** 2 - (3 * r2) ** 2

# # 将符号表达式转为数值函数，供 SciPy 使用
# f_numeric = sp.lambdify([theta1, theta2, a, R], eq, modules=["numpy"])

# # 生成一系列 theta1 值
# theta1_vals = np.linspace(2 * 2 * np.pi, (9 / 1.7) * np.pi, 100)
# r1_vals = np.linspace(width / 2, max_r, 100)

# # 用于存储 theta2 的解
# theta2_solutions = []


# # 定义目标函数，用于 root 求解
# def equation(theta2, theta1_val, r1_val):
#     return f_numeric(theta1_val, theta2, a_val, r1_val)


# # 遍历每个 theta1 和 r1 值，使用 root 求解对应的 theta2
# for theta1_val, r1_val in zip(theta1_vals, r1_vals):
#     initial_guess = 2.0  # 初始猜测值
#     while True:
#         try:
#             # 使用 root 函数求解
#             sol = root(
#                 equation,
#                 initial_guess,
#                 args=(theta1_val, r1_val),
#                 method="hybr",
#                 tol=1e-10,
#                 options={"maxfev": 10000},
#             )

#             # 检查解是否成功，并且 theta2 为正值
#             if sol.success and sol.x[0] > 0:
#                 theta2_solutions.append(sol.x[0])
#                 break  # 成功找到解，跳出循环
#             else:
#                 # 如果解失败或为负值，增加初始猜测值，重新求解
#                 initial_guess += 0.5

#         except Exception as e:
#             print(f"Failed to solve for theta1 = {theta1_val}: {e}")
```

```
#             theta2_solutions.append(None)
#             break

# # 输出 theta2 解
# print(theta2_solutions)

# # 将解带入方程，检查是否满足
# for theta1_val, r1_val, theta2_val in zip(theta1_vals, r1_vals,
theta2_solutions):
#     if theta2_val is not None:
#         # 计算 r2
#         r2_val = r2.subs(theta, theta1_val).subs(R, r1_val)
#         # 计算 x1, y1, x2, y2
#         x1_val = x1.subs(theta1, theta1_val).subs(r1, r1_val).subs(a,
a_val)
#         y1_val = y1.subs(theta1, theta1_val).subs(r1, r1_val).subs(a,
a_val)
#         x2_val = x2.subs(theta2, theta2_val).subs(r1, r1_val).subs(a,
a_val)
#         y2_val = y2.subs(theta2, theta2_val).subs(r1, r1_val).subs(a,
a_val)
#         print(
#             f"theta1 = {theta1_val}, r1 = {r1_val}, theta2 =
{theta2_val},r2 = {r2_val}"
#         )
#         print(f"x1 = {x1_val}, y1 = {y1_val}, x2 = {x2_val}, y2 = {y2_val}")
#         print(f"Equation value: {f_numeric(theta1_val, theta2_val, a_val,
r1_val)}")
#         print("")
import random
import numpy as np
import sympy as sp
from scipy.optimize import minimize

width = 30 / 100
max_r = 4.5

# 定义常量 a 的值
a_val = 1.7 / 2 * np.pi

# 定义符号变量
a, theta, theta1, theta2, R = sp.symbols("a theta theta1 theta2 R")

# 阿基米德螺线方程
```

```
x = a * theta * sp.cos(theta)
y = a * theta * sp.sin(theta)

# 阿基米德螺线的导数
dx_dtheta = a * (sp.cos(theta) - theta * sp.sin(theta))
dy_dtheta = a * (sp.sin(theta) + theta * sp.cos(theta))

# 切线方向的法线方向（垂直）
norm = sp.sqrt(dx_dtheta**2 + dy_dtheta**2)
nx = -dy_dtheta / norm
ny = dx_dtheta / norm

# 圆心坐标
x0 = x + R * nx
y0 = y + R * ny

# 将 theta 替换为 theta1 和 theta2
r1 = R
r2 = r1 / 2

# 计算圆心坐标
x1 = x0.subs(theta, theta1).subs(R, r1)
y1 = y0.subs(theta, theta1).subs(R, r1)
x2 = x0.subs(theta, theta2).subs(R, r2)
y2 = y0.subs(theta, theta2).subs(R, r2)

# 定义符号方程式
eq = (x1 + x2) ** 2 + (y1 + y2) ** 2 - (3 * r2) ** 2

# 将符号表达式转为数值函数，供 SciPy 使用
f_numeric = sp.lambdify([theta1, theta2, a, R], eq, modules=["numpy"])

# 生成一系列 theta1 值
theta1_vals = np.linspace(2 * 2 * np.pi, (9 / 1.7) * np.pi, 100)
r1_vals = np.linspace(width / 2, max_r, 100)

# 用于存储 theta2 的解
theta2_solutions = []


# 定义目标函数，用于 minimize 求解
def equation(theta2, theta1_val, r1_val):
    return np.abs(f_numeric(theta1_val, theta2, a_val, r1_val))
```

```
# 遍历每个 theta1 和 r1 值，使用 minimize 求解对应的 theta2
for theta1_val, r1_val in zip(theta1_vals, r1_vals):
    initial_guess = random.uniform(0, (9 / 1.7) * np.pi)  # 初始猜测值随机
扰动

    try:
        # 使用 minimize 函数求解，采用 BFGS 方法
        sol = minimize(
            equation,
            initial_guess,
            args=(theta1_val, r1_val),
            method="BFGS",
            tol=1e-5,
            options={"maxiter": 1000},  # 限制最大迭代次数，避免无效迭代
        )

        # 检查解是否成功，并且 theta2 为正值
        if sol.success and sol.x[0] > 0:
            theta2_solutions.append(sol.x[0])
        else:
            theta2_solutions.append(None)

    except Exception as e:
        print(f"Failed to solve for theta1 = {theta1_val}: {e}")
        theta2_solutions.append(None)

# 输出 theta2 解
print(theta2_solutions)

# 将解带入方程，检查是否满足
for theta1_val, r1_val, theta2_val in zip(theta1_vals, r1_vals,
theta2_solutions):
    if theta2_val is not None:
        # 计算 r2
        r2_val = r2.subs(theta, theta1_val).subs(R, r1_val)
        # 计算 x1, y1, x2, y2
        x1_val = x1.subs(theta1, theta1_val).subs(R, r1_val).subs(a, a_val)
        y1_val = y1.subs(theta1, theta1_val).subs(R, r1_val).subs(a, a_val)
        x2_val = -x2.subs(theta2, theta2_val).subs(R, r1_val).subs(a,
a_val)
        y2_val = -y2.subs(theta2, theta2_val).subs(R, r1_val).subs(a,
a_val)
        print(
```

```
            f"theta1 = {theta1_val}, r1 = {r1_val}, theta2 = {theta2_val},r2
= {r2_val}"
        )
        print(f"x1 = {x1_val}, y1 = {y1_val}, x2 = {x2_val}, y2 = {y2_val}")
        print(f"Equation value: {f_numeric(theta1_val, theta2_val, a_val,
r1_val)}")
        print("")
```