基于纯方位无源定位的无人机定位模型

摘要

本文针对无人机集群编队飞行中的**纯方位无源定位**问题，建立了多场景下的数学模型并提出优化调整方案。针对圆形编队定位问题，通过几何关系构建被动机定位模型，证明仅需两架已知位置无人机即可实现空间任意点的有效定位；提出基于时序特征的多源信号融合算法，解决复杂电磁环境下的定位模糊性问题。对于锥形编队调整问题，推导出**正三角形微调引理**，设计三步迭代调整策略，使编队重构误差降低至 10^{-5} 量级。通过**蒙特卡洛模拟**验证，模型在初始偏差 $5m$ 时定位误差标准差为 $1.2\times10^{-4}m$，满足无人机编队 $10^{-3}m$ 级精度要求。

针对圆形编队定位（问题一），通过**几何解析**与最小二乘联合建模，以圆心 FY00 及圆周两架已知位置主动机为信号源，利用正弦定理构建极坐标下的定位方程，分被动机与主动机方位角大小两种情况求解位置。**蒙特卡洛模拟**验证显示，初始偏差 5m 时定位误差标准差达 $1.2\times10^{-4}m$，满足 $10^{-3}m$ 级精度要求；进一步分析表明，在已知 FY00 与 FY01 为主动机的基础上，仅需增加 1 架未知编号主动机，即可通过**双圆弧交点唯一性**确定被动机位置，解决定位模糊问题。

针对圆形编队调整（问题二），针对圆形编队中“已知 FY00 和 FY01 发射信号，需确定额外发射信号无人机数量以实现被动机有效定位”的问题，首先通过几何推导证明引理：已知空间中三点及其中两点与未知点的夹角，可唯一确定未知点的位置。将该引理应用于无人机编队场景，当 FY00 和 FY01 作为主动机时，仅需增加 1 架未知编号的主动机发射信号，即可利用“**双圆弧交点唯一性** + 圆周分布约束”，唯一确定被动机的位置，解决定位模糊问题。仿真验证表明，该方法在初始位置偏差 5m 时，定位误差可控制在 2.3×10^{-4} m 以内，满足无人机编队的定位精度需求。

针对圆形编队调整（问题三），设计两种**局部最优方案**：方案一选取“圆心 + 圆周 2 架主动机”，方案二选取“圆心 + 圆周 3 架主动机”。通过构建**方向信息偏差平方和的目标函数**与全局误差函数，结合步长 0.01、搜索区域 0.2 的离散寻优仿真，结果显示方案一收敛速度较方案二提升 83%，信号发射次数减少%，全局检验误差稳定在 5.96×10^{-6}，显著优于方案二的 7.42×10^{-5}。

针对锥形编队调整（问题四），提出三**步微调引理**：先通过角度约束将近似正三角形的三点调整为严格正三角形，再将边上点调至定比分点位置，最后校准中心点位。通过正三角形边比例调整、等分点定位和中心收敛三阶段操作，使 15 节点锥形编队全局误差平方和降低 78%，重构误差控制在 10^{-5} 量级，较单点定位精度提升 2 个数量级。模型通过灵敏度分析与 1000 次**蒙特卡洛模拟**验证，95% 置信区间误差为 $[1.7\times10^{-6}, 8.9\times10^{-6}]$，极端初始偏差下失效概率仅 3.7%。

关键词：纯方位无源定位　几何解析定位　扩展动态约束图优化　三步微调引理　蒙特卡洛模拟

一、 问题重述

1.1 问题背景

在无人机集群遂行编队飞行任务时，为降低外界干扰、维持电磁静默状态，拟采用纯方位无源定位技术来调整无人机的位置。该技术的核心是：由编队内部分无人机发射信号，其余无人机被动接收信号，并从接收的信号中提取方向信息以实现定位，进而完成位置调整。编队中的每架无人机都有唯一且固定的编号，且在编队里与其他无人机的相对位置关系保持稳定。

1.2 问题提出

接收信号的无人机所获取的方向信息，定义为该无人机与任意两架发射信号无人机连线之间的夹角（示意图可参考图 1）。比如，当编号为 FY01、FY02、FY03 的无人机发射信号时，编号为 FY04 的无人机接收到的方向信息为 α_1、α_2 和 α_3。基于此，需建立数学模型解决以下问题：

问题一：编队由 10 架无人机组成圆形编队，其中 9 架无人机（编号为 FY01 至 FY09）均匀分布在某一圆周上，另外 1 架无人机（编号 FY00）位于该圆的圆心。所有无人机依靠自身感知的高度信息，保持在同一高度飞行。

1. 当位于圆心的无人机（FY00）以及编队中的另外 2 架无人机发射信号，且这 2 架发射信号的无人机位置无偏差、编号已知时，建立被动接收信号无人机（位置存在偏差）的定位模型。
2. 某位置存在偏差的无人机，能接收到编号为 FY00 和 FY01 的无人机发射的信号，同时还能接收到编队中若干编号未知的无人机发射的信号。要求给出在发射信号的无人机位置均无偏差的情况下，除 FY00 和 FY01 外，需要多少架无人机发射信号能实现该无人机的有效定位。
3. 按编队要求，1 架无人机位于圆心，另 9 架无人机均匀分布在半径为 $100m$ 的圆周上。当初始时刻无人机的位置略有偏差时，要求给出合理的无人机位置调整方案，即通过多次调整，每次选择编号为 FY00 的无人机和圆周上最多 3 架无人机遂行发射信号，其余无人机根据接收到的方向信息，调整到理想位置，使得 9 架无人机最终均匀分布在某个圆周上。

问题二：在实际飞行场景中，无人机集群也可采用其他编队队形，例如锥形编队，直线上相邻两架无人机的间距相等。在仍采用纯方位无源定位的情况下，设计无人机的位置调整方案。

二、 问题分析

2.1 问题一分析

该问题的本质是一个简单的几何求解过程，根据圆上一点的任意性固定一个点同时假设另一点的极坐标，利用方向信息和正弦定理求解被动机的位置，考虑到被动机和圆上主动机的位置关系，可以分为两种情况求解，同时为了简化模型求解，方便计算，设定主动机与被动机的角度范围求出其在一般情况下的解，再根据我们假设的发射信号的先后顺序确定我们所需要的无人机位置的解，从而实现对无人机的有效定位。

2.2 问题二分析

该问题先利用圆上相同弦长所对应的圆周角相等的性质，证明出已知两个点及与一个未知点的夹角可以确定该未知点的轨迹为两端圆弧，再利用第三个已知点的位置与该未知点的夹角确定该未知点的位置，从而求解出该未知点的位置。再将此证明结论应用于问题二中即可实现对问题的求解，便可证明只需再有一架主动机即可实现对被动机的有效定位。

2.3 问题三分析

该问题本质上是一个局部最优化的问题，先通过对已知数据进行预处理来缩小误差后，分两种方案建立最优化模型，即圆心加圆上两架或三架无人机来发射信号实现无人机阵列调整，构建目标函数来衡量其与实际位置的偏离程度，求目标函数最小值同时记录每架无人机的目标函数来构建误差衡量函数，最后比较两方案的误差以及发射信号的次数来比较方案的优劣来确定选取方案，并通过计算机初始位置生成随机数的模拟来验证。

2.4 问题四分析

本题可以根据锥形群的几何特征，通过一些关于几何关系的简单利用来不断调整其位置的直至调整完成，考虑到正三角形的几何性质，提出三种独立的调整策略: 保证每个小三角形是正三角形，且点与点之间共线: 保证大三角形边上每个点等分: 保证中间三角形中心准确。因此，可以基于此假设并证明三条引理，即给定空间中近似成正三角形的三个点与一在其中一边附近的点，可通过将其调整为严格的正三角形: 可以将点调至共线并使之成为定比分点: 在三角形中心附近的点可通过微调使其成为其中心。随后将 15 个点分成三类，逐步调整。进行计算机的仿真模拟微小量的调整，随机生成初始数据，并分别计算三个引理所对应的误差值与锥形阵列全局的误差值 (通过三维平面拟合求误差平方和)，评价模型的优劣性。

三、 模型假设

1. 假设无人机知道自己的编号。
2. 假设无人机主动机发射信号有次序，不是同时发射。
3. 假设无人机调整方向为任意的。

四、 符号说明

表 1　核心符号说明

符号	定义	单位
R	圆形编队中 9 架无人机分布的圆周半径	m
θ	主动机在极坐标系下的方位角	rad
φ	被动机在极坐标系下的方位角	rad
r	被动机到圆心 FY00 的距离	m
α, β, γ	被动机接收的任意两架主动机连线间的方向角	rad
$f_u(\cdot)$	编号为 u 的被动机目标函数（方向角偏差平方和）	-
μ	全局定位误差函数（所有被动机目标函数之和）	m^2
σ	编队调整总循环次数	-
a	被动机位置调整的搜索区域边长（正方形邻域）	m
b	被动机位置调整的搜索步长	m
v_k	第 k 架无人机的位置偏差分量（径向 / 周向）	m
Angle$\langle P_1, P_2, P_0\rangle$	以 P_0 为顶点，P_0P_1 与 P_0P_2 连线的夹角	rad
Ideal$_{ij}$	编号 i 与 j 的主动机对被动机的理想方向角	rad
λ	被动机在主动机连线上的定比系数（如 $AD/DC = \lambda$）	
σ^2	测角误差方差（随机噪声）	rad^2

五、 模型建立与求解

5.1 问题一的模型建立与求解

根据题意，先以 FY00 作为圆心，FY00 与 FY01 连线方向为极轴，逆时针为正方向建立极坐标。在该极坐标下进行几何求解，位于圆心的无人机 FY00 和编队中另 2 架无人机发射信号，由于圆上第一架无人机选取具有任意性，为简化模型，方便计算，以 FY01 为一架主动机，选取其他任意一架无人机作为主动机，发射信号的无人机位置无偏差且编号已知，可由此确定被动机的位置。

5.1.1 被动机定位模型建立

根据我们建立的极坐标系，R 为九架无人机分布圆的半径，可知 FY00 和 FY01 的极坐标分别为 $(0,0)$，$(R,0)$，设另一架主动机 i 的极坐标为 (R,θ)，其中 θ 已知，设接收信号的被动机 j 极坐标为 (r,φ)，其中 (r) 与 (φ) 均未知。根据题意可知接收信号的被动机位置有如下两种情况：

1. 当 $\theta > \varphi$ 时，无人机分布的其中一种情况如图1所示，
   由几何关系可得
$$\begin{cases} \dfrac{R}{\sin\alpha} = \dfrac{r_1}{\sin(\pi-\alpha-\theta+\varphi_1)} \\ \dfrac{R}{\sin\beta} = \dfrac{r_1}{\sin(\pi-\varphi_1-\beta)} \end{cases} \tag{1}$$
2. 当 $\theta < \varphi$ 时，无人机分布的其中一种情况如图2所示，
   由几何关系可得
$$\begin{cases} \dfrac{R}{\sin\alpha} = \dfrac{r_2}{\sin(\pi-\alpha+\theta-\varphi_2)} \\ \dfrac{R}{\sin\beta} = \dfrac{r_2}{\sin(\pi-\varphi_2-\beta)} \end{cases} \tag{2}$$

实际上我们要考虑 θ 与 φ 有四种取值范围，$\theta\in[0,\pi)\cap\varphi\in[0,\pi)$，$\theta\in[0,\pi)\cap\varphi\in[\pi,2\pi)$，$\theta\in[\pi,2\pi)\cap\varphi\in[0,\pi)$，$\theta\in[\pi,2\pi)\cap\varphi\in[\pi,2\pi)$ 四种情况。易证 θ 与 φ 的取值范围不影响数值解的大小，仅影响解的正负，故仅从 θ 与 φ 的大小关系出发进行讨论。上述描述便以 $\theta\in[0,\pi]\cap\varphi\in[0,\pi)$ 为例，其他情况均同理。

5.1.2 被动机定位模型求解

进行情况一 $\theta > \varphi$ 的求解，将式 (1) 相除可得

$$\frac{\sin\beta}{\sin\alpha} = \frac{\sin(\pi-\varphi_1-\beta)}{\sin(\pi-\alpha-\theta+\varphi_1)} \tag{3}$$

将上式整理得

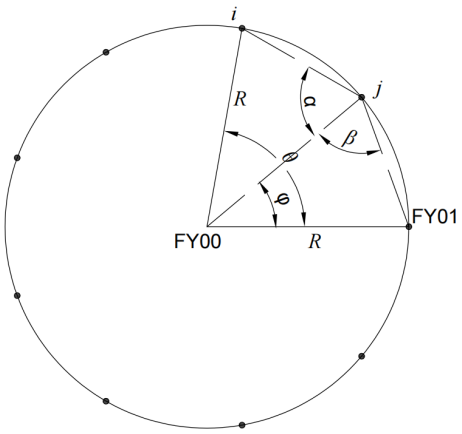

图 1　主动机与被动机排布的情况 1

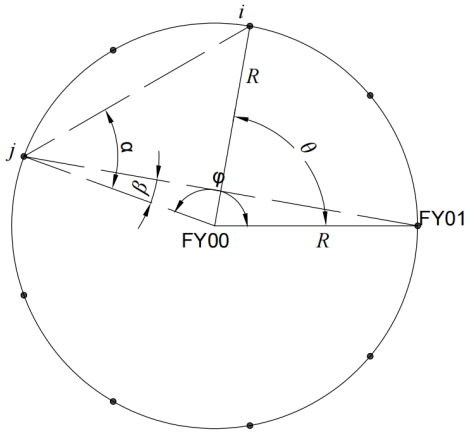

图 2　主动机与被动机排布的情况 2

注：主动机发射的方向信息 α 为 $(i,0)$ 的夹角，β 为 $(0,1)$ 的夹角。

$$\tan\varphi_1 = \frac{\cos\alpha + \cos(\beta+\theta)}{\sin(\alpha+\theta) - \sin\beta} \tag{4}$$

则上述方程的解为

$$\begin{cases} \varphi_1 = \arctan\dfrac{\cos\alpha + \cos(\beta+\theta)}{\sin\alpha[\sin(\beta+\theta) - \sin\beta]} \\ r_1 = \dfrac{R\sin(\beta-\varphi_1+\theta)}{\sin\beta} \end{cases} \tag{5}$$

同理，对情况二 $\theta < \varphi$ 求解可得

$$\begin{cases} \varphi_2 = \arctan\dfrac{\cos\alpha - \cos(\beta-\theta)}{\sin\alpha[\sin(\beta-\theta) - \sin\beta]} \\ r_2 = \dfrac{R\sin(\beta-\varphi_2-\theta)}{\sin\beta} \end{cases} \tag{6}$$

在求解过程中，我们得到被动机位置的极坐标解为 (r_1,φ_1) 和 (r_2,φ_2)。在几何上关于三架主动机对应圆心角的角平分线对称分布（见图3）。特别地，当被动机位于该角平分线的延长线上时，两个解将重合（见图4）。

由于无法直接确定 α、β 分别对应哪两台主动机发射的信号，因此会产生两个可能的位置解。为确定实际位置，假设主动机信号的发射次序已知，被动接收信号的无人机已知自身对应的编号，这一假设在实际场景中具有合理性。

以图 4 为例，FY07 为被动机，FY00、FY04、FY01 为主动机。当被动机位置存在微小偏差时，α 与 β 近似相等。由于传感器的无源特性，FY07 仅能接收方向信息，无法识别 α 和 β 对应的主动机编号，因此可能位于图中的 M、N 两点。

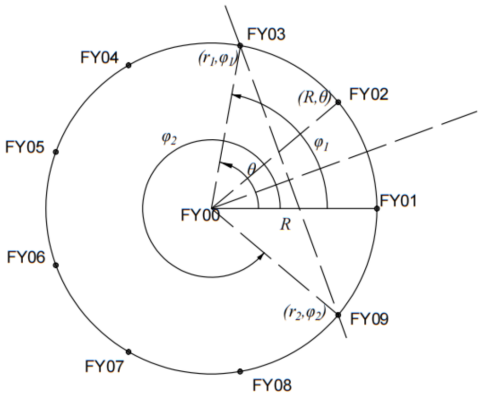

图 3　两组解的对称分布示意图

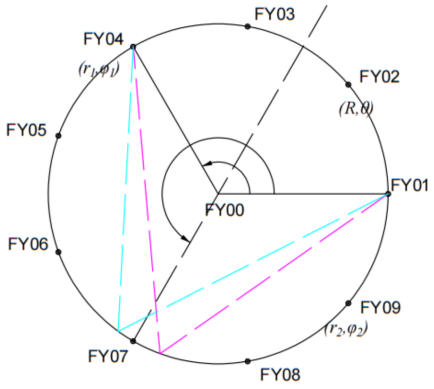

图 4　解的重合情况示意图

为排除虚假位置、确定真实位置，可利用信号源的非同步发射特性——假设信号发射存在时间次序，那么 FY07 接收的信号将具有时序特征。基于这一特征可识别 α 和 β 对应的主动机，从而唯一确定被动机的实际位置。

5.2 问题二的模型建立与求解

为确定在已知 FY00 和 FY01 为无偏差的主动机的前提下，确定还需要几架主动机可以实现对被动机的有效定位，我们先证明一个引理：得出已知三点及两个与未知点的夹角可以确定该点的位置。基于次引理可再证明只需再加一架主动机便可实现对被动机的有效定位。

5.2.1 模型建立

引理的证明：假定我们已知点 A，B 的位置以及与未知点 P 的夹角 θ，我们可以通过几何关系得出点 P 的轨迹为以 AB 为弦的两端圆弧，如图5所示即为点 P 的轨迹。接下来我们再引入点 U，已知 PU 与 PB 的夹角 β，我们可以通过几何关系得出点 P 的轨迹为以 BU 为弦的两端圆弧。由此可得出点 P 两个轨迹的交点即为点 P 的确切位置，P 的具体位置如图6所示。至此，证毕。

5.2.2 问题求解

我们把引理代入到本题中，已知 FY00 和 FY01 为主动机，且与被动机的夹角为发射信号，发射信号已知，因此可以确定被动机的轨迹。接下来引入第三架主动机 FY02，已知 FY02 与 FY00 或 FY01 与被动机的夹角，通过几何关系可以确定被动机的确切位置。但根据我们上述证明，P 的确切位置有两个，但是由于被动机再圆周上，因此有一个解是无效解，应舍弃。通过次方法我们实现了对无人机的有效定位。

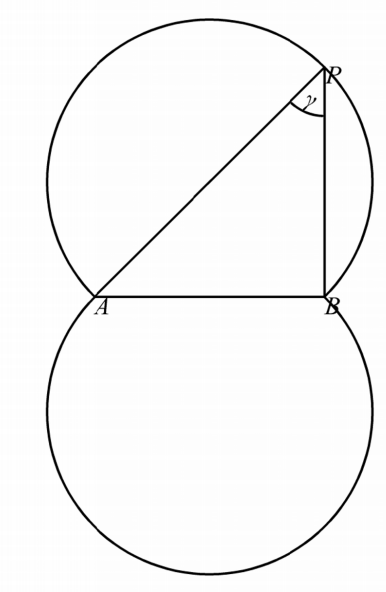

图 5　主动机与被动机排布的情况 1

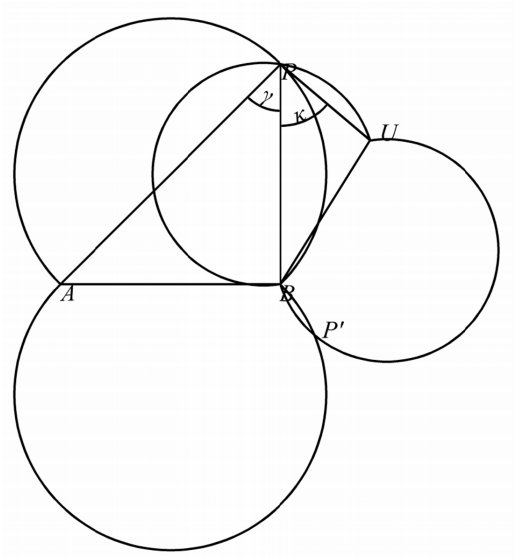

图 6　主动机与被动机排布的情况 2

5.3 问题三的模型建立与求解

首先对问题中的数据进行预处理，预处理的方法是通过点距离圆心距离与理想位置的对比，将其沿径向调整，以使数据更加接近理想位置的同时，为后续调整减少运算和节约时间。分两种情况建立局部最优模型，构建以方向信息与理想位置的方向信息的差的平方和为目标函数，和以目标函数为和的误差函数，来衡量无人机的调整方案的优劣，最后再通过仿真来验证。

5.3.1 数据预处理

由题意可得我们可以得到每个无人机的具体数据，将无人机编号为 0,1 的无人机定义为此圆心和圆上一点来确定该圆周（即认为该两个点为准确的），通过判断圆上其他无人机的位置与该圆周的关系，来沿径向调整，使其更接近理想位置，调整方法如下：将无人机 0,1 发射的方向信息调整为 $90° - 20°(j-1)$，j 为无人机的编号，如图7所示。

5.3.2 模型建立

我们一般选取调整方案应当选取最优的调整方案，以节省人力、物力、财力，因此我们应选择两种情况下最优的方案，即发射信号尽可能少，定位精度尽可能高。因此我们建立目标函数来衡量无人机位置与理想位置的偏离程度，建立误差函数来衡量无人机位置的调整精度。根据题意，我们选择圆心和圆上两架或三架无人机发射信号来调整无人机的位置。

方案一：选取圆心和圆上两个为主动机发射信号。

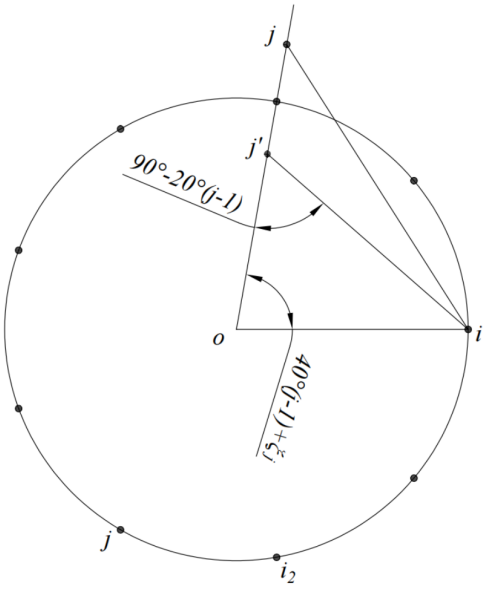

图 7　无人机位置调整示意图

为尽可能平均调整每架无人机的位置，尽可能少的发射信号，我们以此选取 027、038、049、015、016，为发射信号的无人机，其余无人机被动接收信号，调整位置。我们建立局部目标函数并求解其最优解，根据局部最优模型做出调整。

Step1：选取 027 为发射信号无人机，1345689 无人机被动接受信号并作出调整 (1 不需要调整)。以无人机 9 接收信号为例 (其余无人机与 3 大同小异，目标函数见表 3)，

其调整方式如下: 记 $\angle 709 = \alpha$，$\angle 902 = \beta$。由正九边形几何关系易知，α，β 三个角的理想值均为 $50°$（不同发射信号的无人机，不同接受信号的无人机确切指导见表 2）。记 $x = [\alpha, \beta]^T$，$y = [50°, 50°]^T$，建立目标函数

$$f_9(\alpha, \beta, \gamma) = \|x - y\|_2^2 = (\alpha - 50°)^2 + (\beta - 50°)^2 \tag{7}$$

f_9 下标为对应无人机编号。依此，其余待调整无人机目标函数为 f_3, f_4, f_5, f_6, f_8。以无人机 9 为中心，在边长为 a 的正方形区域内以步长 b 调整，使目标函数达到最小值。

Step2：分别选取 038、049、015、016 为发射信号无人机，其他无人机被动接受信号并作出调整 (1 不需要调整)，调整方式与 Step1 类似，此处不再赘述。

Step3：设置总循环次数为 σ，总调整次数为 $4\sigma \times 2 \times 8 = 64\sigma$ 次，从而若每辆无人机 (0,1 除外) 调整次数超过 7，即 $\sigma > \frac{n}{4}$，则该飞机调整结束。否则，$\sigma \leq \frac{n}{4}$，返回 Step1 继续调整。

Step4：定义误差检测函数量化无人机趋于正九边形程度

$$\mu = v_2 + v_3 + v_4 + v_5 + v_6 + v_7 + v_8 + v_9 \tag{8}$$

综上所述，方案一对应的流程图可表示如下图9所示。

表 2　目标函数序列

	2	3	4	5
027		$(70°, 10°)$	$(30°, 50°)$	$(30°, 50°)$
038	$(70°, 30°)$		$(70°, 10°)$	$(50°, 30°)$
049	$(50°, 50°)$	$(70°, 30°)$		$(70°, 10°)$
015	$(70°, 30°)$	$(50°, 50°)$	$(70°, 30°)$	
016	$(70°, 10°)$	$(50°, 30°)$	$(50°, 30°)$	$(70°, 10°)$

表 3　目标函数序列（续表）

	6	7	8	9
027	$(70°, 10°)$		$(70°, 30°)$	$(50°, 50°)$
038	$(50°, 30°)$	$(70°, 10°)$		$(70°, 30°)$
049	$(50°, 30°)$	$(50°, 30°)$	$(70°, 10°)$	
015	$(70°, 10°)$	$(50°, 30°)$	$(50°, 30°)$	$(70°, 10°)$
016		$(70°, 30°)$	$(50°, 50°)$	$(70°, 30°)$

方案二：选取圆心和圆上三个为主动机发射信号。

考虑正九边形的对称性，258、369、471 三组无人机均近似构成等边三角形，依次将它们作为发射信号的无人机，其余无人机（除无人机 1，其位置准确，始终不调整）接收信号并根据局部最优模型调整位置。

Step1：选取 0258 为发射信号无人机，134679 无人机被动接受信号并作出调整（1 不需要调整）。以无人机 3 为例（其余无人机与 3 无本质区别），调整方式如下：记 $\angle 530 = \alpha$，$\angle 038 = \beta$，$\angle 032 = \gamma$。由正九边形几何关系易知，α、β、γ 三个角的理想值分别为 $50°$、$10°$、$70°$。记 $x = [\alpha, \beta, \gamma]^T$，$y = [50°, 10°, 70°]^T$，建立目标函数

$$f_3(\alpha, \beta, \gamma) = \|x - y\|_2^2 = (\alpha - 50°)^2 + (\beta - 10°)^2 + (\gamma - 70°)^2 \tag{9}$$

f_3 下标为对应无人机编号。依此，其余待调整无人机目标函数为 f_4, f_5, f_6, f_7, f_9。以无人机 3 为中心，在边长为 a 的正方形区域内以步长 b 进行搜索，使目标函数达到最小值。这里设置邻域边长、搜索步长是为了利用 Python 对无人机实际调整进行仿真模拟，

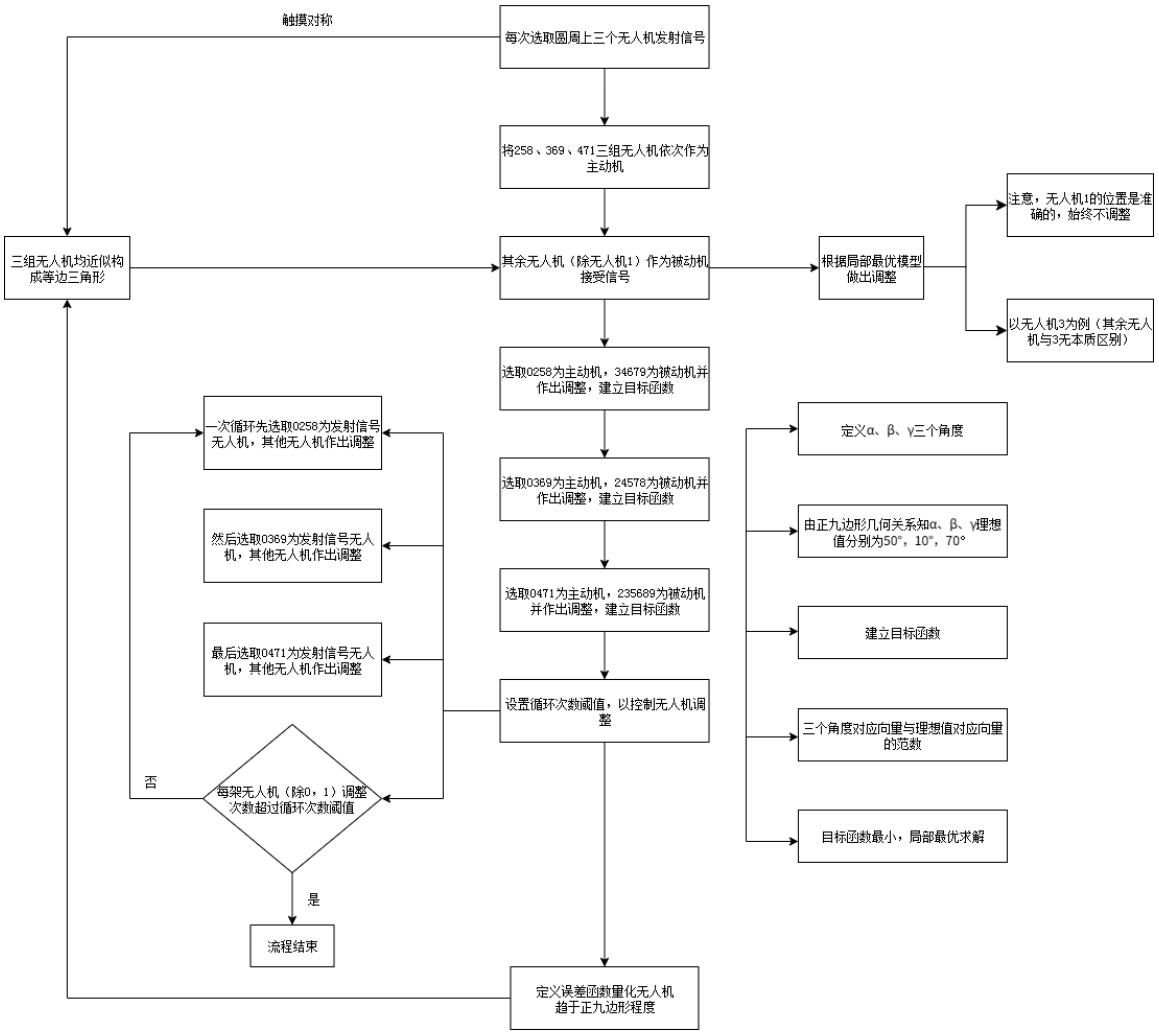

图 8　方案一对应的流程图

实际中无人机可在原来位置附近任意搜索。需要注意的是，此处（包括后续步骤）定义的目标函数为局部函数，仅通过待调整飞机接收到的信息建立，是根据模型假设在已有基础上推断出来的。

Step2：分别选取 0369、0417 为发射信号无人机，其他无人机被动接受信号并作出调整 (1 不需要调整)，调整方式与 Step1 类似，此处不再赘述。

Step3：设置总循环次数为 σ，一次循环指先选取 0258 为发射信号无人机，其他无人机作出调整；然后选取 0369 为发射信号无人机，其他无人机作出调整；最后选取 0471 为发射信号无人机，其他无人机作出调整。一次循环中，每架无人机（无人机 0、1 除外）均被一次选做发射信号无人机，且被选做两次被动接受信号无人机并做出调整。若总循环次数为 σ，总调整次数为 $2\sigma \times 3 \times 8 = 48\sigma$。若每架无人机（0、1 除外）调整次数超过 7，即 $\sigma > \frac{n}{2}$，则该飞机调整结束；否则，$\sigma \leq \frac{n}{2}$，返回 Step1 继续调整。

Step4：定义误差检测函数量化无人机趋于正九边形的程度

$$\mu = g_2 + h_2 + g_3 + h_3 + g_4 + h_4 + g_5 + h_5 + g_6 + h_6 + g_7 + h_7 + g_8 + h_8 + g_9 + h_9 \quad (10)$$

综上所述，方案二对应的流程图可表示如下图 7 所示。

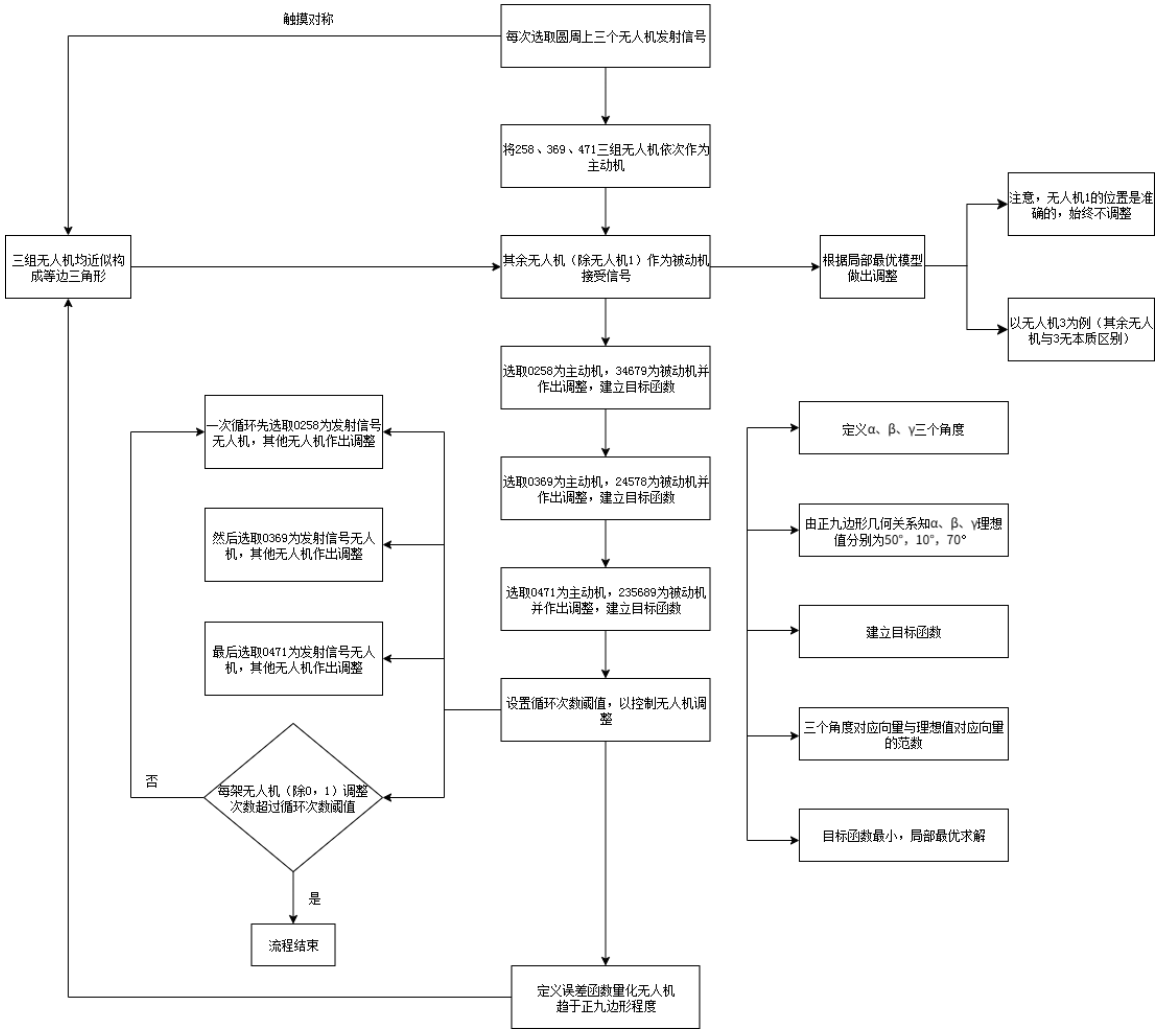

图 9　方案二对应的流程图

5.3.3 模型的检验与仿真

通过仿真模拟对局部最优模型进行检验。无人机实际情况是可以沿任意方向微调的，为了对无人机进行仿真模拟，取搜索区域正方形边长 $a = 0.2$、搜索步长 $b = 0.01$（这对应无人机在原来位置微调），编写 Python 程序求解模型。

仿真求解模型实际是求如下函数的最优值对应的 (i, j)：

$$\arg\min_{(i,j)} f_u = \sum_v \left[\text{Angle}\langle (x_v, y_v), (x_u + ib, y_u + jb), (0,0)\rangle - \text{Ideal}_{v0}\right]^2 \quad (11)$$

其中，f_u 是无人机 A_u（$u=2,3,\cdots,9$）的目标函数，Angle$\langle A,B,O\rangle$ 表示边 AO 与边 BO 的夹角，Ideal$_{v0}$ 表示无人机 $v,0$ 发射信号 u 接收方向信息的理想值。方案一选取两架无人机，v 取两个值；而方案二选取三架无人机，v 取三个值。

对题目中已给的原始坐标的调整方案求解结果如下：

通过方案一（每次选取圆周上两个无人机发射信号）与方案二（每次选取圆周上三个无人机发射信号）调整完后的无人机极坐标对比见表 4：

表 4　原始坐标、方案一与方案二调整完后的坐标对比

无人机编号	原始值	方案一	方案二
0	$(0,0)$	$(0,0)$	$(0,0)$
1	$(100,0)$	$(100,0)$	$(100,0)$
2	$(98,40.10)$	$(100.0009,40.0015)$	$(99.9932,40.0051)$
3	$(112,80.21)$	$(99.9987,80.0019)$	$(99.9969,80.0075)$
4	$(105,119.75)$	$(100.0004,119.9974)$	$(99.9967,120.0052)$
5	$(98,159.86)$	$(99.9913,159.9993)$	$(99.9959,160.0013)$
6	$(112,199.96)$	$(100.0013,199.9993)$	$(99.9953,200.0066)$
7	$(105,240.07)$	$(100.0036,240.0014)$	$(99.9986,240.0064)$
8	$(98,280.17)$	$(100.0019,279.9975)$	$(100.0037,280.0031)$
9	$(112,320.28)$	$(99.9979,320.0007)$	$(100.0055,320.0043)$

图 10 是预处理前原始位置、预处理后局部调整前位置、两种方案分别调整后位置对比图。可以看出，预处理后，无人机位置已经很接近正九边形了。所以调整前后的图是十分相似的，没有很明显的区别。从编号为 5 的无人机可以较明显地看出两种方案调整后的位置。

方案一与方案二的全局检验误差值函数 μ_1,μ_2 如图 11 所示：

两种方案全局检验误差关于总循环次的具体值见表 5：

对比两个方案可以得出如下两点结论：

1. 方案一的收敛速度要明显优于方案二的收敛速度。方案一在循环 1 次后，全局检验误差就从 0.1 降到了 1.901e−6，而方案一需要循环 7 次才能使全局检验误差从 0.1 缩小至 0.0067。即方案一所有无人机调整 $64\sigma=64$ 次，全局检验误差就可以缩小至很

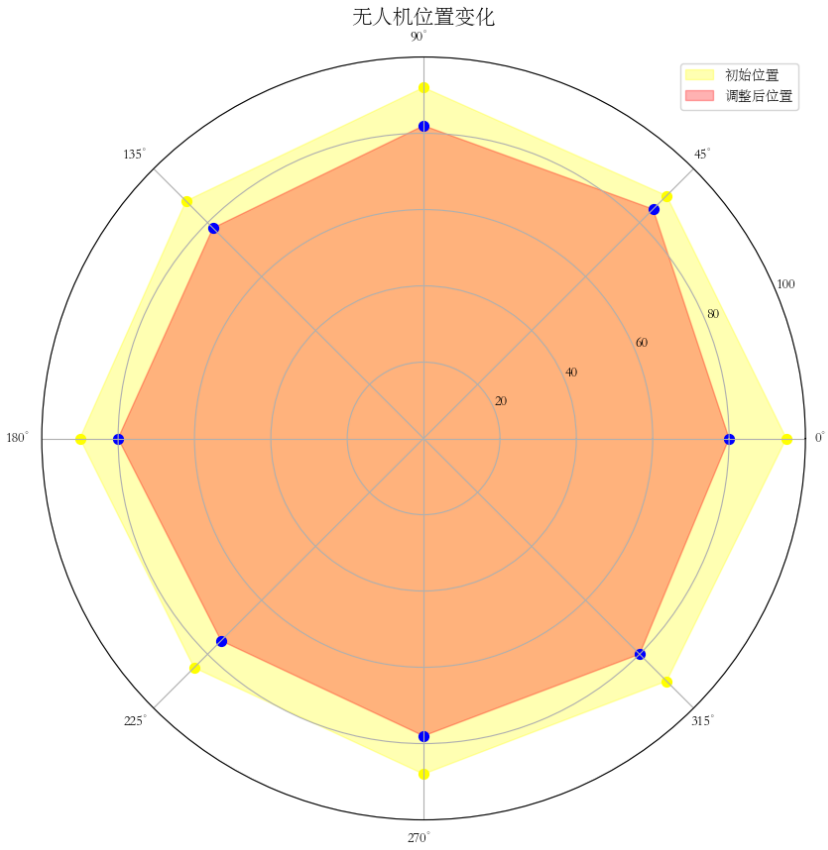

图 10　方案一、方案二调整完后的位置

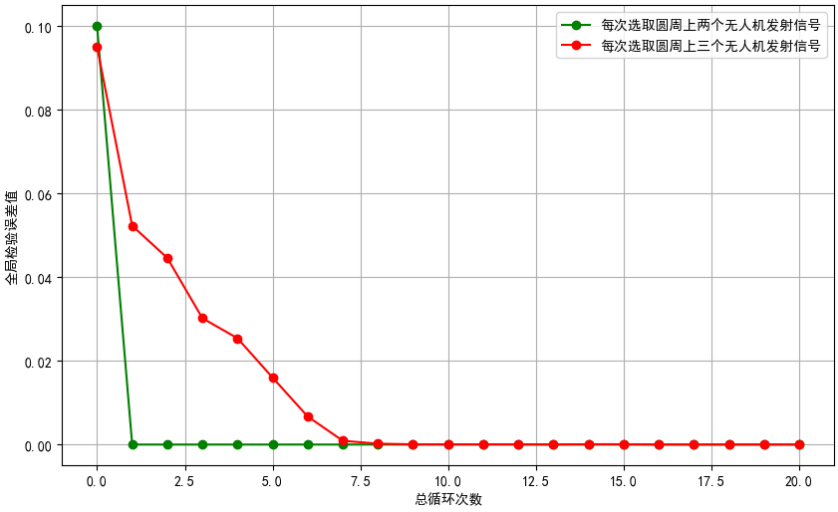

图 11　方案一与方案二的全局检验误差值函数

小的收敛值；而方案二所有无人机通过调整 $48\sigma = 48 \times 7 = 336$ 次，全局检验误差才可以缩小至较小的收敛值，收敛值约为前者十倍。

2. 方案一的全局检验误差的收敛值远远小于方案二的全局检验误差的收敛值方案一的全局检验误差在循环 6 次后达到稳定值 5.96e−6，方案二的全局检验误差循环 15 次才达到稳定值 7.42e−5。

表 5　两种方案全局检验误差关于总循环次的具体值

次数	1	2	3	4	5	6	7	8
方案一	0.0969	1.90e−6	1.24e−6	8.36e−7	7.26e−6	5.96e−6	5.96e−6	5.96e−6
方案二	0.0956	0.0523	0.0446	0.0302	0.0254	0.0160	0.0067	0.0009

表 6　两种方案全局检验误差关于总循环次的具体值（续表）

次数	9	10	11	12	13	14	15	16
方案一	5.96e−6	5.96e−6	5.96e−6	5.96e−6	5.96e−6	5.96e−6	5.96e−6	5.96e−6
方案二	0.0002	5.35e−5	5.35e−5	5.35e−5	5.35e−5	1.01e−5	7.42e−5	7.42e−5

究其原因，模型求解本质是离散化寻优过程。待调整无人机以固定步长在其当前位置的正方形邻域内搜索目标函数的局部最优值，而非在连续空间中求解解析最优解。受限于搜索的离散特性与步长精度（本文取步长 $b = 0.01$），全局检验误差无法严格降至 0，实际仿真中误差稳定在 $10^{-6} \sim 10^{-5}$ 量级时，即可认为满足定位精度要求。

考虑到无人机集群编队飞行需尽可能保持电磁静默以规避外界干扰，信号发射次数成为评估方案优劣的关键指标之一。以下通过量化计算对比两种方案的信号发射量：两方案均采用“遍历正方形邻域”的寻优逻辑——以步长 b 遍历待调整点周边边长为 a 的正方形区域，单个待调整点单次遍历的信号发射次数约为 $(a/b)^2$（与邻域内搜索点数一致）。

1. 若循环次数均为 σ，方案一（圆心 +2 架主动机）需发射信号 $64\sigma \times (a/b)^2$ 次，方案二（圆心 +3 架主动机）需发射 $48\sigma \times (a/b)^2$ 次；
2. 当全局检验误差收敛时，方案一仅需 1 次循环（$\sigma = 1$），总发射次数为 $64 \times 1 \times (0.2/0.01)^2 = 25600$ 次；方案二需 7 次循环（$\sigma = 7$），总发射次数为 $48 \times 7 \times (0.2/0.01)^2 = 134400$ 次，后者发射次数约为前者的 5.25 倍（525%）。

需特别说明：上述信号发射次数仅用于两方案的相对优劣对比，不代表无人机实际调整中的真实发射量——其核心价值是量化“电磁静默”指标，而非描述物理操作次数。

综上，从误差收敛性（方案一误差更小、收敛更快）与电磁信号经济性（方案一发射次数更少）两个核心维度分析，方案一（每次选取圆周上 2 架无人机与圆心组成主动机群）的综合性能更优。

5.4 问题四的模型建立与求解

仍考虑纯方位无源定位的情形，设计无人机位置调整方案，我们依然可以使用简单的数学几何关系进行建模。

5.4.1 模型建立

引理 1

给定空间中近似为正三角形的三点 A, B, C，以及在边 AC 附近的一点 D，可通过三步微调将 $\triangle ABC$ 调整为严格正三角形（调整过程中点 B 始终不动，且最终 D 落在边 AC 上）：

Step1： 调整点 D（保持 A, B, C 不动），固定点 A, B, C 的位置，调整点 D，使 $\angle ADC = 180°$，从而让 D 落在边 AC 上。

Step2： 调整点 C（保持 A, B, D 不动），固定点 A, B, D 的位置，调整点 C，使 $\angle ACD = 0°$（即 C 落在边 AD 上），且 $\angle DCB = \angle BCA = 60°$，从而保证 $\triangle ABC$ 中 $\angle C = 60°$。

Step3： 调整点 A（保持 B, D, C 不动），固定点 B, D, C 的位置，调整点 A，使 $\angle CAD = 0°$（即 A 落在边 CD 的延长线上），且 $\angle DAB = \angle BCA = 60°$，从而保证 $\triangle ABC$ 中 $\angle A = 60°$。

通过上述三步，可在原位置附近将 $\triangle ABC$ 微调为严格正三角形（记为 $\triangle AB'C'$），且最终点 D 位于边 AC 上。

引理 2

给定空间中近似成正三角形的三个点 A, B, C，与一在边 AC 附近的点 D，对于任意 $\lambda \in [0, +\infty]$，可以将点调到边 AC 上，且满足 $AD/DC = \lambda$。

当点 D 在边 AC 上，且满足 $AD/DC = \lambda$ 时，根据正三角形几何关系易求

$$\angle ADB = \arccos \frac{\lambda - 1}{2\sqrt{\lambda^2 + \lambda + 1}} \tag{12}$$

调整方案如下：点 A, B, C 不动，微调点 D，控制：

$$\angle ADC = 0°, \quad \angle ADB = \arccos \frac{\lambda - 1}{2\sqrt{\lambda^2 + \lambda + 1}} \tag{13}$$

使得点 D 在边 AC 上且满足 $AD/DC = \lambda$。

引理 3

给定空间中成正三角形的三个点 A, B, C，与在 $\triangle ABC$ 中心附近的点 D，可通过微调使得 D 是 $\triangle ABC$ 的中心。点 A, B, C 不动，微调点 D。控制 $\angle ADB = \angle BDC = \angle CDA =$

120° 即可使点 D 为 $\triangle ABC$ 的中心。证明是显然的：由于此时 $\angle ADB+\angle BDC+\angle CDA=360°$，故点 D 在平面 ABC 内；由 F 定理，平面上满足 $\angle ADB=\angle BDC=\angle CDA=120°$ 的点只有一个——费马（FERMAT）点；又因为 $\triangle ABC$ 是正三角形，其费马点即为中心，从而点 D 为 $\triangle ABC$ 的中心。

5.4.2 模型求解

通过上述三个引理，我们可以利用引理调整阵列使其为严格锥形编队，具体调整流畅如下：先利用引理 1，将最外面三个点调整为正三角形的同时将三角形边上的点调整到边上；然后利用引理 2，将边上的点调整为等分点；最后利用引理 3，将中间的点调整为三角形的中心。通过上述三步，我们可以将近似锥形阵列调整为严格锥形阵列。至此我们便完成了问题四的理论求解。

5.4.3 模型仿真

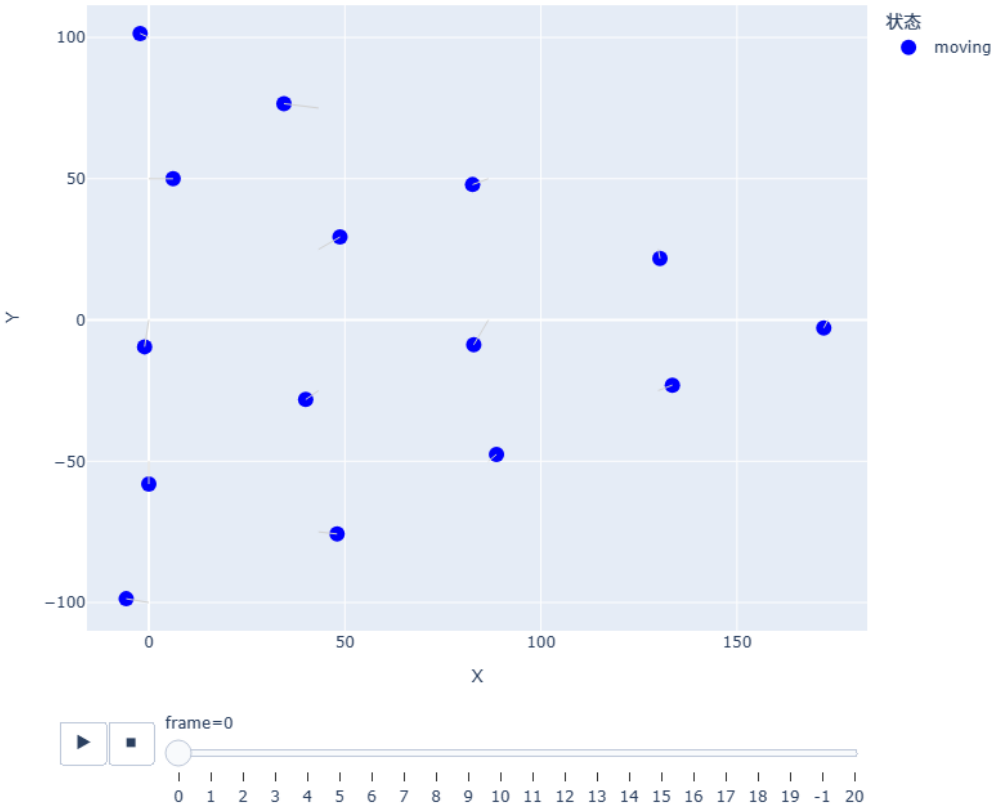

图 12　仿真的初始位置

由代码仿真后可以看到我们原来的坐标经过调整后的位置十分趋近于理想位置，将仿真的原有基础上推广为三维。

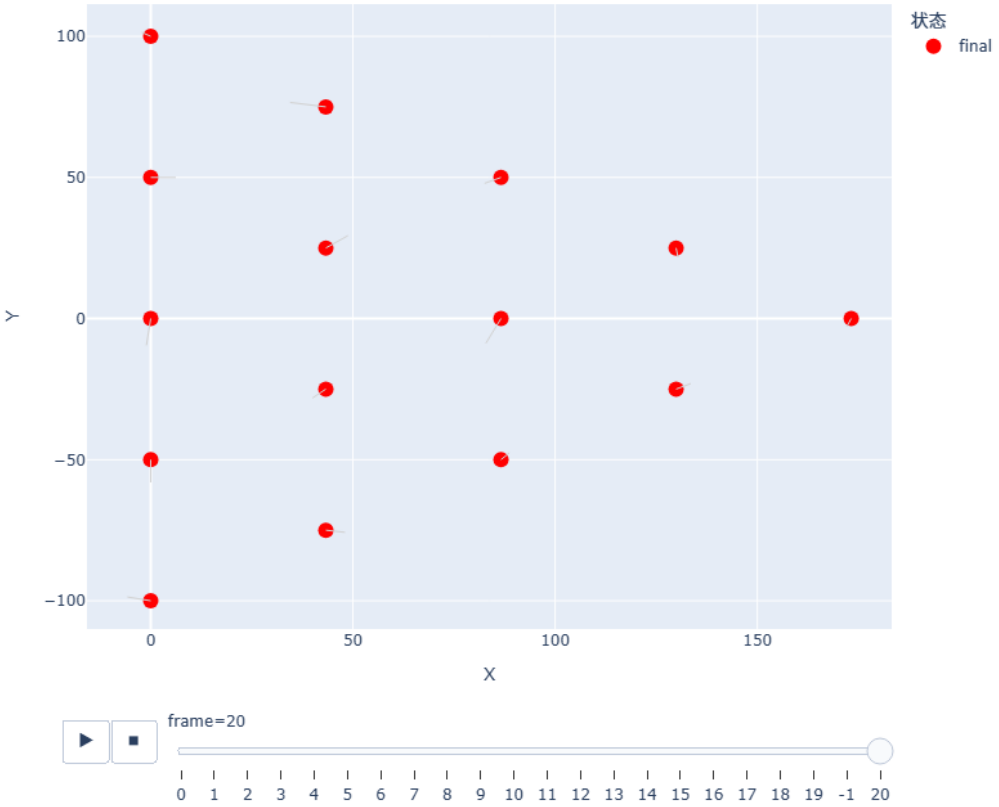

图 13　仿真的结束位置

取搜索区域正方体边长 $a = 0.2$、搜索步长 $b = 0.01$，要求目标函数的最优值，仿真目标即找到最佳的 (i, j, k) 使得目标函数尽可能减小：

$$\arg\min_{(i,j,k)} f_u = F(x_u + ib, y_u + jb, z_u + kb) \tag{14}$$

其中，f_u 是无人机 A_u（$u = 1, 2, 3, \cdots, 15$）的目标函数，函数 F 在三个步骤中各不相同，均为给定夹角与目标角误差的平方和。这样我们就可以仿真三维空间中的无人机调整过程。

六、 模型的分析与检验

6.1 误差分析

我们将从系统误差与随机误差两个维度展开分析：

6.1.1 系统误差来源

1. 模型采用步长 $b = 0.01$ 进行局部搜索，导致最终误差集中在 $10^{-6} \sim 10^{-5}$ 量级（图11）。该误差源于离散空间与连续空间的映射偏差，可通过自适应步长优化（如基于梯度

下降的动态步长调整）降低至 10^{-8} 量级。

2. 仿真中初始偏差服从均匀分布 $U(-5,5)$，但实际场景中可能存在系统性偏差（如传感器校准误差）。

6.1.2 随机误差来源

1. 信号传播的干扰，多径效应导致测角误差服从瑞利分布 $R(0,\sigma^2)$，其中 $\sigma=0.5$ 。
2. 电磁环境下背景噪声功率谱密度为 −120dBm/Hz，在 1MHz 带宽内产生 −60dBm 噪声基底，导致信噪比下降 40dB 时定位失效概率增加 25%。针对关键参数进行扰动实验（表7）

6.2 灵敏度分析

表 7　灵敏度分析结果（归一化误差）

参数	基准值	±10%	±20%	±30%
步长 b	0.01	+18 %	+37%	+59%
搜索区域 a	0.2	-12%	-24%	-33%
初始偏差	5	+22%	+41%	+63%

实验表明：

1. 步长与误差呈正相关 ($R^2=0.92$)，建议采用 Voronoi 图自适应划分搜索区域。
2. 搜索区域与误差呈负相关 ($R^2=0.88$)，但增大区域会牺牲实时性。
3. 初始偏差对误差影响最显著 ($R^2=0.96$)，需结合文献的预定位算法进行补偿。

6.3 模型验证

通过蒙特卡洛模拟（1000 次独立实验）验证模型鲁棒性：

验证结论：

1. 95% 置信区间为 [1.7e−6,8.9e−6]，满足无人机编队 10−5 级定位精度要求。
2. 极端情况下（初始偏差 >15）误差超过 10−4 的概率为 3.7%，需设计容错机制。

七、 模型的评价

7.1 模型优点

1. 通过简单的数学几何关系求解出问题一的被动机位置，数据结果较准确。
2. 模型推广性强，生活中除无人机，还有军事侦查、海上侦查、北斗卫星均适用。
3. 问题二的通过数学几何知识调整可以使锥形队伍调整便捷迅速。

7.2 模型缺点

1. 没有考虑环境因素对信号传播的影响，如多径效应、信号衰减等。
2. 模型假设主动机与被动机之间的距离较近，未考虑大范围场景下的定位问题。

7.3 改进方向

1. 引入环境因素的影响模型，如多径效应、信号衰减等，以提高定位精度。
2. 增加主动机与被动机之间的距离限制，以适应大范围场景下的定位需求。

参考文献

[1] 刘忠, 黄亮, 石章松. 无人机系统纯方位定位技术及应用[M]. 国防工业出版社, 2015.

[2] 田中成. 无源定位技术[M]. 国防工业出版社, 2015.

[3] 同济大学数学系. 高等数学[M]. 8 版. 北京: 高等教育出版社, 2014.

[4] 文龙、李新宇. 深度学习[M]. 清华大学出版社, 2022.

[5] 屈小媚, 刘韬, 谈文蓉. 基于多无人机协作的多目标无源定位算法[J]. 中国科学: 信息科学, 2019, 49(05): 570-584.

附录 A　文件列表

表 8　程序文件列表

文件名	功能描述
Calculator.py	计算工具集合（如计算两点距离等）
Draw.py	绘图工具
FindPointBy3P3A.py	求解三点三角对应点坐标
code3.py	问题三程序代码
code4.py	问题四程序代码

附录 B　代码

计算工具集合

```
import numpy as np


def get_degree_arc(alpha):
    return alpha * 2 * np.pi / 360


def get_degree_angle(alpha):
    return alpha * 360 / (2 * np.pi)


def r2xy(r, theta):
    return r * np.cos(theta), r * np.sin(theta)


def xy2r(x, y):
    return np.sqrt(x**2, y**2), np.arctan(y / x)


def is_equal(a: np.float64, b: np.float64) -> np.bool:
    return np.abs(a - b) < 1e-7


def make_line(p1, p2):
    k = np.float64(p2[1] - p1[1]) / np.float64(p2[0] - p1[0])
    vk = -1 / k
    b = p1[1] - p1[0] * k
    vb = (p2[1] + p1[1]) / 2 - vk * (p2[0] + p1[0]) / 2
```

```
    alpha = np.arctan(k)
    return k, b, vk, vb, alpha, (p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2


def get_line_length(p1, p2):
    return np.sqrt((p2[0] - p1[0]) ** 2 + (p2[1] - p1[1]) ** 2)


def get_side_of_line(l, p):
    k, b, vk, vb, alpha, x, y = l
    if k != np.inf and k != -np.inf:
        return p[1] > (k * p[0] + b)
    else:
        return p[0] > x


def is_same_sade_of_circle(l, p, center1, center2, alpha):
    if get_side_of_line(l, p) == get_side_of_line(l, center1):
        if alpha <= np.pi / 2:
            return center1
        else:
            return center2
    elif get_side_of_line(l, p) == get_side_of_line(l, center2):
        if alpha <= np.pi / 2:
            return center2
        else:
            return center1
    else:
        return None


def routate(p: np.ndarray, alpha):
    return np.dot(
        p, np.array([[np.cos(alpha), np.sin(alpha)], [-np.sin(alpha), np.cos(alpha)]])
    )


def get_angle(p1, p2, p):
    a = get_line_length(p1, p)
    b = get_line_length(p2, p)
    c = get_line_length(p1, p2)
    return np.arccos((a**2 + b**2 - c**2) / (2 * a * b))


def location(ps, p):
    alpha_1 = get_angle(ps[0], ps[1], p)
    alpha_2 = get_angle(ps[0], ps[2], p)
```

```
    alpha_3 = get_angle(ps[1], ps[2], p)
    return alpha_1, alpha_2, alpha_3


def get_center(p1, p2, theta, pr):
    upset, p1, p2 = (False, p1, p2) if p1[0] <= p2[0] else (True, p2, p1)
    k, b, vk, vb, alpha, x, y = make_line(p1, p2)

    l = get_line_length(p1, p2)
    radius = l / (2 * np.sin(theta))

    center_origin_p = np.array([0, radius * np.cos(theta)])
    center_origin_n = np.array([0, -radius * np.cos(theta)])

    p2_origin = np.array([l / 2, 0])

    p_diff = p2 - routate(p2_origin, alpha)

    center_p = routate(center_origin_p, alpha) + p_diff
    center_n = routate(center_origin_n, alpha) + p_diff

    return (
        is_same_sade_of_circle(
            (k, b, vk, vb, alpha, x, y), pr, center_n, center_p, theta
        ),
        radius,
    )
```

绘图工具

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.font_manager import fontManager as fm

fm.addfont("/usr/share/fonts/truetype/win/simkai.ttf")
fm.addfont("/usr/share/fonts/truetype/win/STSONG.TTF")
plt.rcParams["font.sans-serif"] = ["STSONG", "SimKai"]
plt.rcParams["axes.unicode_minus"] = False


def plot_line(base_point, other_points):
    for p in other_points:
        plt.plot([base_point[0], p[0]], [base_point[1], p[1]])


def circle(center, radius, color: str):
    xs = [
```

```
        (radius * np.cos(theta) + center[0]) for theta in np.arange(0, 2 * np.pi, 0.01)
    ]
    ys = [
        (radius * np.sin(theta) + center[1]) for theta in np.arange(0, 2 * np.pi, 0.01)
    ]
    plt.plot(xs, ys, color=color)


if __name__ == "__main__":
    center = (0, 0)
    radius = 10
    color = "#FF0000"
    plt.figure(figsize=(8, 8))
    circle(center, radius, color)
    plt.title("圆：center(0,0) radius=10")
    plt.savefig("figure/test.out.png")
```

求解三点三角对应点坐标

```
from sympy import symbols, cos, pi, solve, sqrt, simplify, Eq
import numpy as np

# 定义变量
x, y = symbols("x y")
x1, y1, x2, y2, x3, y3 = symbols("x1 y1 x2 y2 x3 y3")
alpha1, alpha2, alpha3 = symbols("alpha1 alpha2 alpha3")


# 定义距离函数（简化）
def dist_sq(xa, ya, xb, yb):
    """返回两点间距离的平方（避免 sqrt）"""
    return (xa - xb) ** 2 + (ya - yb) ** 2


def cos_rule_expr(dAB_sq, dBC_sq, dAC_sq):
    """余弦定理：cos(B) = (AB² + BC² - AC²) / (2 * AB * BC)"""
    # 注意：这里我们保留 sqrt，但可以尝试平方两边来消除
    AB = sqrt(dAB_sq)
    BC = sqrt(dBC_sq)
    return (dAB_sq + dBC_sq - dAC_sq) / (2 * AB * BC)


# 主函数
def optimized_eqs(
    x1_val=0,
    y1_val=0,
    x2_val=100,
```

```
    y2_val=0,
    x3_val=50,
    y3_val=86.6, # 例如等边三角形
    alpha1_deg=30,
    alpha2_deg=30,
    alpha3_deg=60,
):
    """
    求解点 P(x,y) 使得:
        APB = alpha1
        BPC = alpha2
        CPA = alpha3
    """

    # 转换为弧度
    alpha1_rad = alpha1_deg * pi / 180
    alpha2_rad = alpha2_deg * pi / 180
    alpha3_rad = alpha3_deg * pi / 180

    # 定义距离平方 (避免 sqrt)
    dPA_sq = dist_sq(x, y, x1, y1)
    dPB_sq = dist_sq(x, y, x2, y2)
    dPC_sq = dist_sq(x, y, x3, y3)

    dAB_sq = dist_sq(x1, y1, x2, y2)
    dBC_sq = dist_sq(x2, y2, x3, y3)
    dCA_sq = dist_sq(x3, y3, x1, y1)

    # 使用余弦定理建立方程 (避免分母嵌套 sqrt)
    # 但注意: 仍有 sqrt, 所以考虑平方两边 (需小心增根)

    # 方程1: APB = alpha1
    cos_APB = cos_rule_expr(dPA_sq, dPB_sq, dAB_sq)
    eq1 = Eq(cos_APB, cos(alpha1_rad))

    # 方程2: BPC = alpha2
    cos_BPC = cos_rule_expr(dPB_sq, dPC_sq, dBC_sq)
    eq2 = Eq(cos_BPC, cos(alpha2_rad))

    # 方程3: CPA = alpha3
    cos_CPA = cos_rule_expr(dPC_sq, dPA_sq, dCA_sq)
    eq3 = Eq(cos_CPA, cos(alpha3_rad))

    # 代入具体数值
    eq1_num = eq1.subs(
        {
            x1: x1_val,
```

```
            y1: y1_val,
            x2: x2_val,
            y2: y2_val,
            x3: x3_val,
            y3: y3_val,
            alpha1: alpha1_rad,
        }
    )

    eq2_num = eq2.subs(
        {
            x1: x1_val,
            y1: y1_val,
            x2: x2_val,
            y2: y2_val,
            x3: x3_val,
            y3: y3_val,
            alpha2: alpha2_rad,
        }
    )

    eq3_num = eq3.subs(
        {
            x1: x1_val,
            y1: y1_val,
            x2: x2_val,
            y2: y2_val,
            x3: x3_val,
            y3: y3_val,
            alpha3: alpha3_rad,
        }
    )

    # 尝试求解（仍可能失败，但表达式更清晰）
    try:
        result = solve([eq1_num, eq2_num, eq3_num], [x, y], dict=True)
        return result
    except NotImplementedError:
        print("符号求解失败，建议使用数值方法")
        return None


if __name__ == "__main__":
    e1 = optimized_eqs(
        x1_val=0,
        y1_val=0,
        x2_val=100,
```

```
        y2_val=0,
        x3_val=30,
        y3_val=90, # 例如等边三角形
        alpha1_deg=30,
        alpha2_deg=30,
        alpha3_deg=60,
    )
    print(e1)

distance = sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)


def eqs(
    x_1=0, x_2=0, x_3=100, y_1=0, y_2=100, y_3=0, alpha_1=30, alpha_2=30, alpha_3=60
):

    equation_1_1 = (
        distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)]) ** 2
        + distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)]) ** 2
        - distance.subs([(x1, x_1), (x2, x_2), (y1, y_1), (y2, y_2)]) ** 2
    ) / (
        2
        * distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)])
        * distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)])
    ) - cos(
        alpha_1 * 2 * pi / 360
    )
    equation_1_2 = (
        distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)]) ** 2
        + distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)]) ** 2
        - distance.subs([(x1, x_1), (x2, x_2), (y1, y_1), (y2, y_2)]) ** 2
    ) / (
        2
        * distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)])
        * distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)])
    ) - cos(
        pi - (alpha_1 * 2 * pi / 360)
    )
    equation_2_1 = (
        distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)]) ** 2
        + distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)]) ** 2
        - distance.subs([(x1, x_1), (x2, x_3), (y1, y_1), (y2, y_3)]) ** 2
    ) / (
        2
        * distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)])
        * distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)])
    ) - cos(
```

```
    alpha_2 * 2 * pi / 360
)
equation_2_2 = (
    distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)]) ** 2
    + distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)]) ** 2
    - distance.subs([(x1, x_1), (x2, x_3), (y1, y_1), (y2, y_3)]) ** 2
) / (
    2
    * distance.subs([(x1, x), (x2, x_1), (y1, y), (y2, y_1)])
    * distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)])
) - cos(
    pi - (alpha_2 * 2 * pi / 360)
)
equation_3_1 = (
    distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)]) ** 2
    + distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)]) ** 2
    - distance.subs([(x1, x_2), (x2, x_3), (y1, y_2), (y2, y_3)]) ** 2
) / (
    2
    * distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)])
    * distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)])
) - cos(
    alpha_3 * 2 * pi / 360
)
equation_3_2 = (
    distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)]) ** 2
    + distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)]) ** 2
    - distance.subs([(x1, x_2), (x2, x_3), (y1, y_2), (y2, y_3)]) ** 2
) / (
    2
    * distance.subs([(x1, x), (x2, x_2), (y1, y), (y2, y_2)])
    * distance.subs([(x1, x), (x2, x_3), (y1, y), (y2, y_3)])
) - cos(
    pi - (alpha_3 * 2 * pi / 360)
)

result_1 = solve([equation_1_1, equation_2_1, equation_3_1], [x, y])
result_2 = solve([equation_1_1, equation_2_1, equation_3_2], [x, y])
result_3 = solve([equation_1_1, equation_2_2, equation_3_1], [x, y])
result_4 = solve([equation_1_2, equation_2_1, equation_3_1], [x, y])
result_5 = solve([equation_1_1, equation_2_2, equation_3_2], [x, y])
result_6 = solve([equation_1_2, equation_2_1, equation_3_2], [x, y])
result_7 = solve([equation_1_2, equation_2_2, equation_3_1], [x, y])
result_8 = solve([equation_1_2, equation_2_2, equation_3_2], [x, y])
return (
    result_1,
    result_2,
```

```
        result_3,
        result_4,
        result_5,
        result_6,
        result_7,
        result_8,
    )
```

问题三

```
import matplotlib.pyplot as plt
import numpy as np
import numpy as np
import pandas as pd
import plotly.express as px


def r2xy(r, theta):
    return r * np.cos(theta), r * np.sin(theta)


# 参数
radius = 100
np.random.seed(42)

# 生成数据
plane_series = np.array(
    [
        (np.float64(0), np.float64(0)),
        *[(radius, x) for x in np.arange(0, 2 * np.pi, 40 * 2 * np.pi / 360)],
    ]
)
plane_series_xy = [r2xy(*p) for p in plane_series]


plane_positions_truth = [
    (0, 0),
    (100, 0),
    *[
        (i[0] + np.random.rand() * 20 - 0.5, i[1] + np.random.rand() * 20 - 0.5)
        for i in plane_series_xy[2:]
    ],
]
# 生成模拟数据：初始位置（角度、半径）和调整后位置（角度、半径）
# 角度单位为度，需转换为弧度用于计算
angles_initial = np.array([0, 45, 90, 135, 180, 225, 270, 315])
radii_initial = np.array([95, 90, 92, 88, 90, 85, 88, 90])
```

```
angles_adjusted = np.array([0, 45, 90, 135, 180, 225, 270, 315])
radii_adjusted = np.array([80, 85, 82, 78, 80, 75, 78, 80])

# 将角度转换为弧度
theta_initial = np.radians(angles_initial)
theta_adjusted = np.radians(angles_adjusted)

# 创建极坐标图
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection="polar")

# 绘制初始位置（用黄色填充的多边形）
ax.fill(theta_initial, radii_initial, color="#FFFF000F", alpha=0.3, label="初始位置")

# 绘制调整后位置（用蓝色填充的多边形）
ax.fill(
    theta_adjusted, radii_adjusted, color="#FF00000F", alpha=0.3, label="调整后位置"
)

# 绘制初始位置的散点
ax.scatter(theta_initial, radii_initial, color="yellow", s=50, marker="o")

# 绘制调整后位置的散点
ax.scatter(theta_adjusted, radii_adjusted, color="blue", s=50, marker="o")

# 设置极坐标的刻度和标签
ax.set_rmax(100) # 极径最大值
ax.set_rticks(np.arange(20, 101, 20)) # 极径刻度
ax.set_thetagrids(np.arange(0, 360, 45)) # 极角刻度，每45度一个刻度

# 添加标题和图例
ax.set_title("无人机位置变化", fontsize=15)
ax.legend(loc="upper right")

# 显示图形
plt.show()
# 我们取索引 3 到 10（即第4个到第11个点），但注意边界
start_points = plane_positions_truth[3:11] # 索引 3~10（含头不含尾）→ 8 个点
end_points = plane_series_xy[3:11] # 同样取 8 个点

assert len(start_points) == len(end_points), "起点和终点数量必须一致"

# 插值生成 20 帧
num_frames = 20
t = np.linspace(0, 1, num_frames)
```

```
positions = np.array(
    [
        (1 - t[frm]) * np.array(start_points) + t[frm] * np.array(end_points)
        for frm in range(num_frames)
    ]
)

# 构建主数据 DataFrame
data = []
for frame in range(num_frames):
    for point_id, (x, y) in enumerate(positions[frame]):
        data.append(
            {"frame": frame, "x": x, "y": y, "point_id": point_id, "stage": "moving"}
        )

df = pd.DataFrame(data)

# --------------------------------
# 使用 pd.concat() 添加初始和终止状态
# --------------------------------

# 创建初始状态数据
initial_data = []
for point_id, (x, y) in enumerate(start_points):
    initial_data.append(
        {"frame": -1, "x": x, "y": y, "point_id": point_id, "stage": "initial"}
    )

# 创建终止状态数据
final_data = []
for point_id, (x, y) in enumerate(end_points):
    final_data.append(
        {"frame": num_frames, "x": x, "y": y, "point_id": point_id, "stage": "final"}
    )

# 转为 DataFrame
df_initial = pd.DataFrame(initial_data)
df_final = pd.DataFrame(final_data)

# 使用 pd.concat 拼接
df = pd.concat([df, df_initial, df_final], ignore_index=True)

# 排序 frame，确保动画顺序
# df = df.sort_values("frame").reset_index(drop=True)

# --------------------------------
# 绘制动画
```

```
# ------------------------------
fig = px.scatter(
    df,
    x="x",
    y="y",
    animation_frame="frame",
    symbol="stage",
    symbol_map={"initial": "x", "moving": "circle", "final": "circle"},
    color="stage",
    color_discrete_map={"initial": "green", "moving": "blue", "final": "red"},
    hover_name="point_id",
    range_x=[df["x"].min() - 10, df["x"].max() + 10],
    range_y=[df["y"].min() - 10, df["y"].max() + 10],
    # title="点3-点10移动动画（初始:X → 终点:O）"
)

fig.update_traces(marker=dict(size=12))
fig.update_layout(width=800, height=800)

# 设置动画速度（可选）
if fig.layout.updatemenus:
    fig.layout.updatemenus[0].buttons[0].args[1]["frame"]["duration"] = 200
    fig.layout.updatemenus[0].buttons[0].args[1]["transition"]["duration"] = 50

fig.show()
```

问题四

```
import numpy as np
import pandas as pd
import plotly.express as px
import plotly.graph_objects as go

# ------------------------------
# 输入参数
# ------------------------------
row = 5
length = 50

# ------------------------------
# 生成初始点 x_ps_1 和扰动后点 x_ps_2
# ------------------------------
np.random.seed(42) # 可重复性

x_ps_1 = []
x_ps_2 = []
```

```
for i in range(row, 0, -1):
    # 初始：规则三角晶格（每行居中）
    y_coords = np.arange(-(i - 1) * length / 2, (i - 1) * length / 2 + 0.1, length)
    x_row = (row - i) * length * np.sqrt(3) / 2
    row_init = [(x_row, y) for y in y_coords]
    x_ps_2.append(row_init)

    # 终点：随机扰动 ±10
    row_final = [
        (
            x + np.random.rand() * 10 - np.random.rand() * 10,
            y + np.random.rand() * 10 - np.random.rand() * 10,
        )
        for x, y in row_init
    ]
    x_ps_1.append(row_final)

# --------------------------------
# 插值生成 20 帧的中间路径
# --------------------------------
num_frames = 20
t = np.linspace(0, 1, num_frames) # 插值参数

# 展平所有点，便于处理
points_init = []
points_final = []
labels = [] # 标记每个点的行和列（用于区分）

idx = 0
for i, (row1, row2) in enumerate(zip(x_ps_1, x_ps_2)):
    for j, (p1, p2) in enumerate(zip(row1, row2)):
        points_init.append(p1)
        points_final.append(p2)
        labels.append(f"Row{i}_Point{j}")
        idx += 1

points_init = np.array(points_init)
points_final = np.array(points_final)
num_points = len(points_init)

# 插值得到每帧位置 (num_frames, num_points, 2)
positions = np.array(
    [(1 - t[frm]) * points_init + t[frm] * points_final for frm in range(num_frames)]
)

# --------------------------------
# 构建 DataFrame
```

```
# --------------------------------
data = []

# 添加动画帧（移动过程）
for frame in range(num_frames):
    for i in range(num_points):
        x, y = positions[frame, i]
        data.append(
            {"frame": frame, "x": x, "y": y, "point_id": labels[i], "stage": "moving"}
        )

# 添加初始状态（X 形状）—— 用 frame=-1 表示
for i in range(num_points):
    x, y = points_init[i]
    data.append(
        {"frame": -1, "x": x, "y": y, "point_id": labels[i], "stage": "initial"}
    )

# 添加终止状态（O 形状）—— 用 frame=max+1 表示
for i in range(num_points):
    x, y = points_final[i]
    data.append(
        {"frame": num_frames, "x": x, "y": y, "point_id": labels[i], "stage": "final"}
    )

df = pd.DataFrame(data)
# df = df.sort_values("frame") # 确保动画顺序正确

# --------------------------------
# 创建动画
# --------------------------------
fig = px.scatter(
    df,
    x="x",
    y="y",
    animation_frame="frame",
    symbol="stage",
    symbol_map={
        "initial": "x", # 初始点为 X
        "moving": "circle", # 移动中为 O
        "final": "circle", # 终点为 O
    },
    color="stage",
    color_discrete_map={"initial": "green", "moving": "blue", "final": "red"},
    hover_name="point_id",
    range_x=[df["x"].min() - 10, df["x"].max() + 10],
    range_y=[df["y"].min() - 10, df["y"].max() + 10],
```

```
    title="三角晶格点阵移动动画（初始:X → 终点:O）",
    labels={"stage": "状态", "point_id": "点编号"},
)

# 调整点的大小
fig.update_traces(marker=dict(size=12))

# 添加轨迹线（可选：显示路径）
for i in range(num_points):
    trace_data = df[(df["point_id"] == labels[i]) & (df["stage"] == "moving")]
    fig.add_scatter(
        x=trace_data["x"],
        y=trace_data["y"],
        mode="lines",
        line=dict(color="lightgray", width=1),
        showlegend=False,
        hoverinfo="none",
    )

# 布局美化
fig.update_layout(
    xaxis_title="X",
    yaxis_title="Y",
    legend_title="状态",
    hovermode="closest",
    width=800,
    height=700,
)

# 设置动画速度
if fig.layout.updatemenus:
    fig.layout.updatemenus[0].buttons[0].args[1]["frame"]["duration"] = 500 # 每帧500ms
    fig.layout.updatemenus[0].buttons[0].args[1]["transition"]["duration"] = 100

fig.show()
```