# 基于长短期记忆网络 (LSTM) 的蔬菜补货与定价决策模型

## 摘要

本文

**针对问题一，**你好

**针对问题二，**

**针对问题三，**

**针对问题四，**

**关键词**：’xx’ ’xx’ ’xx’ ’xx’ ’xx’

# 一、 问题重述

## 1.1 问题背景

为保持编队队形，拟采用纯方位无源定位的方法调整无人机的位置，即由编队中某几架无人机发射信号、其余无人机被动接收信号，从中提取出方向信息进行定位，来调整无人机的位置。编队中每架无人机均有固定编号，且在编队中与其他无人机的相对位置关系保持不变。无人机集群在遂行编队飞行时，为避免外界干扰，应尽可能保持电磁静默，少向外发射电磁波信号。

## 1.2 问题提出

接收信号的无人机所接收到的方向信息约定为：该无人机与任意两架发射信号无人机连线之间的夹角（如图 1 所示）。例如：编号为 FY01、FY02 及 FY03 的无人机发射信号，编号为 FY04 的无人机接收到的方向信息是 $\alpha_1$，$\alpha_2$ 和 $\alpha 3$。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，形成圆形编队，其中 9 架无人机（编号 FY01 FY09）均匀分布在某一圆周上，另 1 架无人机（编号 FY00）位于圆心（见图 2）。无人机基于自身感知的高度信息，均保持在同一个高度上飞行。

1. 位于圆心的无人机（FY00）和编队中另 2 架无人机发射信号，其余位置略有偏差的无人机被动接收信号。当发射信号的无人机位置无偏差且编号已知时，建立被动接收信号无人机的定位模型。
2. 某位置略有偏差的无人机接收到编号为 FY00 和 FY01 的无人机发射的信号，另接收到编队中若干编号未知的无人机发射的信号。若发射信号的无人机位置无偏差，除 FY00 和 FY01 外，还需要几架无人机发射信号，才能实现无人机的有效定位?
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，请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（每次调整的时间忽略不计），使得 9 架无人机最终均匀分布在某个圆周上。利用表 1 给出的数据，仅根据接收到的方向信息来调整无人机的位置，请给出具体的调整方案。

**问题二：**实际飞行中，无人机集群也可以是其他编队队形，例如锥形编队队形（见图 3，直线上相邻两架无人机的间距相等，如 50 m）。仍考虑纯方位无源定位的情形，设计无人机位置调整方案。

# 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　模型核心符号说明**

| 符号 | 说明 | 单位 |
|---|---|---|
| $g$ | 品类标识 | - |
| $n_g$ | 第 $g$ 类品类的样本量 | - |

# 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 $(R,\theta)$，其中 $\theta$ 已知，设接收信号的被动机 $j$ 极坐标为 $(r,\varphi)$，其中 $(r)$ 与 $(\varphi)$ 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 $\theta$ 与 $\varphi$ 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 $\theta$ 与 $\varphi$ 的取值范围不影响数值解的大小，仅影响解的正负，故仅从 $\theta$ 与 $\varphi$ 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。

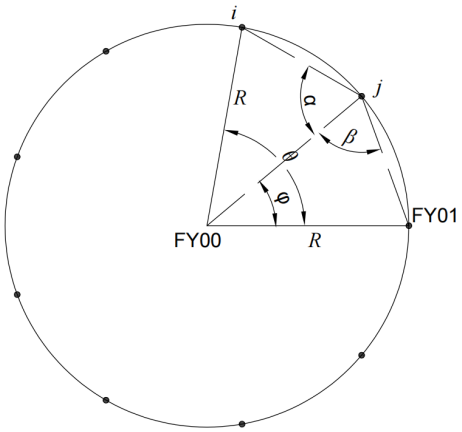


**图 1　主动机与被动机排布的情况 1**

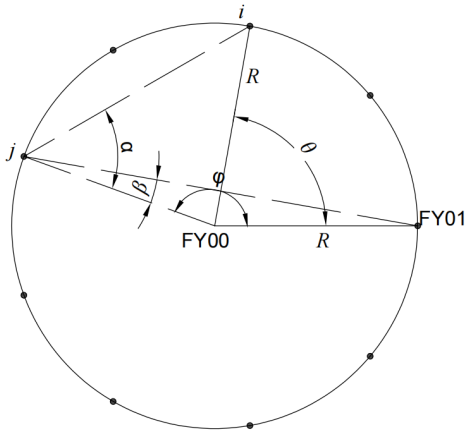


**图 2　主动机与被动机排布的情况 2**

**注：主动机发射的方向信息 $\alpha$ 为 $(i,0)$ 的夹角，$\beta$ 为 $(0,1)$ 的夹角。**

### 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

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

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 $(r_1, \varphi_1)$ 和 $(r_2, \varphi_2)$。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）。

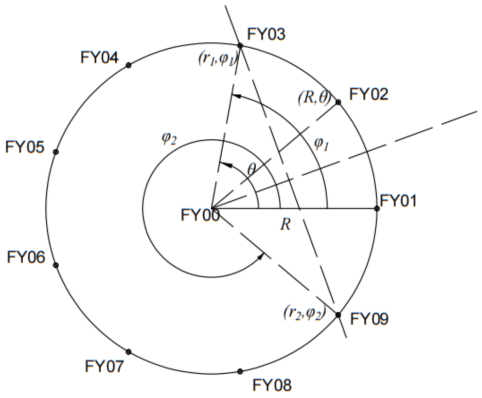


**图 3　两组解的对称分布示意图**

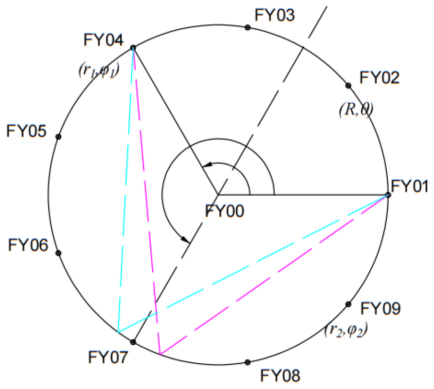


**图 4　解的重合情况示意图**

由于无法直接确定 $\alpha$、$\beta$ 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，$\alpha$ 与 $\beta$ 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 $\alpha$ 和 $\beta$ 对应的主动机编号，因此可能位于图中的 M、N 两点。

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 $\alpha$ 和 $\beta$ 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 $\theta$，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 $\beta$，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。

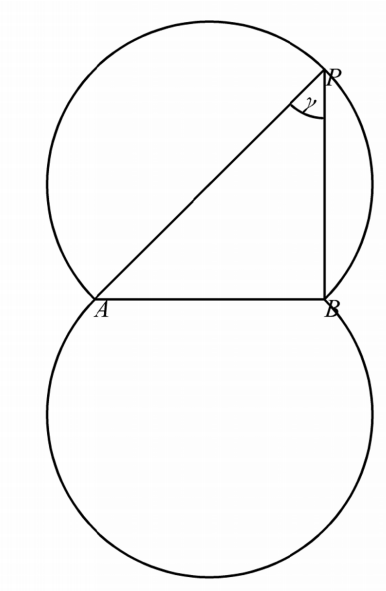


**图 5　主动机与被动机排布的情况 1**

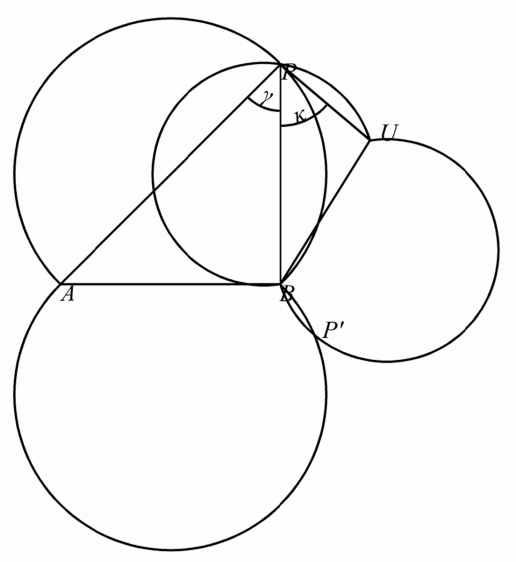


**图 6　主动机与被动机排布的情况 2**

### 5.2.2 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。

## 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

### 5.3.2 模型建立

### 5.3.3 问题求解

### 5.3.4 求解结果

### 5.4 问题四的模型建立与求解

### 5.4.1 模型建立

### 5.4.2 问题求解

### 5.4.3 求解结果

## 六、 模型的分析与检验

### 6.1 误差分析

### 6.2 灵敏度分析

## 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.

# 附录 A　运行结果

# 附录 B　文件列表

**表 2　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 | 问题四程序代码 |

# 附录 C　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
```

```python
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()
```