针对烟幕干扰弹最优投放策略的研究

摘要

随着现代防空作战中无人机的广泛使用，利用无人机投放烟幕干扰弹以实现最长遮蔽效果变得尤为重要。本文针对无人机投放烟幕干扰弹抵御一枚至多枚来袭空地导弹的场景，基于运动学分析构建多维度投放策略优化模型，为不同作战情境下的干扰决策提供科学依据。

对于问题一，我们首先构建**三维笛卡尔坐标系**，明确无人机 FY1 与导弹 M1 的初始位置与运动特性，通过运动学方程推导无人机飞行轨迹、烟幕干扰弹平抛运动轨迹及起爆点坐标，选取真目标上的 **31 个关键点**，构建**有效遮蔽判定条件**（即导弹-真目标视线与烟幕球相交）。采用步长 0.01s 的**遍历法**求解有效遮蔽时长。结果显示，FY1 以给定参数飞行与投弹，实现的有效遮蔽时长仅为 **1.48s**，单一烟幕弹遮蔽效果有限。

对于问题二，本文以**有效遮蔽时长最大化**为目标，构建单目标优化模型，将无人机航向角、飞行速度、烟幕弹投放时刻、起爆延迟时间设为决策变量，同步考虑烟幕弹起爆高度非负、有效时间窗口不晚于导弹命中假目标等 **4 项约束条**件。采用**粒子群优化算法**（PSO）对最大有效遮蔽时长 Δt_s 求解。结果表明，在最优参数组合（飞行速度 **116.61m/s**、航向角 **7.14°**、投放时刻 **0.1195s**、起爆延迟 **0.6603s**）下，有效遮蔽时长提升至 **5.05s**，较问题一提升 241%。

对于问题三，针对无人机 FY1 投放 3 枚烟幕弹的场景，我们将扩展决策变量至 8 维，新增两枚弹投放间隔 ≥1s 等约束，构建多弹协同遮蔽优化模型。引入**自适应惯性权重**与**局部搜索策略**对粒子群算法改进。求解得联合有效遮蔽时长 **6.2s**，填入 result1.xlsx。其中前两枚弹形成 [1.2002,4.6002]s 与 [4.2000,7.4000]s 的重叠遮蔽区间，第三枚弹因导弹飞行轨迹变化未形成有效遮蔽。

对于问题四，针对三架无人机协同投弹的场景，我们构建 12 维决策变量优化模型。提出**重叠分段筛选**和 **Top-K** 评估措施，将每架无人机速度区间划分为 **3 个重叠分段**（[70,92]、[88,112]、[108,140]m/s），组合生成 27 种速度方案并筛选有效遮蔽时长排名前 2 个优域，结合改进自适应变异粒子群算法继续对优域进行搜索求解最大遮蔽时长。结果显示，三机协同的有效遮蔽时长达 **11.14s**，遮蔽效果较单无人机多弹投放显著提升，相关参数已保存至 result2.xlsx。

对于问题五，面对 5 架无人机投弹干扰 M1、M2、M3 三枚导弹的复杂场景，我们建立以三枚导弹的**总有效遮蔽时长**最大化为目标，以投弹时间间隔、投弹数量上限等为约束条件的混合变量优化模型。由于决策变量含连续参数与离散参数，我们采用**自适应双坐标系差分进化算法**（$ABSDE_mv$），通过**特征坐标变换**降低变量耦合性，结合变异、交叉与选择操作实现全局寻优。搜索出满足约束条件的总有效遮蔽时长为 **19.62s**，有效平衡各导弹的遮蔽需求，相关投放策略填入 result3.xlsx。

关键字： 烟幕干扰弹　粒子群优化算法　协同优化　自适应双坐标系差分进化算法

一、 问题重述

随着无人机技术的不断发展，军用无人机被广泛应用于作战指挥、侦察打击行动。使用无人机投放烟幕干扰弹可以形成烟幕遮蔽，实现对来袭武器的有效干扰，掩护目标。本题模拟无人机投放烟雾干扰弹，针对不同问题下导弹数量、烟幕弹数量和无人机数量的差异安排，分别形成了问题一、二、三、四和五。要求我们对无人机飞行方位和速度、投放时间点、烟幕干扰弹起爆点等多个投放策略进行确定，以实现最佳遮蔽效果。

二、 模型假设

1. **忽略空气阻力：**假设在导弹、无人机和烟幕整个运行过程中，忽略风力和空气阻力影响。
2. **忽略导弹自身重力：**假设导弹飞行过程中，速度方向直指假目标，导弹以匀速直线运动靠近假目标。
3. **无人机飞行环境无干扰：**假设无人机在接受任务后，瞬间完成航向调整，飞行过程中无障碍物碰撞影响，以匀速等高度直线运动飞行。
4. **烟幕弹投放与烟幕形成瞬时完成：**假设无人机投放烟幕干扰弹动作瞬时完成，烟幕干扰弹起爆后瞬时形成球状烟幕云团。

三、 符号说明

表 **1**　符号说明

符号	意义	单位
$P_{FYi,0}$	第 i 个无人机的初始位置	-
v_{FYi}	第 i 个无人机的飞行速度	m/s
t_{start}	烟幕云团遮蔽开始时刻	s
t_{end}	烟幕云团遮蔽结束时刻	s
$M_i(t)$	第 i 枚导弹在时刻 t 的位置	
t_{dropi}	第 i 枚干扰弹投放时刻	s
Δt_{bomi}	第 i 枚干扰弹起爆时间	s
θ_i	第 i 个无人机航向角	度

四、 问题一模型的建立与求解

4.1 问题分析与建模思路

问题一给定了无人机的飞行速度和方位，飞行时间和烟幕干扰弹爆炸时间，需要我们建立三维坐标系下的无人机和导弹飞行轨迹，分析无人机飞行、烟幕干扰弹下落和烟幕云团形成的三个运动过程，并定义烟幕云团有效遮蔽时间，建立有效遮蔽的判定模型，最终实现对烟幕遮蔽云团有效遮蔽时间统计。

针对问题一，我们采用运动学方程与几何约束方程结合的方式，分别对烟幕干扰弹投放、起爆及云团运动轨迹进行表示，在三维笛卡尔坐标系下分别表示无人机投放点、烟幕弹起爆点、任意时刻导弹位置及烟幕云团中心位置。对于位置的求解，首先根据无人机固定飞行参数，求解烟幕弹投放点与起爆点位置；接着通过建立时间与空间的关联关系，根据烟幕弹起爆点位置及云团运动特性，推出任意时刻烟幕云团中心位置，同时结合导弹飞行参数推出任意时刻导弹位置。对于有效遮蔽时长的求解，通过导弹视线与烟幕云团的位置判定关系，根据任意时刻烟幕云团、导弹及真目标的位置，筛选出有效遮蔽时刻并计算总时长。模型建立的流程图如图1所示。

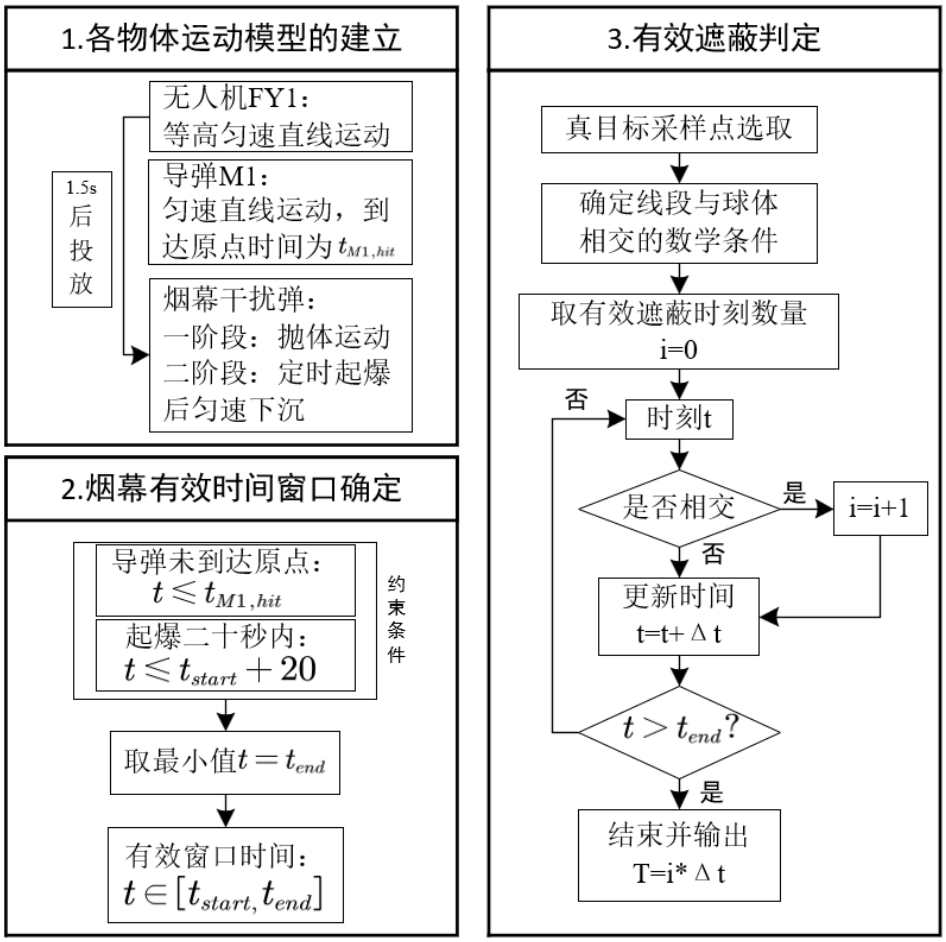

图 **1**　问题一流程图

4.2 导弹 M1 和无人机 FY1 的运动轨迹

4.2.1 FY1 的运动轨迹

首先以假目标为原点 $O(0,0,0)$，水平面为 xy 平面，竖直方向为 z 轴构建三维笛卡尔坐标系，真目标下地面圆心坐标为 $T_0(0,200,0)$，$r=7$ m, 高度 $h=10$ m。FY1 从初始位置 $P_{FY1,0}(17800,0,1800)$ 向原点飞行，飞行方向向量可以构建为 $\vec{d}_{\text{FY1}}=(-17800,0,0)$。因为 FY1 匀速等高飞行，其速度向量可以表示为 $\vec{v}_{\text{FY1}}=(-v_{\text{FY1}},0,0)=(-120,0,0)$。FY1 在受领任务到投放烟幕弹经过 1.5s，定义为延迟投放时间 t_1，烟幕干扰弹间隔 3.6 s 后起爆，定义为延迟起爆时间 t_2。FY1 在 t_1 时间内做匀速直线运动，其运动轨迹函数可以表示为 (1)

$$P_{\text{FY1}}(t)=P_{\text{FY1,0}}+\vec{v}_{\text{FY1}}\cdot t \tag{1}$$

当 $t=t_1$ 时投下烟幕干扰弹，此时 FY1 的坐标可以表示为 P_{drop} =**(17620,0,1800)m**。

4.2.2 M1 的运动轨迹

M1 从初始位置 $P_{\text{M1,0}}(20000,0,2000)$ 以 300m/s 朝假目标原点匀速直线飞行，其飞行方向向量可定义为 $\vec{d}_{\text{M1}}=\overrightarrow{P_{M10}O}=(-20000,0,-2000)$。归一化后的单位方向向量可表示为 (2)

$$\vec{e}_{\text{M1}}=\frac{\vec{d}_{\text{M1}}}{\left|\vec{d}_{\text{M1}}\right|} \tag{2}$$

带入 M1 初始坐标位置 $P_{\text{M1,0}}$，计算得到单位方向向量 $\vec{e}_{\text{M1}}=(-0.9950,0,-0.0995)$

在警戒雷达发现导弹后，在任意时刻 t 下的 M1 位置为 $M1(t)=P_{\text{M1,0}}+v_{\text{M1}}\cdot\vec{e}_{\text{M1}}\cdot t$。M1 的最长运行时间，即击打到假目标的时间 $t_{\text{M1,hit}}=\frac{|P_{\text{M1,0}}|}{v_{\text{M1}}}$，计算 M1 初始位置模长 $|P_{\text{M1,0}}|\approx 20099.7506$ m，可以得出到达原点的时间 $t_{\text{M1, hit}}$ ≈**66.9992 s**。

4.3 求解烟幕干扰弹起爆点 P_{burst}

烟幕干扰弹脱离无人机后，初速度与 FY1 投放时刻的速度 $|\vec{v}_{\text{FY1}}|$ 相同，在重力作用下作平抛运动。一下是对烟幕干扰弹在脱离 FY1 后的运动分析：

- 水平方向上烟幕弹保持匀速直线运动，初速度 $\vec{v}_{\text{bomb, x}}=-120\text{m/s}$, 在经历 $t_2=3.6s$ 后，水平方向沿 x 轴位移 $\Delta x_{\text{bomb}}=-432$ m。
- 竖直方向上做自由落体运动，初速度为 0，加速度 $a_z=-g$(以向上为正方向，g 取 $9.8m^2/s$)。经历 $t_2=3.6s$ 后，在 z 轴方向上的位移计算公式由自由落体公式 $s=v_0t+\frac{1}{2}at^2$ 可得 $\Delta z_{\text{bomb}}=-63.504$ m。

根据平抛运动的位移计算结果,可以得到烟幕干扰弹起爆点坐标 P_{burst} =**(17188,0,1736.496)**。

4.4 烟雾云团有效运行状态求解

烟幕云团出现时刻为 $t_{start} = t_1 + t_2 = 5.1s$。在 $t \geq 5.1$ s 时烟幕云团沿竖直方向匀速下降，烟幕云团中心位置函数可以定义为

$$S(t) = \begin{cases} \text{无定义} & t < 5.1 \text{ s}, \\ (17188, 0, 1736.496 - 3(t - 5.1)) & t \geq 5.1 \text{ s}. \end{cases} \tag{3}$$

由题目可知，有效的烟幕云团存在时间为起爆后 20s 内，且导弹未到达真目标，由此可得到烟幕云团的有效时间为为 $t_{\text{M1, hit}} \approx 66.9993s$。烟幕云团的有效时间窗口为 $t \in [t_{\text{start}}, t_{\text{end}}] =$**[5.1s, 25.1s]**。

4.5 有效遮蔽条件

要实现有效遮蔽，就是以导弹视角观察，真目标被烟幕云团所形成的阴影掩盖，转换成数学语言表达形式为真目标上的任意一点 T_i 与导弹位置 $M1(t)$ 的连线 $M1(t)T_i$ 与烟雾云团相交。

线段 $M1(t)T_i$ 的参数方程如下：其中 $\lambda \in [0, 1], \lambda = 0$ 对应于导弹, $\lambda = 1$ 对应于真目标上一点

$$\vec{r}(\lambda) = M1(t) + \lambda \cdot (T_i - M1(t)). \tag{4}$$

烟幕云团方程可表示为球心在 $S(t)$, 半径为 10 m 的球，$\|\vec{r}(\lambda) - S(t)\|^2 = 100$

为更好覆盖真目标，我们选择了真目标圆柱体上的 31 个具有代表性的关键采样点，作为有效遮蔽的条件的计算。如图2所示，我们在中心轴上每隔 1m 选取了 11 个采样点，在上下底面分别每隔 45° 等角度采样选取一点，具体包含了 8 个边界点和 2 个内部采样点，共计 11 个关键采样点。问题转化为在 $[5.1 \text{ s}, 25.1 \text{ s}]$ 的有效时间窗口内，真目标上的 31 个关键采样点 T_{ki}(k 代表关键采样点) 与 $M1(t)$ 的连线 $M1(t)T_{ki}$ 是否完全与烟雾云团相交。

4.6 有效遮蔽时长求解

由烟雾云团方程和线段参数方程联立，将参数方程代入球体方程，整理得到关于 λ 的一元二次方程为：

$$\|\vec{B} - \vec{A}\|^2\lambda^2 - 2 \cdot (\vec{A} - \vec{C}) \cdot (\vec{B} - \vec{A})\lambda + \|\vec{A} - \vec{C}\|^2 - 100 = 0 \tag{5}$$

其中 $\vec{A} = M1(t), \vec{B} = T_{ki}, \vec{C} = S(t)$。要求该方程有实根且至少一个根在 $[0, 1]$ 内。

即要求一元二次方程判别式非负：$\Delta = b^2 - 4ac \geq 0$，其中 $a = \|\vec{B} - \vec{A}\|^2$，$b = 2 \cdot (\vec{A} - \vec{C}) \cdot (\vec{B} - \vec{A})$，$c = \|\vec{A} - \vec{C}\|^2 - 100$。有效根范围：$\lambda_i \in [0, 1]$。在每一时刻下导弹和烟幕云团的位置都会发生改变，因此需要统计 $t \in [t_{\text{start}}, t_{\text{end}}] = [5.1 \text{ s}, 25.1 \text{ s}]$ 的时间窗

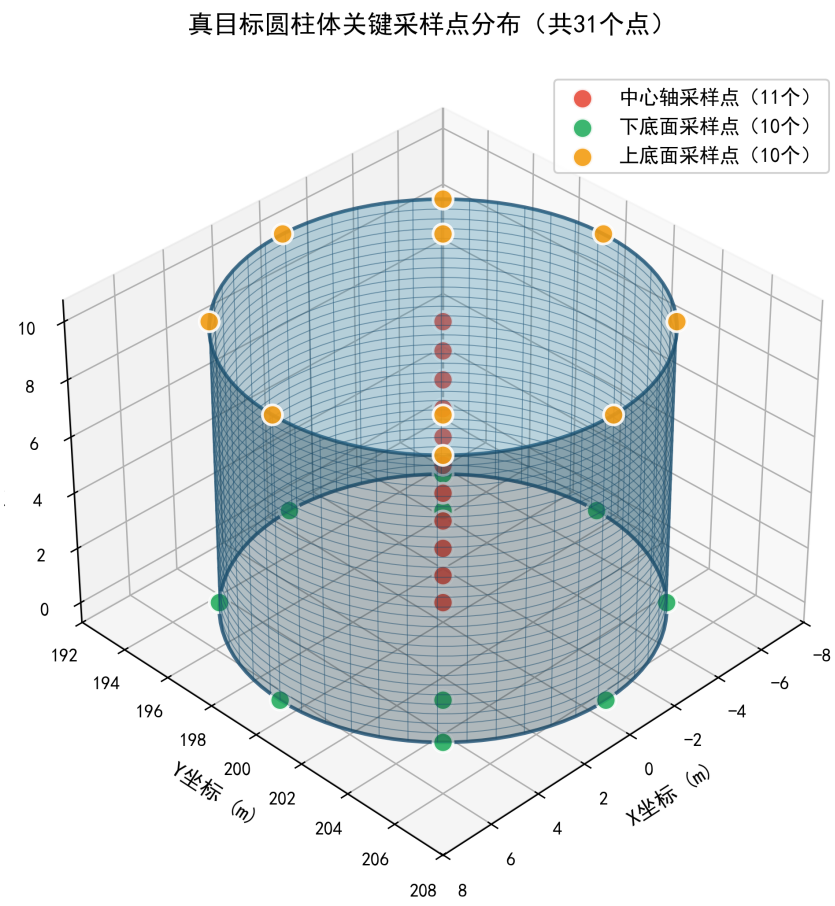

图 **2**　采样点分布图

口内的每一时刻下的方程成立情况。将统计到的方程成立时刻累加得到最终的有效遮蔽时长。

4.6.1 遍历法求解

在本小问我们选择遍历的方法求解有效遮蔽时长 Δt_s，步骤如下：将 t=5.1s 时的 $M1(t)$ 和 $S(t)$ 计算得出，设定时间步长 $step = 0.01s$，判断导弹与圆柱体每个关键点连线是否与烟雾云团相交。进行多次循环，t 每次增加 0.01s。在不满足方程或 $t > t_end$ 时停止步长增加，此时的 Δt 足够接近实际值，结束求解循环。以伪代码的形式呈现我们的求解过程如下：

```
while t_cureent <= t_end:
   t_probe = t_start + step
//(Calculate_M 和 Calculate_S 是计算 M1(t) 和 S(t) 的具体函数)
   M_probe = Calculate_M1(t_probe)
   S_probe = Calculate_S(t_probe)

if Is_Lambda_Equation_Valid(M_probe, S_probe) is true:
   t_current = t_probe
```

```
else: break
delta_t = t_current - t_start
print delta_t
```

4.7 得出有效遮蔽时间

结合以上求解步骤，编写 Python 程序 (附录 A)，可以得出在步长为 0.01s 的有效遮蔽时刻数量为 148 个，由于无人机与导弹运动均为一个连续的过程，因此有效遮蔽时刻在时间窗口内是一个连续分布，故总有效遮蔽时间等于有效遮蔽时刻 × 步长，即有效遮蔽时长为 $\Delta t_s = 1.48s$。开始有效遮蔽时间为 7.97s，截止有效遮蔽时间为 9.44s。由求解结果可知，单个烟幕干扰弹对于目标的遮蔽效果极差，难以形成有效遮蔽窗口期，需要多个烟幕干扰弹的配合使用，才能有效延长遮蔽时间。图3展示了时间轴下的的烟幕有效值遮蔽时长。

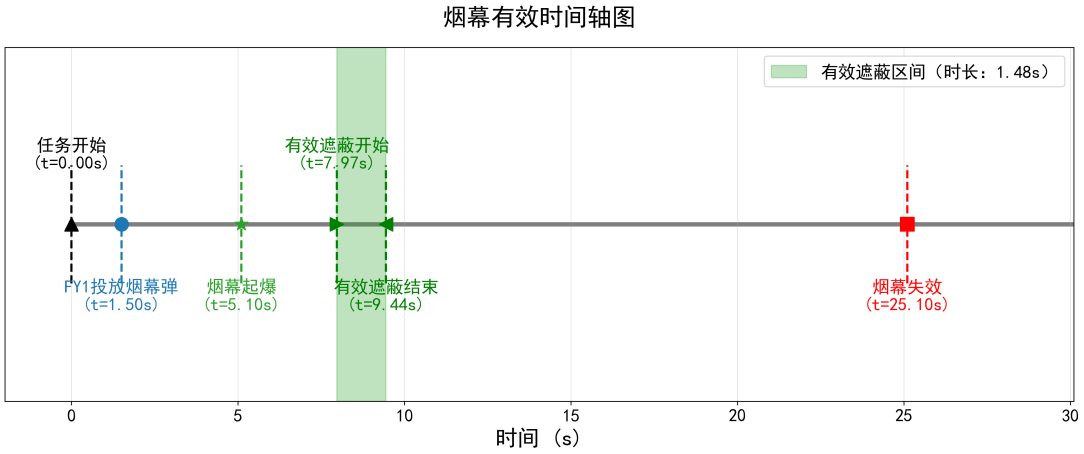

图 **3**　烟幕有效遮蔽时长图

图4展示了导弹向假目标飞行过程中，从导弹视角观察真目标上的关键点形成的导弹视线，定义视线到烟幕中心点的距离。导弹来袭初期，视线-烟幕中心距离变小，当进入有效阈值 10m 时 (即导弹的视线都会穿过烟幕云团形成的球体)，实现有效遮蔽。导弹飞行后期，视线-烟幕中心距离迅速拉大，烟幕云团遮挡失效。

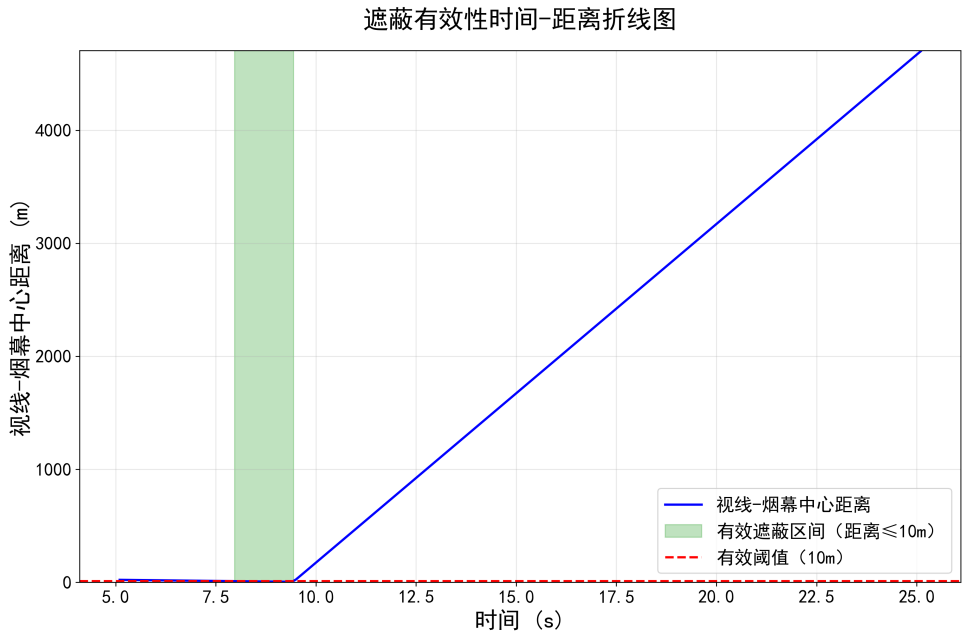

图 **4**　遮蔽时间与距离关系图

五、 问题二模型的建立与求解

5.1 问题分析与建模思路

前文对无人机投放烟幕弹干扰来袭导弹问题进行运动过程拆解与基础分析，对于无人机 FY1 投放单枚烟幕弹以实现遮蔽时间最大化这一情形，我们需要综合考量无人机飞行参数、烟幕弹投放与运动参数等对遮蔽效果的影响，进而构建多约束下的单目标优化模型。

在无人机 FY1 投放单枚烟幕弹干扰导弹 M1 的场景中，每一组无人机的航向角 θ、飞行速度 v_{FY} 在确定后无法改变，烟幕弹的投放时刻 t_{drop} 、投放至起爆的时间间隔 Δt_{bom} 影响最终烟幕云团的产生位置和运行轨迹，进而会对真目标的有效遮蔽时长产生影响。对于给定的这四个初始条件，我们可以通过一系列运动学分析，得到不同参数组合下烟幕云团对真目标的有效遮蔽时长，概括为 $\Delta t_s = f\left(\theta, v_{FY}, t_{\text{drop}}, \Delta t_{\text{bom}}\right)$。

实际问题分析中，由于无人机的飞行速度存在上下限约束，且投放时刻与起爆时间间隔需保证烟幕云团有效时间窗口处于导弹飞行过程中，使得问题成为在特定限制条件下的最优解搜索问题。我们的目标是找到一组无人机飞行参数与烟幕弹投放参数，使得烟幕云团对真目标的有效遮蔽时长最长，同时满足各项约束条件，这便转化为了以有效遮蔽时长最大为目标的单目标非线性优化问题。以下是我们针对问题二的建模思路。

问题二是烟幕弹遮蔽时长最大化优化问题。基于无人机飞行参数与烟幕弹运动特性的关联关系，建立以烟幕弹对 M1 的有效遮蔽时长最大为目标，以无人机速度范围、烟幕弹起爆高度下限、烟幕弹有效时间窗口为约束条件的单目标优化模型，并利用粒子群优化算法进行求解，搜索出满足约束条件的最优无人机飞行方向、飞行速度、烟幕弹投

放点及起爆点。流程图如图5所示。

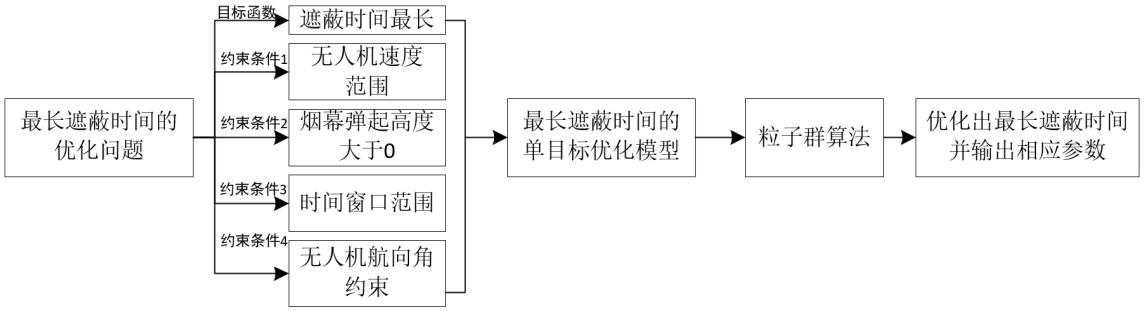

图 **5**　问题二分析流程图

5.2 最大烟幕遮蔽时间单目标优化模型的建立

问题一中已经分析得出影响烟幕遮蔽时间的四个参数，由此可以建立最大烟幕遮蔽时间的单目标优化模型，构建出相关决策变量、目标函数和约束条件

5.2.1 决策变量和目标函数

决策变量包括两方面, 共计四项参数组合，记作 $X_2 = (\theta, v_{FY}, t_{\text{drop}}, \Delta t_{\text{bom}})$：

- 无人机**飞行参数**：无人机航向角 θ 和飞行速度 v_{FY}，在确定后无法改变，无人机保持匀速直线等高飞行。
- 烟幕弹**投放参数**：无人机在受领任务后的投放时刻 t_{drop} 和投放烟幕弹至起爆的时间间隔 $\Delta t_{bom} \in [0, 19.17s]$。

目标函数为最大有效遮蔽时长 Δt_s,Δt_s 的计算遵循问题一中使用遍历法进行求解，每一时刻下的方程 (5) 是否满足，统计得到在该参数下的有效遮蔽时长。

$$\max \Delta t_s \tag{6}$$

5.2.2 约束条件

1. **无人机速度约束**：飞行速度需在 $70 \leq v_{FY1} \leq 140\ \text{m/s}$ 。
2. **烟幕弹起爆高度大于 0**：烟幕弹脱离无人机后仅受重力作用，需要满足烟幕弹在自由落体至地面前发生爆炸，即 $1800 - \frac{1}{2} \times g \times \Delta t_{\text{bom}}^2 \geq 0$，计算得到 Δt_{det}
3. **投放时刻约束**：烟幕云团维持 20s 有效结束的时刻，不能晚于导弹抵达假目标：即 $t_{\text{drop}} + \Delta t_{\text{bom}} + 20 \leq 66.999\ \text{s}$。
4. **无人机航向角约束**：无人机航向角范围需要在 $[0, 2\pi)$ 范围内。

综上可以形成本题的约束条件公式如下：

$$\text{s.t.}\begin{cases} 70 \leq v_{FY1} \leq 140\ \mathrm{m/s} \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{\text{bom}}^2 \geq 0 \\ t_{\text{drop}} + \Delta t_{\text{bom}} + 20 \leq 66.999\ \mathrm{s} \\ \theta \in [0, 2\pi) \end{cases} \tag{7}$$

5.2.3 最大烟幕遮蔽时间单目标优化模型的给出

根据以上构建的决策变量、目标函数和约束条件，我们可以构建出以**最大遮蔽时间**的**单目标优化模型**：

$$\max \Delta t_s$$
$$\text{s.t.}\begin{cases} 70 \leq v_{FY1} \leq 140\ \mathrm{m/s} \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{\text{bom}}^2 \geq 0 \\ t_{\text{drop}} + \Delta t_{\text{bom}} + 20 \leq 66.999\ \mathrm{s} \\ \theta \in [0, 2\pi) \end{cases} \tag{8}$$

5.3 最大烟幕遮蔽时间单目标优化模型的求解——粒子群算法

在众多启发式算法中，粒子群算法作为一种构造简单、且收敛速度快的算法 [1]。可以实现快速搜索效率和记忆能力的平衡，广泛使用于各种优化问题的求解。其核心思想是在有限的解空间中创建 N 个粒子，每个粒子单独搜索最优解并将最优解向整个构建的粒子群共享，在不断迭代过程中实现优化的目的 [2]。下面给出具体的粒子群算法求解步骤：

Step1 初始化粒子群：

首先，设定初始粒子数目为 50，最大的迭代次数为 100。定义第 i 个粒子在第 k 代的位置向量为 5.2.1 给出的四项参数组合 X_2，将不同量纲的参数归一化，避免产生搜索偏向性。

$$\boldsymbol{X}_i^{(k)} = \left(v_i^{(k)}, \theta_i^{(k)}, t_{\text{drop},i}^{(k)}, \Delta t_{\text{bom},i}^{(k)}\right)^T \quad (9)$$

每个粒子的移动方向和步长为

$$\boldsymbol{V}_i^{(k)} = \left(v_{v,i}^{(k)}, v_{\theta,i}^{(k)}, v_{t_{\text{drop}},i}^{(k)}, v_{\Delta t_{\text{bom}},i}^{(k)}\right)^T \quad (10)$$

惯性权重初始值 $\omega = 0.9$，学习因子 $c_1 = c_2 = 2$，在四项参数组合的优化变量范围内随机生成粒子群。

Step2 计算粒子适应度函数值

计算每个粒子在第 k 代位置向量下的有效遮蔽时间，为粒子的适应度 $T\left(\boldsymbol{X}_i^{(k)}\right)$，不满足约束条件的粒子适应度设为 0。

Step3 计算个体与全局最优解：

先将每个粒子的适应度值与其历史最优适应度值 $T\left(\boldsymbol{P}_i^{(k)}\right)$ 相比较，若 $T\left(\boldsymbol{X}_i^{(k)}\right) > T\left(\boldsymbol{P}_i^{(k)}\right)$，则将更新该粒子的个体最优位置为当前位置。再将所有粒子的 $T\left(\boldsymbol{P}_i^{(k)}\right)$ 与全局最优适应度值相比较，取最大的个体最优适应度作为全局最优适应度，用公式可表达为 $\boldsymbol{G}^{(k+1)} = \arg\max_i T\left(\boldsymbol{P}_i^{(k+1)}\right)$。

Step3 结束条件：

判断迭代次数是否大于 100，若小于 100，则返回 step2 重复计算，若达到最大迭代次数，则输出此时的全局最优位置对应的参数组合以及最大遮蔽时长 Δt_s, 结束算法。

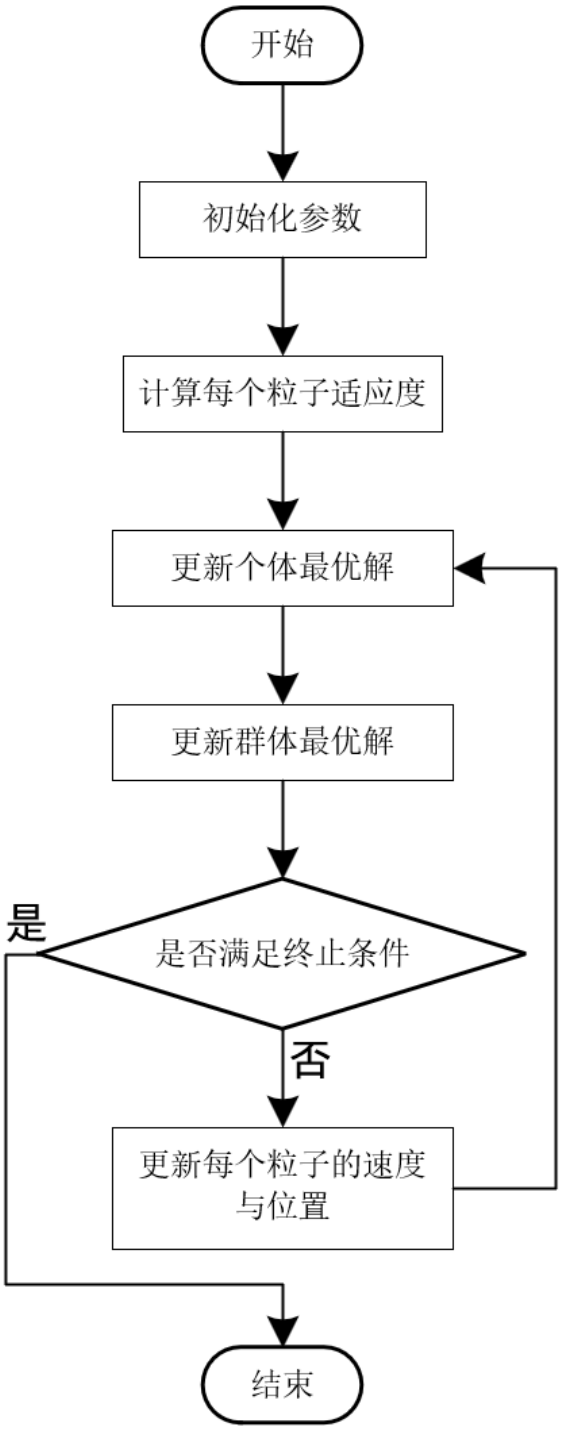

图 **6**　算法流程图

通过编写 Python 程序，编写粒子群算法实现全局寻优迭代，得到了参数组合的最优解集，见附录 B，以下是计算结果的呈现与分析。

5.4 结果展示与分析

如图6, 在 200 次的迭代次数下，粒子更新的有效遮蔽时间初期快速上升，在迭代 50 次后逐渐趋向平稳，在 100 次迭代后粒子逐渐搜索到了全局最优的参数组合下的最大遮蔽时间 $\Delta t_s = 5.1s$。

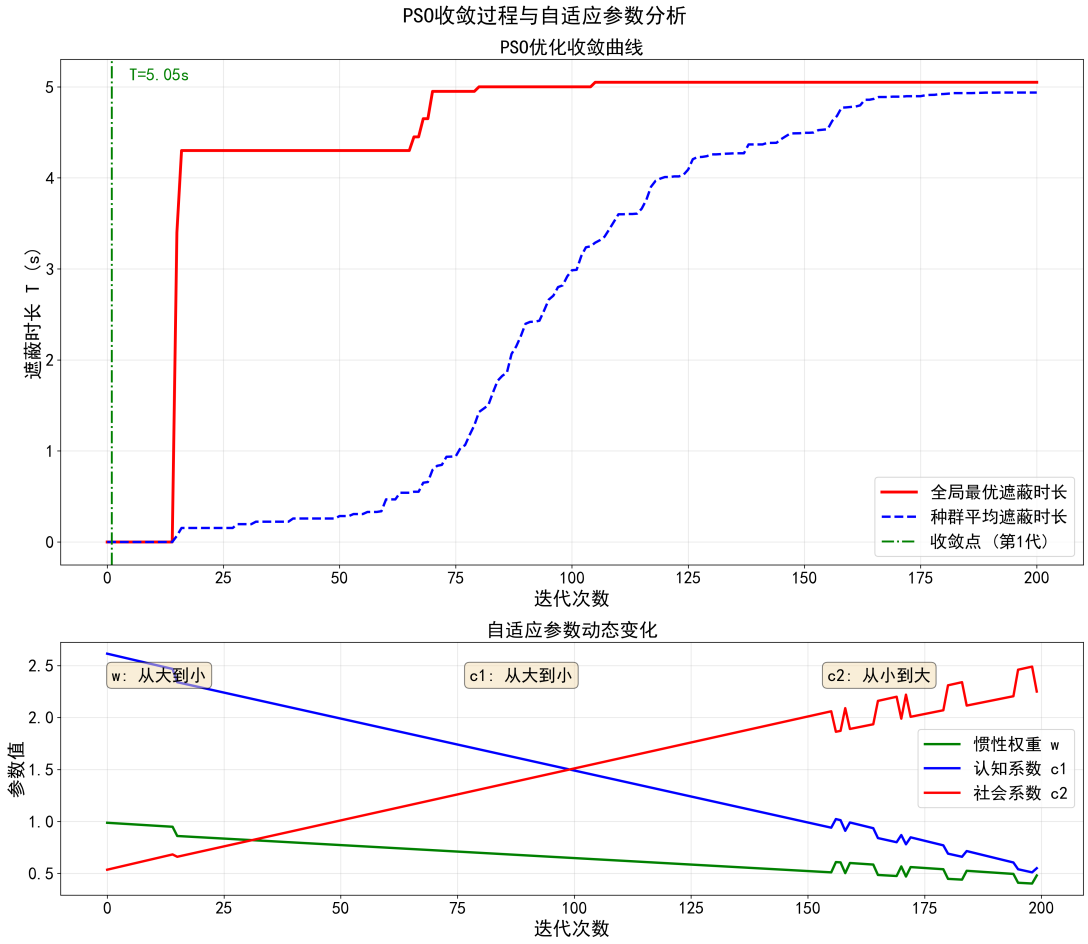

图 6　PSO 迭代收敛曲线图

图6还展示 200 次迭代下的惯性权重 w、认知系数 c1 和社会系数 c2 在迭代过程中的变化情况。惯性权重 w 代表了粒子对前一代学习速度的继承情况，其在每次迭代过程中会逐渐下降，降低搜索速度以找到更加精确的最优结果。认知系数 c1 和社会系数 c2 分别表示粒子从个体最优值和全局最优值上的学习比例，c1 会逐渐变小，c2 会逐渐增大。这是因为迭代初期粒子优先趋向于回顾自己已找到的最优位置，增强个体的探索能力，迭代后期粒子会逐渐向全局最优位置靠近，增加群体的协作能力。图7展示的是部分粒子的运动轨迹图，以及搜索到的最优粒子轨迹和最优解，由于解空间庞大，仅展示了部分全局最优轨迹图。右边展示的是在迭代过程中种群多样性的变化情况，由于粒子在不断迭代中会逐渐靠近全局最优粒子，发生集聚现象，种群的多样性降低。

得到的无人机飞行参数和烟幕弹投放参数见表2。由最大遮蔽时间 $\Delta t_s = 5.05s$ 可

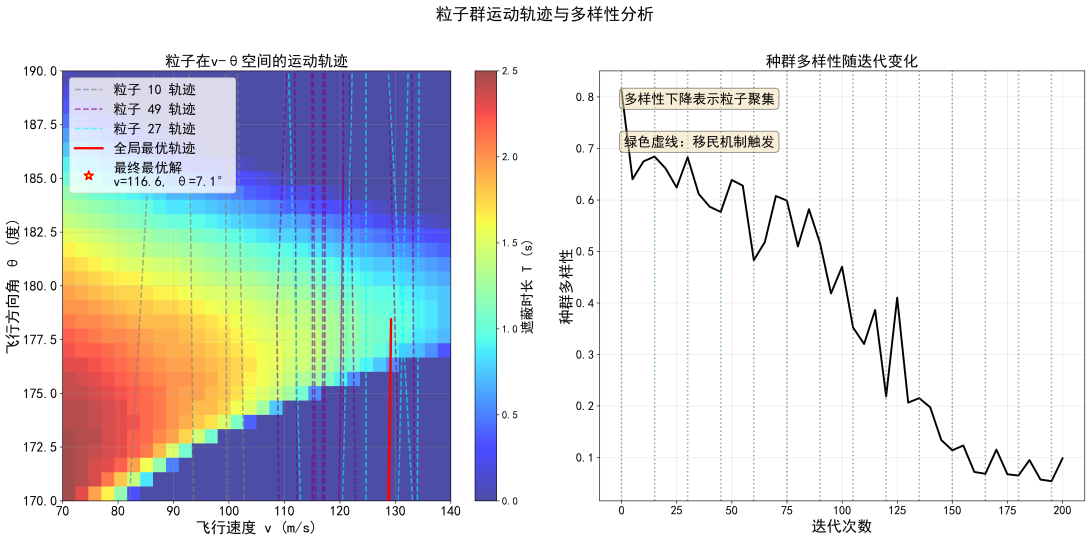

图 **7**　粒子运动轨迹图

得，相较于问题一的给定参数形成的遮蔽时间，采用粒子群算法优化后的最佳参数组合下的遮蔽时间增加了 241%。无人机在 t=0.1195s 时投放烟幕干扰弹，延迟 0.6603s 后烟幕干扰弹发生爆炸。在 t=0.93s 时开始实现有效遮蔽，经历 5.05s 的有效遮蔽时间。

表 **2**　最佳参数表

v_{FY}	θ	t_{drop}	Δt_{bom}	起爆点坐标	Δt_s
$116.61m/s$	$7.14°$	$0.1195s$	$0.6603s$	(17890.22,11.3,1797.8)m	$5.05s$

六、 问题三模型的建立与求解

6.1 问题分析与建模思路

基于前两问对无人机投放单枚烟幕弹干扰导弹问题的分析与求解，对于无人机 FY1 投放三枚烟幕弹干扰导弹 M1 的场景，每一组无人机的飞行参数（航向角、飞行速度）以及三枚烟幕弹各自的投放参数（投放时刻、投放至起爆的时间间隔），都会对应不同的烟幕弹起爆后形成的云团运动轨迹，进而影响对真目标的有效遮蔽时长。由问题二的有效遮蔽时长 5.1s 可得仅一枚烟幕干扰弹形成的干扰时间仍然不足以形成实质性的干扰空窗期，需要增加烟幕干扰弹的数量。通过运动学与几何分析，可以得到不同参数组合下三枚烟幕云团对于真目标的有效遮蔽时长，记为（11）

$$\Delta t_s = f\left(\theta, v_{FY}, t_{drop1}, \Delta t_{bom1}, t_{drop2}, \Delta t_{bom2}, t_{drop3}, \Delta t_{bom3}\right) \tag{11}$$

其中 $t_{\text{drop i}}$ 为第 i 枚烟幕弹投放时刻，$\Delta t_{\text{bom i}}$ 为第 i 枚烟幕弹投放至起爆的时间间隔。

实际情况中，由于任意两枚烟幕弹的投放间隔至少为 1s。且需要保证三枚烟幕云团的有效时间窗口均处于导弹飞行过程中，这些约束条件使得问题三成为在特定限制条件下的最优解规划问题，且核心是同一平台下多个烟幕干扰弹的协同问题。我们需要找到一组无人机飞行参数与三枚烟幕弹的投放参数，使得三枚烟幕云团形成的有效遮蔽时间最长，由此可以构建出决策变量与目标函数，形成以总有效遮蔽时长最大为目标的单目标非线性优化问题。

综上所述，我们希望找到一组可行的无人机飞行与三枚烟幕弹投放参数组合，由于该优化问题中目标及约束条件涉及复杂的运动学与几何关系且维度较高，多为复杂的非线性关系，难以用一般的线性规划方法解决，我们决定采用改进粒子群优化算法来求解。普通粒子群算法在求解高维决策变量的求解过程中常常陷入局部最优解 [3]，采用自适应惯性权重的改进粒子群算法，在每轮迭代过程中根据粒子的运动状态动态调整各个粒子的惯性权重，可有效减少粒子群体的无效迭代次数，避免陷入局部最优 [4]。同时针对 PSO 搜索精细度不好的问题，采取局部优化的方法，确保最终解精度

6.2 多烟幕弹投放遮蔽时间优化模型的建立

问题二中已经分析得出影响烟幕遮蔽时间的四个参数，本题将烟幕干扰弹的数量增加到了三个，由此可以建立最大烟幕遮蔽时间的单目标优化模型，构建出相关决策变量、目标函数和约束条件。

6.2.1 决策变量和目标函数

决策变量包括两方面, 共计 8 项参数组合，记作 (12)

$$X_3 = (\theta, v_{FY}, t_{\text{drop 1}}, \Delta t_{\text{bom 1}}, t_{\text{drop 2}}, \Delta t_{\text{bom 2}}, t_{\text{drop 3}}, \Delta t_{\text{bom 3}}) \tag{12}$$

- 无人机**飞行参数**：无人机航向角 θ 和飞行速度 v_{FY}，在确定后无法改变，无人机保持匀速直线等高飞行。
- 烟幕弹**投放参数**：无人机在受领任务后第 i 枚烟幕弹投放时刻 t_{dropi} ，第 i 枚烟幕弹投放至起爆的时间间隔 $\Delta t_{\text{bom i}}$。

目标函数为最大有效遮蔽时长 Δt_s,Δt_s 的计算遵循问题一中使用遍历法进行求解每一时刻下的方程 (5) 是否满足，统计得到在该参数下的有效遮蔽时长。每个烟幕云团形成的有效遮蔽时间段不会重复，可将三个时间段累加进行目标函数。

$$max \sum_{i}^{3} \Delta t_{si} \tag{13}$$

6.2.2 约束条件

1. **无人机速度约束**：飞行速度需在 $70 \leq v_{FY} \leq 140$ m/s 范围内。

2. **烟幕弹起爆高度大于 0**：烟幕弹脱离无人机后仅受重力作用，需要满足烟幕弹在自由落体至地面前发生爆炸，即 $1800-\frac{1}{2}\times 9.8\times \Delta t_{bomi}^2\geq 0$，计算得到 $\Delta t_{\text{bomi}}\leq 19.16s$。
3. **投放时刻约束**：烟幕云团有效维持时长为 20s，需要确保其结束时刻不能晚于导弹到达假目标时刻：即 $t_{\text{drop i}}+\Delta t_{\text{bom i}}+20\leq 66.999\ \text{s}$
4. **投放间隔约束**：为确保投放过程的合理性与安全性，规定每架无人机投放两枚烟幕干扰弹至少间隔 1 秒。对于任意两枚烟幕弹的投放时刻 t_{drop_i} 和 $t_{drop_j}(i\neq j)$，需要满足 $\left|t_{\text{drop }_i}-t_{\text{drop }_j}\right|\geq 1$。
5. **无人机航向角约束**：无人机航向角范围需要在 $[0,2\pi)$ 范围内。

综上可以形成本题的约束条件公式如下：

$$\text{s.t.}\begin{cases}70\leq v_{FY}\leq 140\ \text{m/s}\\ 1800-\frac{1}{2}\times 9.8\times \Delta t_{\text{bom i}}^2\geq 0\\ t_{\text{drop i}}+\Delta t_{\text{bom i}}+20\leq 66.999\ \text{s}\\ \theta\in[0,2\pi)\\ \left|t_{\text{drop }_i}-t_{\text{drop }_j}\right|\geq 1\end{cases}\tag{14}$$

其中 $i,j\in[1,3]$ 且 $i\neq j$。

6.2.3 最大烟幕遮蔽时间单目标优化模型的给出

根据以上构建的决策变量、目标函数和约束条件，我们可以构建出以**最大遮蔽时间**的**单目标优化模型**：

$$max\sum_{i}^{3}\Delta t_{si}$$

$$\text{s.t.}\begin{cases}70\leq v_{FY}\leq 140\ \text{m/s}\\ 1800-\frac{1}{2}\times 9.8\times \Delta t_{\text{bom i}}^2\geq 0\\ t_{\text{drop i}}+\Delta t_{\text{bom i}}+20\leq 66.999\ \text{s}\\ \theta\in[0,2\pi)\\ \left|t_{\text{drop }_i}-t_{\text{drop }_j}\right|\geq 1\end{cases}\tag{15}$$

6.3 多烟幕弹投放遮蔽时间优化问题的求解——改进粒子群算法

5.3 节已经介绍了利用粒子群算法求解单目标优化模型的步骤，每个粒子迭代速度可以表示为

$$\boldsymbol{V}_i^{(k+1)}=\omega_i^t\cdot\boldsymbol{V}_i^{(k)}+c_1r_1\cdot\left(\boldsymbol{P}_i^{(k)}-\boldsymbol{X}_i^{(k)}\right)+c_2r_2\cdot\left(\boldsymbol{G}^{(k)}-\boldsymbol{X}_i^{(k)}\right)\tag{16}$$

其中 ω_i^t 为粒子i 在第t 次迭代中的取值，$P_i^{(k)}$ 表示第 i 个粒子的历史最优适应度，$G^{(k)}$ 为全局适应度。$\left(P_i^{(k)}-X_i^{(k)}\right)$ 为自我学习量，$\left(G^{(k)}-X_i^{(k)}\right)$ 为社会学习量 [4]。随着迭

代次数的增加，一般情况下 ω 会线性减小 [5], 在寻优过程中会逐渐降低寻找速度。但在求解高维复杂问题时会出现大量粒子群体的无效迭代，严重影响 PSO 算法的收敛能力。在后期的局部搜索中常由于种群多样性过低容易陷入局部最优，无效迭代次数增加。针对以上问题，我们对现有的粒子群算法做出如下改进：

1. **自适应惯性权重**：添加每次迭代过程中粒子的适应度值监控机制。对于适应度变差的粒子惯性权重变为 0，适应度变好的粒子惯性权重继续延续线性递减。若在之后的迭代过程中适应度值发生改进，则惯性权重继续延续线性递减。
2. **局部增强策略**：采取多种策略，粒子进行自适应变异处理，减少局部收敛风险；采用禁忌搜索方法，优化局部搜索精度和分布式探索，减少收敛到局部最优的可能；采用种群重启的办法，强制跳出局部最优。
   - **自适应变异处理**：当种群多样性 CD<0.015 时，触发变异操作，对每个粒子添加正态分布扰动，并修正参数范围，种群多样性公式如下：$\boldsymbol{X}_i^{(k)}$ 表示第 i 个粒子对的欧式距离，$\overline{\boldsymbol{X}^{(k)}}$ 为第 k 代所有粒子对的欧式距离均值，即计算粒子间距离的标准差。

$$D = \sqrt{\frac{1}{NP-1}\sum_{k=1}^{NP}\left(\boldsymbol{X}_i^{(k)} - \overline{X^{(k)}}\right)^2} \tag{17}$$

   添加的变异操作可以用如下公式表示 $X_{i,j}^{new} = X_{i,j}^{old} + N\left(0, \sigma'\right)$，$x_{i,j}$ 表示第 i 个粒子的第 j 个维度。改进后的变异操作集成在代码的adaptive mutation()函数下。
   - **禁忌搜索**：通过建立禁忌列表，避免粒子重复访问适应度较差的区域，即减少对有效遮蔽时间较短的参数空间区域访问。对适应度排名前 20% 的精英粒子优化局部搜索。生成 5 个相邻域解，公式如下。$x_{i,j,k}^{\text{neigh}}$ 为第 i 个精英粒子的第 j 维解空间下的第 k 个领域解，其中 $\delta_{j,k}$ 为随机变量，有 20% 概率调整该维度，取移动的步长 Δ_j 分别为，v_{FY1} ±2 m/s, θ 为 ±0.1rad，其余时间参数 $t_{\text{drop i}}$ 和 $\Delta t_{\text{bom i}}$ 为 ±0.1 s

$$x_{i,j,k}^{\text{neigh}} = x_{i,j}^{\text{old}} + \delta_{j,k} \times \Delta_j \tag{18}$$

   比较设定禁忌列表中 x_{t} 与生成的领域解 x_{neigh} 的欧式距离，小于 $\epsilon = 0.01$ 时判定为禁忌解，不允许进行迭代。编写的禁忌搜索操作集成在代码的 tabu search enhancement() 函数下
   - **种群重启**：保留种群中 20% 的精英粒子直接作为新种群的起始部分，对于搜索到较差有效遮蔽时长的粒子予以重启，重新初始化 50% 的粒子，选取每一维度下的上下界，满足约束条件。公式如下：其中 L_j, U_j 为维度 j 的下界和上界

$$x_{k,j}^{\text{init}} = L_j + \text{rand}(0,1) \times (U_j - L_j) \tag{19}$$

   在代码编写的 pso optimize q3 enhanced () 函数下直接实现了种群重启操作

```
Algorithm: 改进粒子群算法求解最大烟幕遮蔽时长 (EPSO)
Input:    pop_size     ;max_iter// 种群规模和最大迭代次数
参数范围 [xmin, xmax]
增强策略参数 {mutation_rate, chaos_prob, crossover_prob,
  local_search_prob, multi_swarm_prob,
  restart_ratio, elite_ratio, temperature_decay, ...}
Output:   g_best;   g_best_val // 全局最优解和全局最优适应度值

初始化:
在 [0,1]^D 中随机生成粒子位置 Xi
初始化速度 Vi
计算适应度 f(Xi), 设置个体最优 pbest_i
选出全局最优 g_best
初始化 no_improvement_count = 0, temperature = 1.0

迭代过程（t = 1, …, max_iter）:   自适应更新惯性权重 w

fod i in pop_size:
  选择邻域最优 lbest 或全局最优 g_best 作为信息源
  更新速度 Vi ← wVi + c1 * rand() * (pbest_i - Xi) + c2 * rand() * (best - Xi)
  更新位置 Xi ← Xi + Vi, 并修正边界与约束
  计算适应度 f(Xi), 更新 pbest 与 g_best
  计算种群多样性 diversity
  更新无改进计数 no_improvement_count

do 增强策略:
  if diversity < 阈值:   执行自适应变异（高斯/自适应扰动）
  随机概率 chaos_prob:
  执行混沌变异
  随机概率 local_search_prob:   对精英个体执行局部搜索（禁忌搜索增强）
  随机概率 crossover_prob:   对精英个体执行交叉，替换劣质个体
  随机概率 multi_swarm_prob:   执行多子群优化
  if temperature 下降到低值 and 连续无改进:  执行模拟退火接受机制
  if 连续无改进 and diversity 过低:    执行种群重启（保留精英，其余随机重置）
  记录优化历史 (t, g_best_val, mean_pbest_val)

if no_improvement_count 超过阈值:   提前终止
  迭代结束:
    在 g_best 附近执行局部搜索优化
    output g_best 与 g_best_val
```

使用伪代码的形式对改进后的粒子群算法进行阐述如上 [9]：我们通过针对问题二中编写的 Python 代码进行改进，编写程序实现对每次迭代下的粒子适应度状态进行有效监控，在每次迭代过程中都趋向最大烟幕遮蔽时长，在 pso optimize enhanced() 函数中集成了整个问题求解的基本步骤，得到最大烟幕遮蔽时间 Δt_s，编写的 Python 程序见

附录 C。图8展示的是 200 次迭代过程中的收敛曲线与种群多样性变化。种群多样性变化曲线体现了改进后的自适应变异操作和种群重启操作。在近 50 次的迭代后，粒子收敛到了最佳联合遮蔽时长。

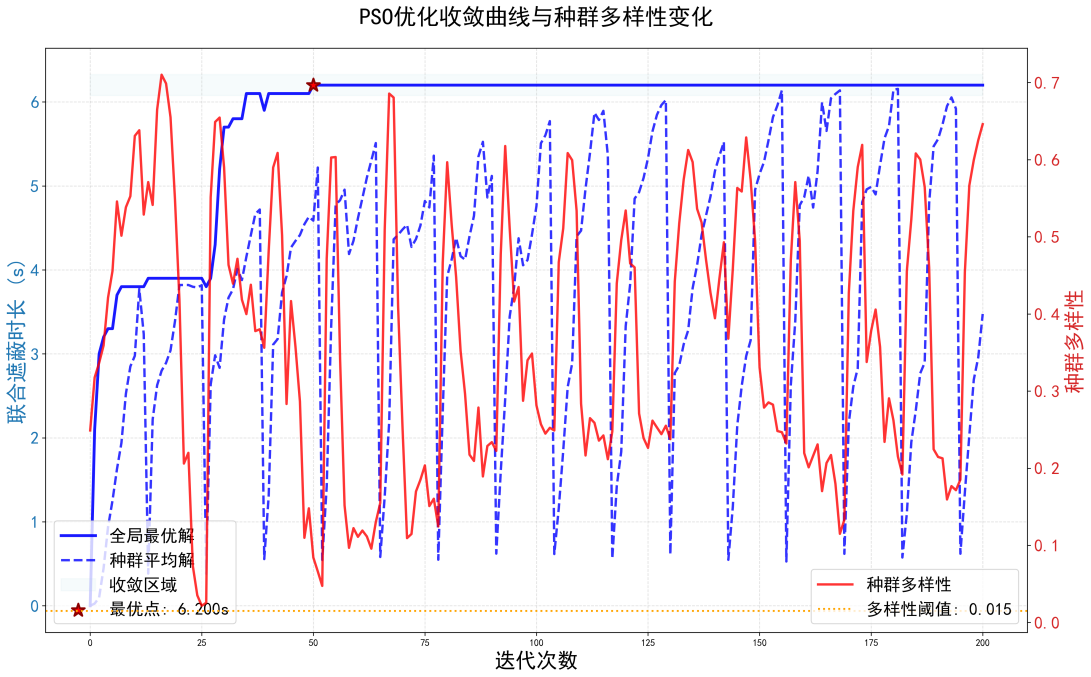

图 **8**　**PSO** 运行情况图

6.4 结果展示与分析

根据以上建模过程，我们可以求得无人机 FY1 投放 3 枚烟幕干扰弹的具体参数组合策略如表3，以及实现的有效遮蔽时间 Δt_s，并将结果保存到文件 result1.xlsx 中。得到的联合有效遮蔽时长为 6.2s

表 **3**　**result1**

θ	v_{FY1} (m/s)	编号	投放点 (x,y,z)m	起爆点 (x,y,z)m	Δt_s
7.07°	120.35m/s	1	(17811.94,1.48,1800)	(17823.95,2.97,1799.95)	3.3s
		2	(17931.37,16.29,1800)	(17943.32,17.77,1799.5)	4.6s
		3	(18059.7,32.2,1800)	(18175.11,46.51,1795.42)	0s

同时还得到了每个烟雾弹的投掷与爆炸参数，以及每个烟幕云团的有效遮蔽时段，见表4。形成的两个有效遮蔽时段有重叠，选取的是 [1.2002s,7.4002s] 的有效遮蔽时间段，即有效遮蔽时间为 6.2s。

表 **4**　**result1** 补充

编号	$t_{\text{drop i}}$	$\Delta t_{\text{bom i}}$	起爆时刻	有效遮蔽时段
1	0.1s	0.1002s	0.2002s	[1.2002, 4.6002]s
2	1.1s	0.1s	1.2s	[4.2000, 7.4000]s
3	2.17s	0.97s	3.14s	无有效遮蔽区间

第三枚烟幕干扰弹无论在何时投下，都无法形成有效烟幕干扰，原因是导弹飞行速度过快，初期无人机的飞行轨迹可以覆盖导弹视线，在此投下的烟幕干扰弹可以形成有效值遮蔽时长，但当导弹不断接近假目标时，无人机无论在何时投下烟幕干扰弹，都无法覆盖导弹与真目标的视线，形成有效遮蔽区间。3D 模拟烟幕干扰情况可以由图9直观呈现，便于理解。

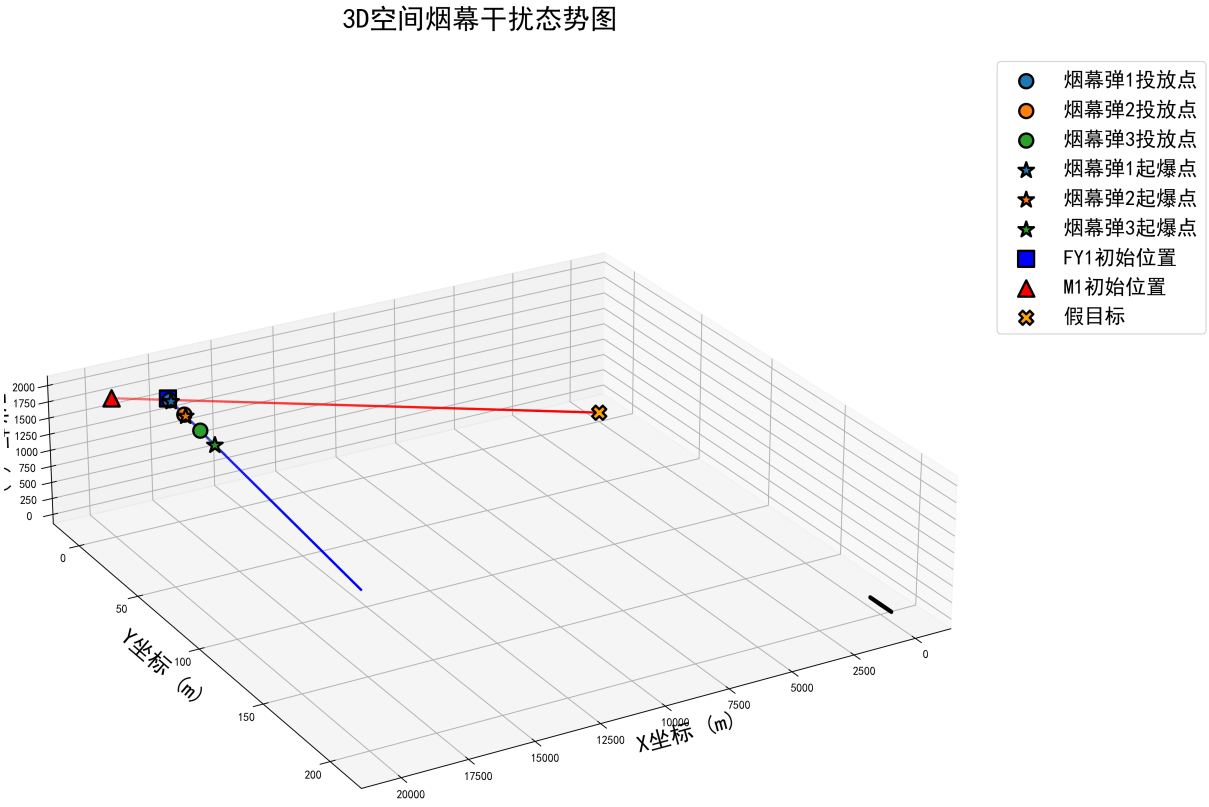

图 **9**　**3D** 烟幕干扰态势图

七、 问题四模型的建立与求解

7.1 问题分析与建模思路

基于前三问的模型建立与求解，我们对于单架无人机透支烟幕干扰弹的场景已做了详细的分析，明确了无人机的飞行参数和烟幕弹投掷爆炸参数会对真目标的有效遮蔽时间产生影响。问题四场景拓展为多架无人机 (FY1,FY2,FY3) 各投放一个烟幕干扰弹对 M1 进行干扰，求解有效遮蔽时间。在整个过程中，每架无人机的飞行参数、烟幕弹投放与起爆时间相互影响，会使烟幕弹云团的有效遮蔽时段产生叠加或互补。为让真目标的有效遮蔽时间尽可能长，我们需分别对三架无人机的飞行速度、航向，烟幕弹的投掷和爆炸时间进行调整和优化，消除多架无人机协同可能带来的遮蔽空白，使真目标能被持续或更久地遮蔽。

由前三问启发，我们同样构建出多架无人机协同的烟幕弹投掷最优规划问题，首先确定目标函数和决策变量。目标函数是三个烟幕云团形成的有效遮蔽时间总和最大。决策变量为三架无人机的飞行参数和干扰弹的投掷时刻和爆炸时刻，共计 12 项决策参数，都为连续决策变量。该问题的约束条件主要包括了各无人机的独立运动约束与烟幕弹物理特性约束，忽略了无人机之间的位置冲突。

综上，我们希望求解最大遮蔽时间的无人机协同优化问题，得到一个合理且可行的多无人机协同方案，由于该优化问题中目标及约束条件均为复杂的非线性关系，我们决定采用智能优化算法来进行合理地解决。

7.2 无人机协同遮蔽时间优化模型的建立

问题二中已经分析得出影响烟幕遮蔽时间的四个参数，本题将无人机的数量增加到了三个，由此可以建立最大烟幕遮蔽时间的无人机协同优化模型，构建出相关决策变量、目标函数和约束条件。

7.2.1 决策变量和目标函数

决策变量包括两方面, 共计 12 项参数组合，记作 (20)

$$X_4 = (\theta_i, v_{FYi}, t_{\text{drop 1}}, \Delta t_{\text{bom 1}}, t_{\text{drop 2}}, \Delta t_{\text{bom 2}}, t_{\text{drop 3}}, \Delta t_{\text{bom 3}}) \tag{20}$$

- 无人机**飞行参数**：第 i 个无人机航向角 θ_i 和飞行速度 v_{FYi}，在确定后无法改变，无人机保持匀速直线等高飞行。($i \in [1, 3]$)
- 烟幕弹**投放参数**：第 i 个无人机在受领任务后的烟幕弹投放时刻 $t_{\text{drop i}}$ ，烟幕弹投放至起爆的时间间隔 $\Delta t_{\text{bom i}}$。

目标函数为三架无人机投射的烟幕弹形成的有效遮蔽时长 Δt_s 最大化, 每个烟幕云团产生的有效遮蔽时间区间的计算遵循问题一中使用遍历法进行求解每一时刻下的方

程 (5) 是否满足，统计得到在该参数下的有效遮蔽时长。由于不同烟幕云团的有效遮蔽时段可能产生叠加或互补，需要对整体有效遮蔽时长作重新定义。设第 i 枚烟幕弹的有效遮蔽时间区间为 $[t_{i,\,\text{start}}\,, t_{i,\,\text{end}}]\quad (i=1,2,3)$：则目标函数可以定义为无重复的总有效遮蔽区间最大化：

$$\max \bigcup_{i=1}^{3} [t_{\text{i,start}}, t_{\text{i,end}}] \tag{21}$$

目标函数可以自动处理时段重叠的情况。

7.2.2 约束条件

1. **无人机速度约束**：飞行速度需在 $70 \le v_{FY} \le 140\ \text{m/s}$ 范围内。
2. **烟幕弹起爆高度大于 0**：烟幕弹脱离无人机后仅受重力作用，需要满足烟幕弹在自由落体至地面前发生爆炸，即 $1800 - \frac{1}{2} \times 9.8 \times \Delta t_{bomi}^2 \ge 0$，计算得到 $\Delta t_{\text{bomi}} \le 19.16s$。
3. **投放时刻约束**：烟幕云团维持 20s 有效结束的时刻，不能晚于导弹抵达假目标：即 $t_{\text{drop i}} + \Delta t_{\text{bom i}} + 20 \le 66.999\ \text{s}$。
4. **无人机航向角约束**：无人机航向角范围需要在 $[0, 2\pi)$ 范围内

综上可以形成本题的约束条件公式如下：

$$\text{s.t.} \begin{cases} 70 \le v_{FY} \le 140\ \text{m/s} \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{\text{bom i}}^2 \ge 0 \\ t_{\text{drop i}} + \Delta t_{\text{bom i}} + 20 \le 66.999\ \text{s} \\ \theta \in [0, 2\pi) \end{cases} \tag{22}$$

其中 $i, j \in [1,3]$ 且 $i \ne j$。

7.2.3 最大烟幕遮蔽时间无人机协同优化模型的给出

根据以上构建的决策变量、目标函数和约束条件，我们可以构建出**最大遮蔽时间**的**无人机协同优化模型**：

$$\max \bigcup_{i=1}^{3} [t_{\text{i,start}}, t_{\text{i,end}}]$$

$$\text{s.t.} \begin{cases} 70 \le v_{FY} \le 140\ \text{m/s} \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{\text{bom i}}^2 \ge 0 \\ t_{\text{drop i}} + \Delta t_{\text{bom i}} + 20 \le 66.999\ \text{s} \\ \theta \in [0, 2\pi) \end{cases} \tag{23}$$

7.3 无人机协同优化模型的求解-改进的自适应变异粒子群算法

5.3 节已给出利用粒子群算法求解单目标优化模型的步骤，6.3 节在多烟幕弹投放遮蔽优化问题中使用改进后的粒子群算法进行了求解，对惯性权重进行自适应调整，并添加局部增强策略，以避免陷入局部最优解。问题四的解空间维度进一步增加，计算量和计算维度庞大，陷入局部最优解的可能性进一步加大，需要在问题三的基础上对粒子群算法作进一步改进，改进筛选策略，减小搜索范围。在优域内通过增强 PSO 和并行计算方式，逐步逼近最大遮蔽时间的最佳参数组合。

7.3.1 重叠分段筛选策略

图10展示了问题二中四个参数对于遮蔽时长的影响，可以发现不同参数的变化对于求解有效遮蔽时长的影响由较大差异，相对于无人机飞行方向、投放延迟时间、起爆延迟时间，改变无人机的飞行速度对于求解有效遮蔽时长不敏感。而改变其他参数对于有效遮蔽时长会产生明显差异，需要分段对无人机飞行速度参数进行合理优化，以求得最大遮蔽时长。

对此我们针对无人机飞行速度区间 $v_{FYi} \in [70, 140] m/s$ 进行分段处理，同时为了避免因分段过粗导致最优解落在分段间隙，将三架无人机的速度范围均划分为三个重叠分段，确保相邻分段下均有 4m/s 的重叠，无搜索盲区，三个速度分段为：$[70, 92] m/s$、$[88, 112] m/s$、$[108, 140] m/s$。相关的代码集成到了 segmented optimize q4() 函数中。

7.3.2 Top-K 评估筛选

对于形成的 3 架无人机的三个速度区间进行组合，共得到 27 种速度分段组合形式。通过编写代码对每一种组合下应用改进后的 PSO 优化算法，得到当前组合下的最大联合遮蔽时长，作为当前组合下的评分。将每种组合下的评分进行降序排序处理，选取前 2 种组合下评分最高的组合，后续全组合并行计算仅仅在这两个优域内处理。相对于仅使用改进后的 PSO 优化算法，通过评估筛选能明显降低搜索范围，更快找到全局最大有效遮蔽范围。相关的实现逻辑集成在程序 select topk safe() 函数和 q4 run segment combo() 函数中

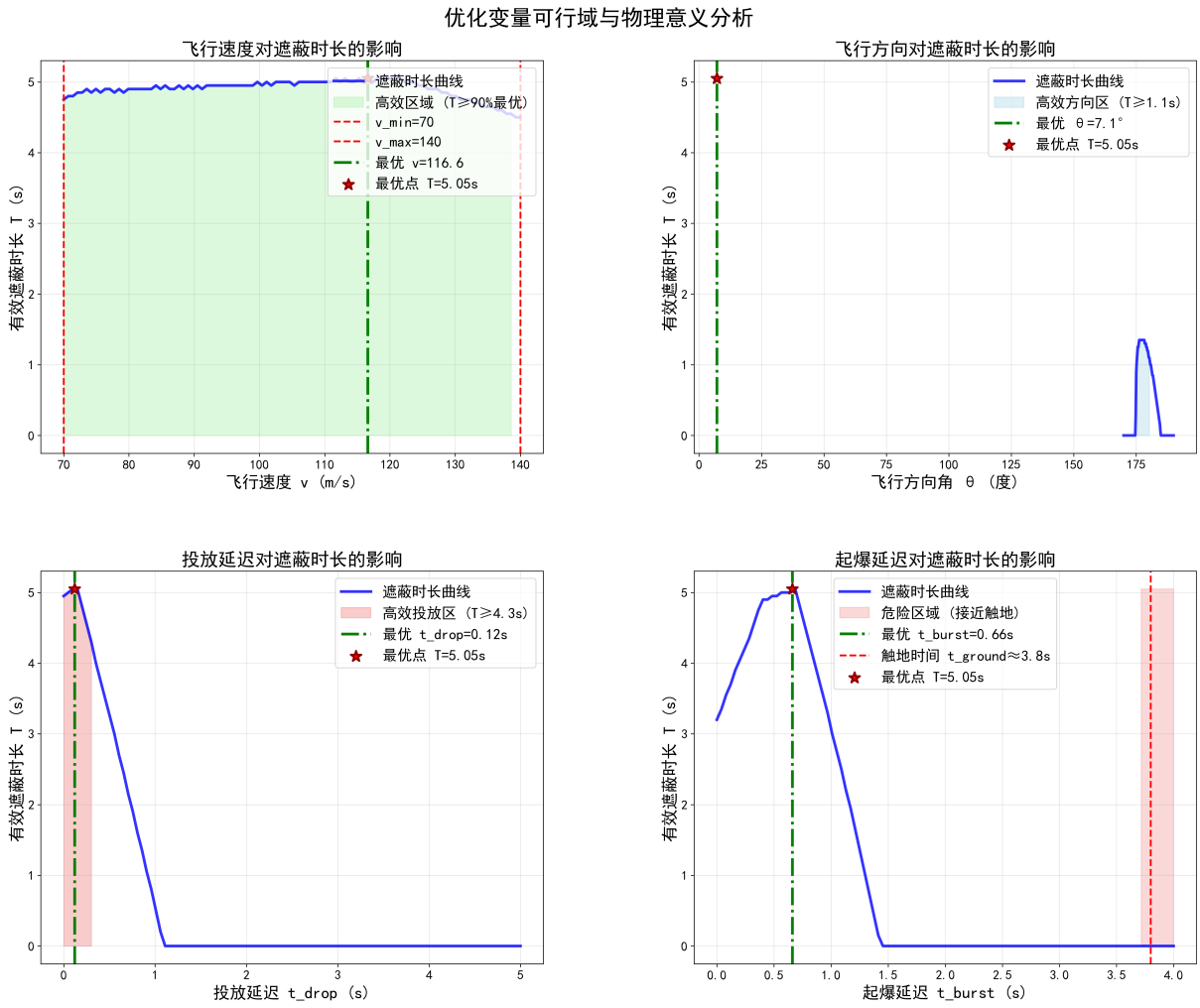

图 **10**　不同变量对遮蔽时长的影响图

7.4 结果求解与分析

通过对问题三编写的 Python 代码应用重叠分段策略和 Top-K 评估筛选，编写的代码间附录 D 我们可以评估到有效遮蔽时长最高的两个速度区间组合分别为

1. $v_{FY1} \in [70, 92], v_{FY0} \in [88, 112], v_{FY3} \in [108, 140]$
2. $v_{FY1} \in [70, 92], v_{FY0} \in [108, 140], v_{FY3} \in [108, 140]$

确定好最佳速度区间组合后，再次应用问题三中改进的 PSO 算法，初始化粒子种群数为 80，经历 200 次迭代后得到最终的每个烟幕云团的有效遮蔽时长如表1所示，得到的整体有效遮蔽时长 $\Delta t_s = 11.14s$, 将无人机的飞行参数与烟幕弹的投掷爆炸参数填入 result2.xlsx。三架无人机协同投弹形成的烟幕云团可以实现有效遮蔽时长的进一步增加，为求解问题五打下了基础。

表 **5** **result2**

无人机编号	θ_i	v_{FYi} (m/s)	投放点 (x,y,z)m	起爆点 (x,y,z)m	Δt_s
FY1	1.507°	70	(17765.01,0.92,1800)	(17575.64,5.9,1764.11)	8.14s
FY2	310°	100	(11523.66,679.09,1400)	(11108.64,45.39,1120.69)	6.71s
FY3	72.225°	130	(6426.83,-239.7,700)	(6476.5,81.48,669.38)	3s

八、 问题五模型的建立与求解

8.1 问题分析与建模思路

前四问的核心问题均围绕单一维度展开，针对的是单一干扰弹的基础遮蔽场景，而问题五进一步升级为多维度下多场景的有效遮蔽时间优化问题，不仅考虑多枚导弹来袭，还对每个无人机的投弹量进行资源分配来求解整体有效遮蔽时间。解决此类问题首先要构建决策变量和目标函数，在问题五中，决策变量的维度激增，不仅包括了 5 架无人机的飞行参数，还包括了每架无人机投弹数量和投掷爆炸参数。决策变量之间相互耦合，互相影响，难以通过调整某一参数实现遮蔽时间最大化。同时，离散决策变量和连续决策变量的同时出现也让问题变得更加复杂。目标函数方面，需要综合考虑对三枚导弹产生的遮蔽时长，烟幕云团的放置不仅需要实现单个导弹的遮蔽时间最大化，还要平衡多导弹的有效遮蔽时长。整个的问题求解是一个复杂非线性优化问题，其核心是在多枚导弹的遮蔽需求之间进行优先级平衡。

约束条件主要由几方面组成：(1) 无人机速度和方位约束，每架无人机携带的干扰弹数量约束 (2) 每架无人机的投弹间隔至少为 1s(3) 干扰弹从投放到爆炸的时间不能晚于导弹抵达真目标，且爆炸后的烟幕云团要处于有效空域。

综上所述，我们希望求解得到一个合理的烟幕云团遮蔽协作方案。由于该优化问题中具有连续决策变量和离散决策变量，且决策变量之间的耦合性较高，选择普通的优化算法已经难以求解。借助自适应双坐标系的差分进化算法 ($ABSDE_{mv}$) 来求解这种混合变量优化问题 (Mixed-variable Optimization Problems, MVOPs)[6]。

8.2 最大烟幕遮蔽时间混合变量优化模型的建立

前四个问题中已经分析得出影响烟幕遮蔽时间的四个参数，本题将无人机的数量增加到了 5 个，并增加为 3 枚导弹同时来袭。由此可以建立最大烟幕遮蔽时间优化模型，构建出相关决策变量、目标函数和约束条件。

8.2.1 决策变量和目标函数

决策变量包括三方面,，记作 (24)

$$X_4 = \left(\theta_i, v_{FYi}, t_{i,k}^{\text{drop}}, \Delta t_{i,k}^{\text{bom}}, \delta_{i,k}\right) \tag{24}$$

其中 $t_{i,k}^{\text{drop}}$ 为第 i 架无人机投下第 k 枚烟幕弹的时间 $(k = 1, 2, 3)$，$t_{j,k}^{\text{bom}}$ 为第 j 架无人机第 k 枚干扰弹的起爆时间。

- 无人机**飞行参数**：第 i 个无人机航向角 θ_i 和飞行速度 v_{FYi}，在确定后无法改变，无人机保持匀速直线等高飞行。$(i \in [1, 5])$。
- 烟幕弹**启用逻辑**：$\delta_{i,k} \in \{0, 1\}$ 为第 i 架无人机第 k 枚干扰弹的启用标识，$\delta_{i,k} = 1$ 表示该枚干扰弹投入使用，$\delta_{i,k} = 0$ 表示不使用。
- 烟幕弹**投放参数**：第 i 个无人机在受领任务后第 k 个烟幕弹投放的时间 $t_{i,k}^{\text{drop}}$ ，第 i 架无人机第 k 枚烟幕干扰弹的时间间隔 $\Delta t_{i,k}^{\text{bom}}$ 。

目标函数为三架无人机投射的烟幕弹形成的有效遮蔽时长 Δt_s 最大化, 每个烟幕云团产生的有效遮蔽时间区间的计算遵循问题一中使用遍历法进行求解每一时刻下的方程 (5) 是否满足，统计得到在该参数下的有效遮蔽时长。由于不同烟幕云团的有效遮蔽时段可能产生叠加或互补，需要对整体有效遮蔽时长作重新定义。设第 i 个无人机的 k 个烟幕弹的有效遮蔽时间区间为 $[t_{i,k,\text{ start}}, t_{i,k,\text{ end}}] \quad (i = 1, 2, ..., 5), \quad (k = 1, 2, 3)$ 则目标函数可以定义为无重复的总有效遮蔽区间最大化：

$$\max \bigcup_{i=1}^{5} [t_{\text{i,k,start}}, t_{\text{i,k,end}}] \tag{25}$$

目标函数可以自动处理时段重叠的情况。

8.2.2 约束条件

1. **无人机速度约束**：飞行速度需在 $70 \leq v_{FYi} \leq 140$ m/s 范围内。
2. **烟幕弹投放数量约束**：每架无人机至多可以投放 3 个烟幕干扰弹，因此对每架无人机需要满足 $\sum_{k=1}^{3} \delta_{i,k} \leq 3$。
3. **投弹间隔时间约束**：根据要求，每架无人机投放烟幕干扰弹的间隔时间至少间隔 1s，对同一无人机的任意两枚启用的烟幕干扰弹需满足 $\left| t_{i,k_1}^{\text{drop}} - t_{i,k_2}^{\text{drop}} \right| \geq 1 (k_1 \neq k_2)$。
4. **烟幕弹起爆高度大于 0**：烟幕弹脱离无人机后仅受重力作用，需要满足烟幕弹在自由落体至地面前发生爆炸，即 $1800 - \frac{1}{2} \times 9.8 \times \Delta t_{i,k}^{bomi^2} \geq 0$，计算得到 $\Delta t_{i,k}^{\text{bom}} \leq 19.16s$。
5. **投放时刻约束**：烟幕云团维持 20s 有效结束的时刻，不能晚于导弹抵达假目标：即 $t_{i,k}^{\text{drop}} + \Delta t_{i,k}^{\text{bom}} + 20 \leq 66.999$ s。
6. **无人机航向角约束**：无人机航向角范围需要在 $[0, 2\pi)$ 范围内, 即 $\theta_i \in [0, 2\pi)$。

综上可以形成本题的约束条件公式如下：

$$\text{s.t.}\begin{cases} 70 \le v_{FY_i} \le 140\ \mathrm{m/s} \\ \sum_{k=1}^{3} \delta_{i,k} \le 3 \\ \left| t_{i,k_1}^{\text{drop}} - t_{i,k_2}^{\text{drop}} \right| \ge 1\,(k_1 \ne k_2) \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{i,k}^{\text{bomi }2} \ge 0 \\ t_{i,k}^{\text{drop}} + \Delta t_{i,k}^{\text{bom}} + 20 \le 66.999\ \mathrm{s} \\ \theta_i \in [0, 2\pi) \end{cases} \tag{26}$$

其中 $i \in [1,5]$,$k \in [1,3]$且$k_1 \ne k_2$。

8.2.3 最大烟幕遮蔽时间混合变量优化模型的给出

根据以上构建的决策变量、目标函数和约束条件，我们可以构建出以**最大遮蔽时间**的**单目标优化模型**：

$$\max \bigcup_{i=1}^{5} [t_{\text{i,k,start}}, t_{\text{i,k,end}}]$$

$$\text{s.t.}\begin{cases} 70 \le v_{FY} \le 140\ \mathrm{m/s} \\ 1800 - \frac{1}{2} \times 9.8 \times \Delta t_{\text{bom i}}^{2} \ge 0 \\ t_{\text{drop i}} + \Delta t_{\text{bom i}} + 20 \le 66.999\ \mathrm{s} \\ \theta \in [0, 2\pi) \end{cases} \tag{27}$$

8.3 最大烟幕遮蔽时间优化模型的求解-自适应双坐标系差分进化算法

构建的决策变量中 $\delta_{i,k}$ 属于逻辑变量，取值为 0 或 1，其余都为连续决策变量。决策变量具有较大的耦合性，对此可以使用特征坐标变换的形式，降低变量之间的相关性，提高差分进化算法的搜索能力。差分进化算法属于启发式随机搜索算法，主要通过变异、交叉和选择三种进化操作来进行迭代 [7]。下面是我们利用自适应差分进化算法求解的过程：

Step1 生成初始种群：对于逻辑变量 $\delta_{i,k}$ 也编码成实数作为初始个体，设置种群规模为 50，迭代次数为 200。

Step2 个体评估：对于每个个体，将离散变量维度编码的实数进行四舍五入，得到可行的参数组合，计算当前形成的有效遮蔽时长。

Step3 坐标系变换：定义前 g 代的成功率为（28）（其中 Num_S 为子代个体计算的有效遮蔽时间优于父代的数目，NP 为种群数量），并分析第 g 代的成功率和上一代的成功率相比是否由提高，若有提升，则保持原始坐标系不变，否则变换到特征坐标系。

$$SR(g) = \frac{Num\_S}{NP} \tag{28}$$

Step4 变换特征坐标系：选择当前种群中前 20% 的优质个体每个决策变量之间的协方差矩阵 $\boldsymbol{C}^g=(c_{i,j},c_{i,j}=\mathrm{cov}(i,j))$，并对其进行特征分解，再在该坐标系下进行后续操作。

$$\boldsymbol{C}^g=Q^g\bigwedge^g(Q^g)^{-1} \tag{29}$$

Step5 变异与交叉：在选定的坐标系下为每个个体应用多种变异策略与参数组合，生成若干个候选个体。对候选个体与当前最优个体的有效遮蔽时间进行评估，计算与最优个体的距离，选出最优候选个体，进行交叉操作，得到实验个体。

Step6 选择更新：将试验个体映射回原坐标系，评估适应度，并与父代比较，对个体进行淘汰或保存，形成下一代种群。

当种群迭代次数达到设定的最大迭代次数下，算法停止运行，并输出当前最优个体和最大遮蔽时间。图11是总结的自适应双坐标系差分进化算法的流程图

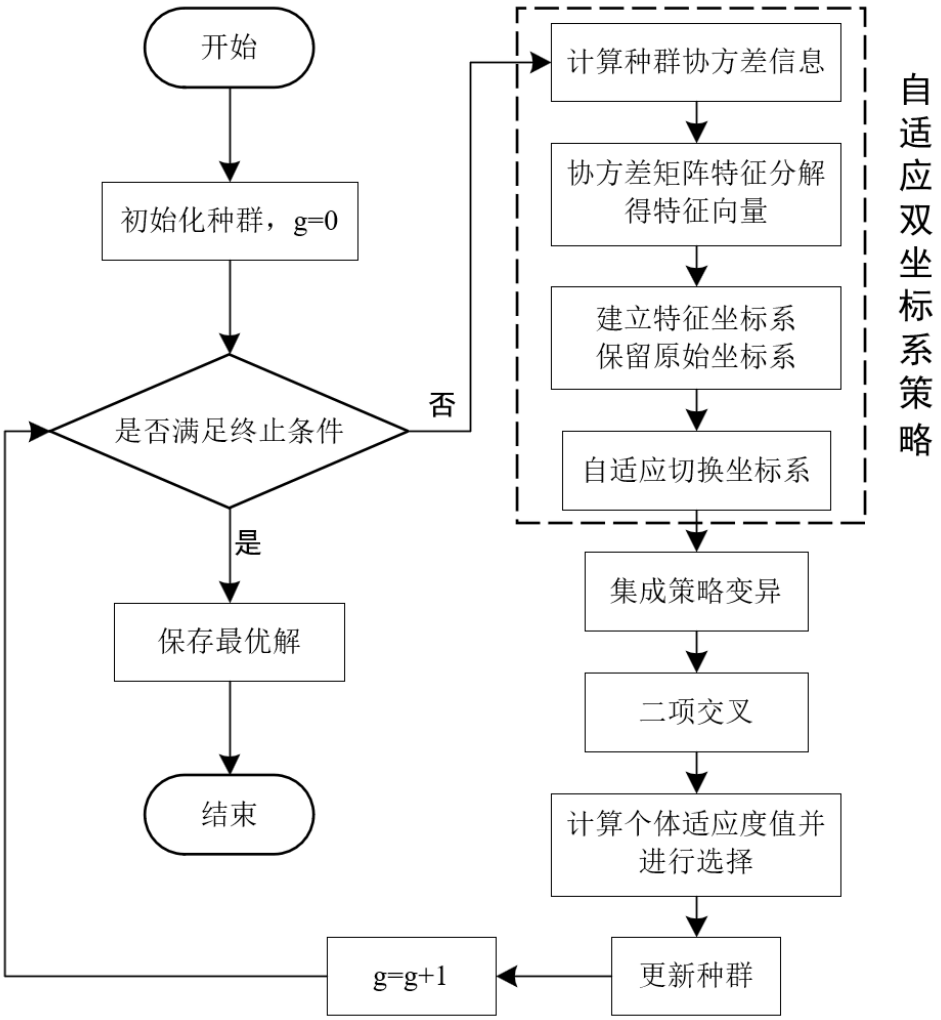

图 **11**　自适应双坐标系差分进化算法流程图

8.3.1 结果求解

团队编写 Python 程序呈现整个 $ABSDE_{mv}$ 的完整求解过程，见附录 E。得出最大总有效遮蔽时长下的无人机飞行参数与烟幕弹投掷参数如表6所示，并将求解结果填入 result3.xlsx。k 表示第 i 个无人机投掷的第 k 个烟幕弹，“-”表示该烟幕弹未对导弹实现干扰。

表 **6　result3**

FYi	k	投放点坐标	起爆点坐标	导弹编号	有效干扰时长
1	1	(17800,0,1800)	(17800,0,1800)	1	2.73s
	2	(17871.35,9.19,1800)	(17904.62,13.47,1789.93)	2	4.25s
	3	(19130.02,171.28,1800)	(19198.75,180.14,1795.45)	-	-
2	1	(12417.67, 516.26, 1400)	(12461.34, 423.85, 1396.83)	2	4.98s
	2	(12695.97, -72.59, 1400	(12829.59, -355.33, 1370.28)	-	-
	3	(13401.28, -1564.96, 1400)	(13787.62, -2382.41, 1151.6)	-	-
3	1	(6518.7, -259, 700)	(6529.81, -200.28, 697.39)	3	2.54s
	2	(6533.9, -178.5, 700)	(6570.02, 12.21, 672.49)	1	2.42s
	3	(6549.2, -98.03, 700)	(6587.25, 103.25, 669.38)	-	-
4	1	(11013.7, 2000, 1800)	(11030.9, 2000, 1799.7)	-	-
	2	(11083.7, 2000, 1800)	(11188.7, 2000, 1788.97)	-	-
	3	(11153.7, 2000, 1800)	(11210.2, 2000, 1796.8)	-	-
5	1	(12169, -553.2, 1300)	(12076.5, -392.2, 1284.7)	3	2s
	2	(12091.8, -418.7, 1300)	(11844.6, 11.6, 1190.8)	1	3.96s
	3	(11921.3, -121.8, 1300)	(11827.9, 40.7, 1284.4)	-	-

有部分无人机携带的烟幕弹起爆后并未产生有效干扰，故形成的最佳无人机投掷策略时，可以不携带该烟幕弹。第四架无人机携带的烟幕干扰弹都无法对导弹产生有效干扰，原因是 FY4 所处位置与所有导弹来袭位置相冲突，无法通过调整相关参数实现有效遮蔽。统计得到最长有效遮蔽时间 $\bigcup_{i=1}^{5}[t_{\mathrm{i,k,start}}, t_{\mathrm{i,k,end}}]$ =**19.62s**。

九、 模型评价

9.0.1 模型的优点

前四问模型的建立都以最大遮蔽时长展开，我们采用粒子群优化算法，并根据每题的变量维度，求解复杂度，对粒子群算法进行全面改进，有效避免了陷入最优解的情况，同时提高了算法搜索解空间的效率，逼近最大遮蔽时长。相较于单一的粒子群算法求解，应用改进后的自适应惯性权重与局部搜索策略粒子群算法在求解时，全局搜索效率提高了近 20%。针对问题五的复杂多变量组合优化问题，将离散变量和连续变量进行连续解空间编码，应用自适应坐标变换的差分进化方法对问题进行求解，增强了模型对复杂多变的实际场景的适应性，能够根据具体问题的特点灵活调整求解方法 [8]。

9.0.2 模型的缺点

在处理多变量、多约束问题时，使用的改进粒子群算法计算复杂度仍较高，还需要对搜索效率，局部最优值避免等环节进行调整。[8]

十、 参考文献与引用

参考文献

[1] 屈力刚，杨忠文，杨野光，等．采用改进粒子群算法的铣削参数优化研究［J］．机械设计与制造，$2022, 377(7) : 187 - 191$ ．．

[2] 吕柏行, 郭志光, 赵韦皓, 等. 标准粒子群算法的优化方式综述 [J]. 科学技术创新,2021,(28):33-37.

[3] CLERC M, KENNEDY J. The particle swarm-explosion, stability, and convergence in a multidimensional complex space[J]. IEEE Transactions on Evolutionary Computation, 2002, 6(1): 58-73.

[4] 敖永才, 师奕兵, 张伟, 等. 自适应惯性权重的改进粒子群算法 [J]. 电子科技大学学报,2014,43(06):874-880.

[5] SHI Y H, EBERHART R C. Parameter selection in particle swarm optimization[C]//7th International Conference, Evolutionary Programming VII. Berlin Heidelberg: Springer, 1998, 1447: 591-600.

[6] 周新宇, 黄君洪, 彭虎, 等. 自适应双坐标系的差分进化算法求解混合变量优化问题 [J]. 计算机学报,2024,47(09):2116-2140.

[7] 李建林, 高坡, 刘志强. 基于差分进化算法的风电场机组功率曲线优化研究 [J]. 机电工程技术,2025,54(11):109-112.

[8] doubao,doubao-深度思考，字节跳动，2025-09-07

[9] DeepSeek，DeepSeek － V3，深度求索，2025 － 09 － 07

附录 A　问题一的 Python 代码

```python
import numpy as np
import math
import os

# 定义模拟所需的所有物理常数和初始参数。
class SimulationConfig:
    GRAVITY_ACCEL = 9.8  # 重力加速度 (m/s²)
    UAV_SPEED = 120  # 无人机恒定飞行速度 (m/s)
    UAV_INITIAL_POS = np.array([17800.0, 0.0, 1800.0])  # 无人机初始坐标 (x,y,z)
    MISSILE_SPEED = 300  # 导弹恒定飞行速度 (m/s)
    MISSILE_INITIAL_POS = np.array([20000.0, 0.0, 2000.0])  # 导弹初始坐标 (x,y,z)
    DROP_DELAY_TIME = 1.5  # 从任务开始到烟幕弹投放的时间延迟 (s)
    BURST_DELAY_TIME = 3.6  # 从投放到烟幕起爆的时间 (s)
    SMOKE_SINK_RATE = 3.0  # 烟幕云垂直下沉的速度 (m/s)
    SMOKE_EFFECTIVE_RADIUS = 10.0  # 烟幕球体的半径 (m)
    SMOKE_LIFESPAN = 20.0  # 烟幕起爆后保持有效的持续时间 (s)
# 目标参数
ASSET_SPECS = {
    "base_center": np.array([0.0, 200.0, 0.0]), # 圆柱形目标底面的中心
    "radius": 7.0,  # 圆柱体半径 (m)
    "height": 10.0  # 圆柱体高度 (m)}

def calculate_squared_distance(pos1: np.ndarray, pos2: np.ndarray) -> float:
return np.sum((pos1 - pos2) ** 2)

def check_line_sphere_collision(line_start: np.ndarray, line_end: np.ndarray,
sphere_center: np.ndarray, sphere_radius: float) -> bool:

#判断一条线段是否与一个球体相交。
vec_line = line_end - line_start
vec_center_to_start = sphere_center - line_start

line_len_sq = calculate_squared_distance(line_start, line_end)
if line_len_sq < 1e-9: # 处理线段几乎是一个点的情况
return calculate_squared_distance(line_start, sphere_center) <= sphere_radius ** 2
projection_ratio = np.dot(vec_center_to_start, vec_line) / line_len_sq

# 找到无限直线上离球心最近的点
clamped_ratio = np.clip(projection_ratio, 0.0, 1.0)
closest_point_on_segment = line_start + clamped_ratio * vec_line

# 检查最近点到球心的距离是否在球体半径之内
dist_to_center_sq = calculate_squared_distance(closest_point_on_segment,
    sphere_center)
```

```
return dist_to_center_sq <= sphere_radius ** 2

def create_asset_sample_points() -> list[np.ndarray]:
asset_points = []
spec = SimulationConfig.ASSET_SPECS
base_center = spec["base_center"]
asset_radius = spec["radius"]
asset_height = spec["height"]

axis_heights = np.linspace(0, asset_height, 5) # 从底部到顶部共5个点
for z_offset in axis_heights:
asset_points.append(np.array([base_center[0], base_center[1], base_center[2] +
    z_offset]))
height_levels = [0, asset_height / 2, asset_height] # 底部、中部、顶部
points_per_circle = 16
for z_level in height_levels:
for i in range(points_per_circle):
   angle = 2.0 * math.pi * i / points_per_circle
   x_coord = base_center[0] + asset_radius * math.cos(angle)
   y_coord = base_center[1] + asset_radius * math.sin(angle)
   z_coord = base_center[2] + z_level
   asset_points.append(np.array([x_coord, y_coord, z_coord]))
return asset_points

def compute_uav_location(time_elapsed: float) -> np.ndarray:
   flight_direction = np.array([-1.0, 0.0, 0.0])
   return SimulationConfig.UAV_INITIAL_POS + SimulationConfig.UAV_SPEED *
       flight_direction * time_elapsed

def compute_munition_burst_location() -> np.ndarray:
#计算烟幕弹药起爆点的三维坐标。
   drop_pos = compute_uav_location(SimulationConfig.DROP_DELAY_TIME)
   munition_velocity = np.array([-SimulationConfig.UAV_SPEED, 0.0, 0.0])
   time_to_ground = math.sqrt(2.0 * drop_pos[2] / SimulationConfig.GRAVITY_ACCEL)
   time_of_flight = min(SimulationConfig.BURST_DELAY_TIME, time_to_ground)

# 使用标准运动学方程计算最终的起爆位置
   burst_x = drop_pos[0] + munition_velocity[0] * time_of_flight
   burst_y = drop_pos[1] # y方向没有水平速度
   burst_z = drop_pos[2] - 0.5 * SimulationConfig.GRAVITY_ACCEL * (time_of_flight
       ** 2)
return np.array([burst_x, burst_y, max(0.0, burst_z)])

def compute_missile_location(mission_time: float) -> np.ndarray:
   trajectory_vector = -SimulationConfig.MISSILE_INITIAL_POS
   # 归一化得到单位方向向量
   direction_unit_vec = trajectory_vector / np.linalg.norm(trajectory_vector)
```

```
88  return SimulationConfig.MISSILE_INITIAL_POS + SimulationConfig.MISSILE_SPEED *
        direction_unit_vec * mission_time
89
90  def compute_smoke_center_location(mission_time: float, burst_pos: np.ndarray) ->
        np.ndarray:
91      #在给定任务时间计算烟幕云的中心位置。
92
93      total_burst_time = SimulationConfig.DROP_DELAY_TIME +
            SimulationConfig.BURST_DELAY_TIME
94      time_since_burst = max(0.0, mission_time - total_burst_time)
95      smoke_z = burst_pos[2] - SimulationConfig.SMOKE_SINK_RATE * time_since_burst
96      return np.array([burst_pos[0], burst_pos[1], max(0.0, smoke_z)])
97
98  def analyze_shielding_duration() -> dict:
99  #主分析函数，用于计算总有效遮蔽时长。
100 # 预计算固定的位置和对象
101     munition_burst_pos = compute_munition_burst_location()
102     asset_key_points = create_asset_sample_points()
103     burst_event_time = SimulationConfig.DROP_DELAY_TIME +
            SimulationConfig.BURST_DELAY_TIME
104     missile_impact_time = np.linalg.norm(SimulationConfig.MISSILE_INITIAL_POS) /
            SimulationConfig.MISSILE_SPEED
105     sim_start_time = burst_event_time
106     sim_end_time = min(burst_event_time + SimulationConfig.SMOKE_LIFESPAN,
            missile_impact_time)
107     if sim_start_time >= sim_end_time:
108 return {"duration": 0.0, "timestamps": [], "start": 0.0, "end": 0.0}
109
110 # 模拟循环设置
111 time_discretization_step = 0.01
112 num_steps = math.ceil((sim_end_time - sim_start_time) / time_discretization_step)
113 shielded_timestamps = []
114
115 for step in range(num_steps):
116     current_time = sim_start_time + step * time_discretization_step
117     missile_pos = compute_missile_location(current_time)
118     smoke_pos = compute_smoke_center_location(current_time, munition_burst_pos)
119
120 current_smoke_radius = SimulationConfig.SMOKE_EFFECTIVE_RADIUS
121 if smoke_pos[2] < current_smoke_radius:
122 current_smoke_radius = smoke_pos[2] # 半径不能大于其离地面的高度
123
124 # 检查在此刻目标的任何部分是否被遮蔽
125     is_shielded_this_step = False
126 for point in asset_key_points:
127     if check_line_sphere_collision(missile_pos, point, smoke_pos,
            current_smoke_radius):
```

```
            is_shielded_this_step = True
        break # 如果一个点被遮蔽，则认为整个目标在该时间步被遮蔽
        if is_shielded_this_step:
            shielded_timestamps.append(current_time)
        if not shielded_timestamps:
    return {"duration": 0.0, "timestamps": [], "start": 0.0, "end": 0.0}

        total_shield_time = len(shielded_timestamps) * time_discretization_step
    return {
        "duration": round(total_shield_time, 4),
        "timestamps": shielded_timestamps,
        "start": round(shielded_timestamps[0], 4),
        "end": round(shielded_timestamps[-1], 4)
    }

    def display_results(results: dict):
    #格式化并打印结构化的结果报告。
    print("\n" + "=" * 80)
    print("烟幕弹药遮蔽分析 - 详细报告")
    print("=" * 80)
    print("\n[1. 场景参数]")
    print(f"  - 无人机初始位置: {SimulationConfig.UAV_INITIAL_POS} m")
    print(f"  - 导弹初始位置: {SimulationConfig.MISSILE_INITIAL_POS} m")
    print(f"  - 目标底部中心: {SimulationConfig.ASSET_SPECS['base_center']} m")
    print(f"  - 目标尺寸: 半径={SimulationConfig.ASSET_SPECS['radius']}m,
        高度={SimulationConfig.ASSET_SPECS['height']}m")
    burst_pos = compute_munition_burst_location()
    burst_time = SimulationConfig.DROP_DELAY_TIME + SimulationConfig.BURST_DELAY_TIME
    impact_time = np.linalg.norm(SimulationConfig.MISSILE_INITIAL_POS) /
        SimulationConfig.MISSILE_SPEED
    print("\n[2. 运动学计算]")
    print(f"  - 弹药起爆位置: {np.round(burst_pos, 4)} m")
    print(f"  - 弹药起爆时间: {burst_time:.2f} s")
    print(f"  - 预计导弹命中时间: {impact_time:.4f} s")
    print("\n[3. 最终遮蔽结果]")
    print(f"  - 总有效遮蔽时长: {results['duration']:.4f} s")
    print(f"  - 遮蔽窗口开始时间: {results['start']:.4f} s")
    print(f"  - 遮蔽窗口结束时间: {results['end']:.4f} s")
    print("\n[4. 详细遮蔽时间段]")
    if results["timestamps"]:
    intervals = []
    start_interval = results["timestamps"][0]
    for i in range(1, len(results["timestamps"])):
    # 如果时间戳之间的间隔大于步长，则视为一个新的时间段
    if results["timestamps"][i] - results["timestamps"][i-1] > 0.011:
        end_interval = results["timestamps"][i-1]
        intervals.append((start_interval, end_interval))
```

```
    start_interval = results["timestamps"][i]
    intervals.append((start_interval, results["timestamps"][-1])) # 添加最后一个时间段
for i, (start, end) in enumerate(intervals, 1):
    print(f" - 时间段 {i}: [{start:.4f}s - {end:.4f}s] (持续: {end-start:.4f}s)")
else:
print(" - 未找到有效遮蔽时间段。")
print("\n" + "=" * 80)
def main():
    simulation_results = analyze_shielding_duration()
    display_results(simulation_results)
    print("模拟完成。")

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

附录 B　粒子群算法–问题二的 Python 程序

```
import numpy as np
import math
from typing import Tuple, List, Dict, Any

class SimulationParameters:
#封装所有仿真所需的物理常数、初始条件和模型参数。
    def __init__(self):
        self.GRAVITY_ACCELERATION: float = 9.8  # 重力加速度 (m/s²)
        # 烟幕弹属性
        self.SMOKE_DESCENT_SPEED: float = 3.0       # 烟幕中心匀速下沉速度 (m/s)
        self.SMOKE_EFFECTIVE_RADIUS: float = 10.0  # 烟幕有效遮蔽半径 (m)
        self.SMOKE_LIFESPAN: float = 20.0         # 烟幕有效持续时长 (s)
        #导弹属性
        self.MISSILE_SPEED: float = 300.0  # 导弹M1的飞行速度 (m/s)
        self.MISSILE_STARTING_POINT: np.ndarray = np.array([20000.0, 0.0, 2000.0])
            # 导弹M1的初始位置 (x,y,z)
        # FY1初始状态
        self.UAV_STARTING_POINT: np.ndarray = np.array([17800.0, 0.0, 1800.0])  #
            无人机FY1的初始位置 (x,y,z)
        #真目标（圆柱体）规格
        self.TARGET_SPECS: Dict[str, Any] = {
            "base_center": np.array([0.0, 200.0, 0.0]),  # 圆柱体下底面圆心 (x,y,z)
            "radius": 7.0,                                # 圆柱体半径 (m)
            "height": 10.0                                # 圆柱体高度 (m)}

        # PSO 优化参数
        self.PSO_CONFIG: Dict[str, Any] = {
```

```
            'population_size': 50,
            'max_iterations': 200,
            'inertia_weight': 0.7,
            'cognitive_coefficient': 1.5,
            'social_coefficient': 1.5}

        self.OPTIMIZATION_BOUNDS: Dict[str, Tuple[float, float]] = {
            'uav_velocity': (70.0, 140.0),
            'flight_angle': (0.0, 2 * math.pi),
            'drop_delay': (0.1, 15.0),
            'burst_delay': (0.1, 10.0)}
        CONFIG = SimulationParameters()
        _TARGET_SAMPLE_POINTS_CACHE = None
    def get_target_sample_points() -> np.ndarray:
        #获取目标的采样点数组。
            global _TARGET_SAMPLE_POINTS_CACHE
        if _TARGET_SAMPLE_POINTS_CACHE is None:
            _TARGET_SAMPLE_POINTS_CACHE = _generate_target_points()
        return _TARGET_SAMPLE_POINTS_CACHE

def _generate_target_points() -> np.ndarray:
    points = []
    center = CONFIG.TARGET_SPECS["base_center"]
    radius = CONFIG.TARGET_SPECS["radius"]
    height = CONFIG.TARGET_SPECS["height"]

    for h_ratio in [0.0, 0.25, 0.5, 0.75, 1.0]:
        z = center[2] + height * h_ratio
        points.append([center[0], center[1], z])

# 上下底面及中层圆周采样
    for z_level in [center[2], center[2] + height / 2, center[2] + height]:
        num_points = 16 if z_level != center[2] + height / 2 else 8
    for i in range(num_points):
        angle = i * 2 * math.pi / num_points
        x = center[0] + radius * math.cos(angle)
        y = center[1] + radius * math.sin(angle)
        points.append([x, y, z_level])
    return np.array(points)

class SimulationEngine:#封装整个计算过程的核心逻辑，包括轨迹计算和遮蔽评估。
    def __init__(self):
        self.cache = {}
    def _get_uav_pos(self, t: float, vel: float, angle: float) -> np.ndarray:
        direction = np.array([math.cos(angle), math.sin(angle), 0])
        return CONFIG.UAV_STARTING_POINT + vel * direction * t
    def _get_missile_pos(self, t: float) -> np.ndarray:
```

```
        direction = -CONFIG.MISSILE_STARTING_POINT
        unit_dir = direction / np.linalg.norm(direction)
        return CONFIG.MISSILE_STARTING_POINT + CONFIG.MISSILE_SPEED * unit_dir * t
    def _get_detonation_params(self, vel: float, angle: float, t_drop: float,
        t_burst: float) -> Tuple[np.ndarray, float]:
        drop_pos = self._get_uav_pos(t_drop, vel, angle)
        bomb_vel = vel * np.array([math.cos(angle), math.sin(angle), 0])
        t_ground = math.sqrt(max(0.0, 2.0 * drop_pos[2] /
            CONFIG.GRAVITY_ACCELERATION))
        air_time = min(t_burst, t_ground)
        burst_pos_x = drop_pos[0] + bomb_vel[0] * air_time
        burst_pos_y = drop_pos[1] + bomb_vel[1] * air_time
        burst_pos_z = max(0.0, drop_pos[2] - 0.5 * CONFIG.GRAVITY_ACCELERATION *
            (air_time ** 2))
        detonation_point = np.array([burst_pos_x, burst_pos_y, burst_pos_z])
        total_detonation_time = t_drop + t_burst
    return detonation_point, total_detonation_time
    def _get_smoke_center_pos(self, t: float, t_burst_total: float, p_burst:
        np.ndarray) -> np.ndarray:
        if t < t_burst_total:
        return p_burst
            sink_time = t - t_burst_total
            smoke_z = p_burst[2] - CONFIG.SMOKE_DESCENT_SPEED * sink_time
        return np.array([p_burst[0], p_burst[1], max(0.0, smoke_z)])

    def evaluate_shielding_duration(self, params: Dict[str, float]) ->
        float:#计算给定参数下的有效遮蔽时长。
        key = tuple(sorted(params.items()))
        if key in self.cache:
            return self.cache[key]
            vel, angle, t_drop, t_burst = params['uav_velocity'],
                params['flight_angle'], params['drop_delay'], params['burst_delay']
            p_burst, t_burst_total = self._get_detonation_params(vel, angle, t_drop,
                t_burst)
            target_points = get_target_sample_points()
        missile_hit_time = np.linalg.norm(CONFIG.MISSILE_STARTING_POINT) /
            CONFIG.MISSILE_SPEED
        start_time = t_burst_total
        end_time = min(t_burst_total + CONFIG.SMOKE_LIFESPAN, missile_hit_time)
        if start_time >= end_time:
        return 0.0
            time_step = 0.05
            effective_steps = 0

        num_steps = int(np.ceil((end_time - start_time) / time_step))
    for i in range(num_steps):
        current_time = start_time + i * time_step
```

```
        p_missile = self._get_missile_pos(current_time)
        p_smoke = self._get_smoke_center_pos(current_time, t_burst_total, p_burst)
        vec_missile_to_target = target_points - p_missile
        vec_missile_to_smoke = p_smoke - p_missile
        dot_product = np.sum(vec_missile_to_target * vec_missile_to_smoke, axis=1)
        len_sq_missile_target = np.sum(vec_missile_to_target**2, axis=1)
        t = np.divide(dot_product, len_sq_missile_target,
            where=len_sq_missile_target > 1e-9, out=np.zeros_like(dot_product))
        t_clamped = np.clip(t, 0, 1)
        closest_points = p_missile + t_clamped[:, np.newaxis] * vec_missile_to_target
        dist_sq_to_smoke = np.sum((closest_points - p_smoke)**2, axis=1)
    if np.any(dist_sq_to_smoke <= CONFIG.SMOKE_EFFECTIVE_RADIUS**2):
        effective_steps += 1
        result = effective_steps * time_step
        self.cache[key] = result
    return result

# 粒子群优化（PSO）求解器
class ParticleSwarmOptimizer:#实现粒子群优化算法，用于寻找最优的无人机投放策略。
    def __init__(self, engine: SimulationEngine):
        self.engine = engine
        self.bounds = CONFIG.OPTIMIZATION_BOUNDS
        self.pso_params = CONFIG.PSO_CONFIG
        self.dim = len(self.bounds)
        self.lower_bounds = np.array([self.bounds[k][0] for k in self.bounds])
        self.upper_bounds = np.array([self.bounds[k][1] for k in self.bounds])
        self.range = self.upper_bounds - self.lower_bounds

    def solve(self) -> Dict[str, Any]:
        # 初始化种群
        pop_size = self.pso_params['population_size']
        positions = self.lower_bounds + np.random.rand(pop_size, self.dim) *
            self.range
        velocities = np.zeros((pop_size, self.dim))
        personal_best_pos = positions.copy()
        personal_best_fitness = np.array([self._objective_function(p) for p in
            positions])
        global_best_idx = np.argmax(personal_best_fitness)
        global_best_pos = personal_best_pos[global_best_idx].copy()
        global_best_fitness = personal_best_fitness[global_best_idx]
        history = []

    for i in range(self.pso_params['max_iterations']):
        # 更新速度和位置
        r1, r2 = np.random.rand(pop_size, self.dim), np.random.rand(pop_size,
            self.dim)
```

```
        w, c1, c2 = self.pso_params['inertia_weight'],
            self.pso_params['cognitive_coefficient'],
            self.pso_params['social_coefficient']
        velocities = w * velocities + \
        c1 * r1 * (personal_best_pos - positions) + \
        c2 * r2 * (global_best_pos - positions)
        positions += velocities
        # 边界处理
        positions = np.clip(positions, self.lower_bounds, self.upper_bounds)
        # 评估新位置
        current_fitness = np.array([self._objective_function(p) for p in positions])
        # 更新个体最优
        update_mask = current_fitness > personal_best_fitness
        personal_best_pos[update_mask] = positions[update_mask]
        personal_best_fitness[update_mask] = current_fitness[update_mask]
        # 更新全局最优
        current_global_best_idx = np.argmax(personal_best_fitness)
        if personal_best_fitness[current_global_best_idx] > global_best_fitness:
        global_best_fitness = personal_best_fitness[current_global_best_idx]
        global_best_pos = personal_best_pos[current_global_best_idx].copy()
        history.append(global_best_fitness)
    if (i + 1) % 10 == 0:
        print(f"迭代 {i+1}/{self.pso_params['max_iterations']}, 最优遮蔽时长:
            {global_best_fitness:.4f}s")
    return self._format_results(global_best_pos, global_best_fitness)

def _objective_function(self, particle: np.ndarray) -> float:
        params = {
            'uav_velocity': particle[0],
            'flight_angle': particle[1],
            'drop_delay': particle[2],
            'burst_delay': particle[3]    }
    return self.engine.evaluate_shielding_duration(params)

def _format_results(self, best_particle: np.ndarray, best_fitness: float) ->
    Dict[str, Any]:
vel, angle, t_drop, t_burst = best_particle
        p_burst, t_burst_total = self.engine._get_detonation_params(vel, angle,
            t_drop, t_burst)
    return {
        'optimal_params': {
            'uav_velocity': vel,
            'flight_angle_rad': angle,
            'flight_angle_deg': math.degrees(angle),
            'drop_delay': t_drop,
            'burst_delay': t_burst,},
        'derived_results': {
```

```
            'detonation_point': p_burst,
            'total_detonation_time': t_burst_total,
            'max_shielding_duration': best_fitness }}
def display_final_report(results: Dict[str, Any]):#打印最终的优化结果报告。
    opt_params = results['optimal_params']
    derived = results['derived_results']
    print("\n" + "="*80)
    print("问题2: 最优烟幕投放策略 - 优化结果报告")
    print("="*80)
    print("\n【最优投放策略参数】")
    print(f"  - 无人机飞行速度: {opt_params['uav_velocity']:.4f} m/s")
    print(f"  - 飞行方向角: {opt_params['flight_angle_deg']:.2f}°
        ({opt_params['flight_angle_rad']:.4f} rad)")
    print(f"  - 投放延迟: {opt_params['drop_delay']:.4f} s")
    print(f"  - 起爆延迟: {opt_params['burst_delay']:.4f} s")
    print("\n【衍生计算结果】")
    print(f"  - 烟幕起爆点坐标: {derived['detonation_point'].round(4)}")
    print(f"  - 任务开始至起爆总时长: {derived['total_detonation_time']:.4f} s")
    print("\n【核心优化指标】")
    print(f"  - 最大有效遮蔽时长: {derived['max_shielding_duration']:.4f} s")
    print("\n" + "="*80)

if __name__ == "__main__":
print("启动问题2的粒子群优化求解器...")
simulation_engine = SimulationEngine()
pso_solver = ParticleSwarmOptimizer(simulation_engine)
optimal_solution = pso_solver.solve()
display_final_report(optimal_solution)
```

附录 C　改进粒子群算法–问题三的 Python 程序

```
import numpy as np
import os
import math
import pandas as pd

# ======================== 1. 参数配置 ========================
# 结果保存路径
out_dir = 'q3_fy1_3bombs_optimal'
os.makedirs(out_dir, exist_ok=True)

# 物理常数（固定不变）
G = 9.8            # 重力加速度
SINK_SPD = 3       # 烟幕下沉速度
SMOKE_R = 10       # 烟幕有效半径
```

```
SMOKE_T = 20       # 烟幕有效时长
M_SPD = 300        # 导弹速度

# 初始位置
U0 = np.array([17800, 0, 1800])  # 无人机初始位置
M0 = np.array([20000, 0, 2000])  # 导弹初始位置

# 导弹命中时间（提前计算好）
M_HIT_T = np.linalg.norm(M0) / M_SPD

# 目标参数（圆柱体）
TARGET = {
    "center": np.array([0, 200, 0]),  # 目标中心
    "r": 7,      # 半径
    "h": 10      # 高度
}

# 优化参数范围（8个变量）
BOUNDS = {
    'v': (70, 140),            # 无人机速度
    'a': (0, 2 * math.pi),     # 飞行方向角
    't1': (0.1, M_HIT_T - 20 - 0.1 - 2),  # 第1枚投放延迟
    't2': (0.1, M_HIT_T - 20 - 0.1 - 1),  # 第2枚投放延迟
    't3': (0.1, M_HIT_T - 20 - 0.1),      # 第3枚投放延迟
    'tb1': (0.1, 15),          # 第1枚起爆延迟
    'tb2': (0.1, 15),          # 第2枚起爆延迟
    'tb3': (0.1, 15)           # 第3枚起爆延迟
}

# PSO参数
PSO_CFG = {
    'n': 50,       # 种群大小
    'max_i': 100, # 最大迭代
    'w_s': 0.9,    # 初始惯性权重
    'w_e': 0.4,    # 最终惯性权重
    'c1': 2.0,     # 认知系数
    'c2': 2.0      # 社会系数
}

# 全局配置
CFG = {
    'dt': 0.1,     # 时间步长
    'ground': 0.0 # 地面高度
}

# 弹投放间隔（固定1秒）
DROP_INT = 1
```

```
62
63  # 半空间裁剪开关
64  USE_CLIP = True
65
66  # ======================== 2. 工具函数 ========================
67  def dist_sq(p1, p2):
68  """计算两点距离平方（省去开方，提高效率）"""
69  return np.sum((p1 - p2) ** 2)
70
71  def make_target_points():
72  """生成目标采样点（圆柱体表面和内部关键点）"""
73  points = []
74  c = TARGET["center"]
75  r = TARGET["r"]
76  h = TARGET["h"]
77
78  # 中心轴点（5个）
79  for z in [0, h/4, h/2, 3*h/4, h]:
80  points.append([c[0], c[1], z])
81
82  # 下底面圆周（16个点）
83  for i in range(16):
84  ang = i * math.pi / 8
85  x = r * math.cos(ang)
86  y = c[1] + r * math.sin(ang)
87  points.append([x, y, 0])
88
89  # 上底面圆周（16个点）
90  for i in range(16):
91  ang = i * math.pi / 8
92  x = r * math.cos(ang)
93  y = c[1] + r * math.sin(ang)
94  points.append([x, y, h])
95
96  # 中间层圆周（8个点）
97  for i in range(8):
98  ang = i * math.pi / 4
99  x = r * math.cos(ang)
100 y = c[1] + r * math.sin(ang)
101 points.append([x, y, h/2])
102
103 return np.array(points, dtype=float)
104
105 # 预生成目标点（全局使用）
106 T_POINTS = make_target_points()
107
108 def fix_particle(p):
```

```
    """修复粒子参数，确保满足约束条件"""
    v, a, t1, t2, t3, tb1, tb2, tb3 = p

    # 1. 参数范围限制
    v = max(BOUNDS['v'][0], min(BOUNDS['v'][1], v))
    a = max(BOUNDS['a'][0], min(BOUNDS['a'][1], a))
    t1 = max(BOUNDS['t1'][0], min(BOUNDS['t1'][1], t1))
    t2 = max(BOUNDS['t2'][0], min(BOUNDS['t2'][1], t2))
    t3 = max(BOUNDS['t3'][0], min(BOUNDS['t3'][1], t3))
    tb1 = max(BOUNDS['tb1'][0], min(BOUNDS['tb1'][1], tb1))
    tb2 = max(BOUNDS['tb2'][0], min(BOUNDS['tb2'][1], tb2))
    tb3 = max(BOUNDS['tb3'][0], min(BOUNDS['tb3'][1], tb3))

    # 2. 投放间隔约束（t2≥t1+1，t3≥t2+1）
    if t2 < t1 + DROP_INT:
    t2 = t1 + DROP_INT
    if t3 < t2 + DROP_INT:
    t3 = t2 + DROP_INT

    # 3. 烟幕有效时间约束（起爆总时间≤导弹命中前20秒）
    max_t = M_HIT_T - 20
    if t1 + tb1 > max_t:
    tb1 = max(0.1, max_t - t1)
    if t2 + tb2 > max_t:
    tb2 = max(0.1, max_t - t2)
    if t3 + tb3 > max_t:
    tb3 = max(0.1, max_t - t3)

    return np.array([v, a, t1, t2, t3, tb1, tb2, tb3])

    # ======================== 3. 运动模型 ========================
    def uav_pos(t, v, a):
    """计算无人机位置（匀速直线运动）"""
    # 飞行方向向量（水平面）
    d = np.array([math.cos(a), math.sin(a), 0])
    return U0 + v * d * t

    def bomb_params(v, a, td, tb):
    """计算烟幕弹起爆参数（平抛运动）"""
    # 投放点位置
    p0 = uav_pos(td, v, a)

    # 炸弹水平速度
    v_bomb = v * np.array([math.cos(a), math.sin(a), 0])

    # 落地时间（如果炸弹会落地）
    t_ground = math.sqrt(2 * p0[2] / G) if p0[2] > 0 else 0
```

```python

# 实际起爆时间（不能晚于落地）
tb_actual = t_ground if (t_ground > 0 and tb > t_ground) else max(tb, 0)

# 起爆点坐标
xb = p0[0] + v_bomb[0] * tb_actual
yb = p0[1] + v_bomb[1] * tb_actual
zb = max(p0[2] - 0.5 * G * tb_actual**2, CFG['ground'])

# 总起爆时间
t_total = td + tb_actual

return np.array([xb, yb, zb]), t_total

def missile_pos(t):
"""计算导弹位置（匀速直线飞向原点）"""
# 方向向量（指向原点）
d = -M0
d_unit = d / np.linalg.norm(d)
return M0 + M_SPD * d_unit * t

def smoke_pos(t, t_burst, p_burst):
"""计算烟幕中心位置（匀速下沉）"""
if t < t_burst:  # 未起爆
return p_burst

# 下沉时间
t_sink = t - t_burst

# 烟幕高度（不低于地面）
z = max(p_burst[2] - SINK_SPD * t_sink, CFG['ground'])

return np.array([p_burst[0], p_burst[1], z])

# ======================== 4. 遮蔽判断 ========================
def is_occluded(p_m, p_s, r_eff):
"""判断导弹到目标的视线是否被烟幕遮蔽"""
if r_eff < 1e-3:  # 烟幕半径太小
return False

# 导弹到目标点的向量
a = p_m  # 导弹位置
qs = T_POINTS  # 目标点
ab = qs - a  # 线段向量

# 导弹到烟幕中心的向量
ac = p_s - a
```

```
203
204 # 计算投影参数t
205 len2_ab = np.sum(ab**2, axis=1)
206 valid = len2_ab > 1e-12  # 有效线段
207
208 t_proj = np.zeros(len(len2_ab))
209 t_proj[valid] = np.sum(ab[valid] * ac, axis=1) / len2_ab[valid]
210
211 # 限制t在[0,1]范围内
212 t_clamped = np.clip(t_proj, 0.0, 1.0)
213
214 # 线段上最近点
215 p_near = a + t_clamped.reshape(-1, 1) * ab
216
217 # 最近点到烟幕中心的距离平方
218 d2 = np.sum((p_near - p_s) ** 2, axis=1)
219
220 # 判断是否遮蔽（考虑半空间裁剪）
221 if USE_CLIP:
222 return np.any((d2 <= r_eff**2) & (p_near[:, 2] >= CFG['ground']))
223 else:
224 return np.any(d2 <= r_eff**2)
225
226 def calc_shelter_time(p):
227 """计算3枚弹的联合遮蔽时间（目标函数）"""
228 v, a, t1, t2, t3, tb1, tb2, tb3 = p
229
230 # 计算3枚弹的起爆参数
231 p1_b, t1_t = bomb_params(v, a, t1, tb1)
232 p2_b, t2_t = bomb_params(v, a, t2, tb2)
233 p3_b, t3_t = bomb_params(v, a, t3, tb3)
234
235 # 确定时间窗口
236 t_start = min(t1_t, t2_t, t3_t)
237 t_end = min(max(t1_t + SMOKE_T, t2_t + SMOKE_T, t3_t + SMOKE_T), M_HIT_T)
238
239 if t_start >= t_end:
240 return 0.0  # 无有效时间
241
242 # 时间离散化
243 dt = CFG['dt']
244 n_steps = int(math.ceil((t_end - t_start) / dt))
245 count = 0  # 有效时刻计数
246
247 for i in range(n_steps):
248 t = min(t_start + i * dt, t_end)
249
```

```
# 导弹位置
p_m = missile_pos(t)

# 检查3枚弹是否有遮蔽
occluded = False

# 检查第1枚
if t1_t <= t <= t1_t + SMOKE_T:
p_s = smoke_pos(t, t1_t, p1_b)
r_eff = SMOKE_R if USE_CLIP else min(SMOKE_R, p_s[2])
if is_occluded(p_m, p_s, r_eff):
occluded = True

# 检查第2枚（第1枚无效时才检查）
if not occluded and t2_t <= t <= t2_t + SMOKE_T:
p_s = smoke_pos(t, t2_t, p2_b)
r_eff = SMOKE_R if USE_CLIP else min(SMOKE_R, p_s[2])
if is_occluded(p_m, p_s, r_eff):
occluded = True

# 检查第3枚（前两枚无效时才检查）
if not occluded and t3_t <= t <= t3_t + SMOKE_T:
p_s = smoke_pos(t, t3_t, p3_b)
r_eff = SMOKE_R if USE_CLIP else min(SMOKE_R, p_s[2])
if is_occluded(p_m, p_s, r_eff):
occluded = True

if occluded:
count += 1

return count * dt

# ======================== 5. PSO优化 ========================
def run_pso():
"""PSO优化主函数"""
n_pop = PSO_CFG['n']
max_iter = PSO_CFG['max_iter']

# 初始化种群
particles = []
for _ in range(n_pop):
# 随机生成粒子参数
v = np.random.uniform(BOUNDS['v'][0], BOUNDS['v'][1])
a = np.random.uniform(BOUNDS['a'][0], BOUNDS['a'][1])
t1 = np.random.uniform(BOUNDS['t1'][0], BOUNDS['t1'][1])
t2 = np.random.uniform(BOUNDS['t2'][0], BOUNDS['t2'][1])
t3 = np.random.uniform(BOUNDS['t3'][0], BOUNDS['t3'][1])
```

```
tb1 = np.random.uniform(BOUNDS['tb1'][0], BOUNDS['tb1'][1])
tb2 = np.random.uniform(BOUNDS['tb2'][0], BOUNDS['tb2'][1])
tb3 = np.random.uniform(BOUNDS['tb3'][0], BOUNDS['tb3'][1])

# 修复约束
p = fix_particle(np.array([v, a, t1, t2, t3, tb1, tb2, tb3]))
particles.append(p)

particles = np.array(particles)

# 初始化速度（小随机值）
velocities = np.zeros_like(particles)
for i in range(8):
min_val = [BOUNDS[k][0] for k in ['v', 'a', 't1', 't2', 't3', 'tb1', 'tb2',
    'tb3']][i]
max_val = [BOUNDS[k][1] for k in ['v', 'a', 't1', 't2', 't3', 'tb1', 'tb2',
    'tb3']][i]
velocities[:, i] = np.random.uniform(-0.05*(max_val-min_val),
    0.05*(max_val-min_val), n_pop)

# 初始化最优解
p_best = particles.copy()
p_best_val = np.array([calc_shelter_time(p) for p in particles])

g_best_idx = np.argmax(p_best_val)
g_best = p_best[g_best_idx].copy()
g_best_val = p_best_val[g_best_idx]

print(f"PSO初始化完成，最佳遮蔽时间：{g_best_val:.2f}s")

# 迭代优化
for iter in range(1, max_iter + 1):
# 计算当前惯性权重（线性递减）
w = PSO_CFG['w_s'] - (PSO_CFG['w_s'] - PSO_CFG['w_e']) * (iter / max_iter)

# 更新每个粒子
for i in range(n_pop):
r1, r2 = np.random.rand(), np.random.rand()

# 速度更新
velocities[i] = (w * velocities[i] +
PSO_CFG['c1'] * r1 * (p_best[i] - particles[i]) +
PSO_CFG['c2'] * r2 * (g_best - particles[i]))

# 位置更新
particles[i] += velocities[i]

```

```
341 # 修复约束
342 particles[i] = fix_particle(particles[i])
343
344 # 计算当前适应度
345 curr_val = np.array([calc_shelter_time(p) for p in particles])
346
347 # 更新个体最优
348 update_mask = curr_val > p_best_val
349 p_best[update_mask] = particles[update_mask].copy()
350 p_best_val[update_mask] = curr_val[update_mask]
351
352 # 更新全局最优
353 curr_best_val = np.max(p_best_val)
354 if curr_best_val > g_best_val:
355 g_best_val = curr_best_val
356 g_best = p_best[np.argmax(p_best_val)].copy()
357
358 # 打印进度
359 if iter % 10 == 0:
360 v_opt, a_opt, t1_opt, t2_opt, t3_opt = g_best[:5]
361 print(f"迭代 {iter:3d}/{max_iter}: 最佳时间 = {g_best_val:.2f}s | "
362 f"速度={v_opt:.1f}m/s | 角度={math.degrees(a_opt):.0f}° | "
363 f"投放时间={t1_opt:.1f}/{t2_opt:.1f}/{t3_opt:.1f}s")
364
365 # 提取最优参数
366 v_opt, a_opt, t1_opt, t2_opt, t3_opt, tb1_opt, tb2_opt, tb3_opt = g_best
367
368 # 计算各弹参数
369 p1_drop = uav_pos(t1_opt, v_opt, a_opt)
370 p1_burst, t1_total = bomb_params(v_opt, a_opt, t1_opt, tb1_opt)
371
372 p2_drop = uav_pos(t2_opt, v_opt, a_opt)
373 p2_burst, t2_total = bomb_params(v_opt, a_opt, t2_opt, tb2_opt)
374
375 p3_drop = uav_pos(t3_opt, v_opt, a_opt)
376 p3_burst, t3_total = bomb_params(v_opt, a_opt, t3_opt, tb3_opt)
377
378 # 整理结果
379 result = {
380 'v': round(v_opt, 2),
381 'a_deg': round(math.degrees(a_opt), 1),
382 'total_time': round(g_best_val, 2),
383 'bomb1': {
384 't_drop': round(t1_opt, 2), 'tb': round(tb1_opt, 2),
385 'p_drop': np.round(p1_drop, 2), 'p_burst': np.round(p1_burst, 2),
386 't_total': round(t1_total, 2)
387 },
```

```
    'bomb2': {
        't_drop': round(t2_opt, 2), 'tb': round(tb2_opt, 2),
        'p_drop': np.round(p2_drop, 2), 'p_burst': np.round(p2_burst, 2),
        't_total': round(t2_total, 2)
    },
    'bomb3': {
        't_drop': round(t3_opt, 2), 'tb': round(tb3_opt, 2),
        'p_drop': np.round(p3_drop, 2), 'p_burst': np.round(p3_burst, 2),
        't_total': round(t3_total, 2)
    }
}

return result

# ======================== 6. 结果保存 ========================
def save_result(result):
"""保存结果到Excel"""
data = []

for i in [1, 2, 3]:
bomb = result[f'bomb{i}']
data.append({
    '无人机编号': 'FY1',
    '飞行速度(m/s)': result['v'],
    '飞行方向角(°)': result['a_deg'],
    '烟幕弹编号': i,
    '投放点X(m)': bomb['p_drop'][0],
    '投放点Y(m)': bomb['p_drop'][1],
    '投放点Z(m)': bomb['p_drop'][2],
    '起爆点X(m)': bomb['p_burst'][0],
    '起爆点Y(m)': bomb['p_burst'][1],
    '起爆点Z(m)': bomb['p_burst'][2],
    '投放延迟(s)': bomb['t_drop'],
    '起爆延迟(s)': bomb['tb'],
    '联合遮蔽时长(s)': result['total_time']
})

df = pd.DataFrame(data)
file_path = os.path.join(out_dir, 'result1.xlsx')

with pd.ExcelWriter(file_path, engine='openpyxl') as writer:
df.to_excel(writer, sheet_name='烟幕弹最优参数', index=False)

# 添加说明
note = pd.DataFrame({'说明': ['飞行方向角：以X轴为正向，逆时针为正（0~360°）']})
note.to_excel(writer, sheet_name='烟幕弹最优参数', startrow=len(df)+2, index=False)
```

```
print(f"\n结果已保存: {file_path}")

# ======================== 7. 主程序 ========================
if __name__ == "__main__":
print("=" * 60)
print("FY1无人机3枚烟幕弹最优投放策略")
print("=" * 60)

# 执行优化
print("\n开始PSO优化...")
best_result = run_pso()

# 输出结果
print("\n" + "=" * 60)
print("最优参数")
print("=" * 60)

print(f"1. 无人机速度: {best_result['v']} m/s")
print(f"2. 飞行方向角: {best_result['a_deg']} °")
print(f"3. 联合遮蔽时间: {best_result['total_time']} s")

for i in [1, 2, 3]:
bomb = best_result[f'bomb{i}']
print(f"\n第{i}枚弹:")
print(f"  投放延迟: {bomb['t_drop']} s | 起爆延迟: {bomb['tb']} s")
print(f"  投放点: {bomb['p_drop']}")
print(f"  起爆点: {bomb['p_burst']}")
print(f"  起爆时间: {bomb['t_total']} s")

# 保存结果
save_result(best_result)
print("\n计算完成!")
```

附录 D　重叠分段筛选的改进粒子群算法——问题四的 Python 程序

```
import numpy as np
import os
import math
import pandas as pd
from typing import Tuple, List, Dict, Any
import time
from scipy.optimize import minimize
import concurrent.futures
import multiprocessing as mp
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor
```

```python
from functools import partial

output_dir = 'q4_3drones_optimal'
if not os.path.exists(output_dir):
os.makedirs(output_dir)
g = 9.8  # 重力加速度 (m/s²)
v_smoke_sink = 3  # 烟幕匀速下沉速度 (m/s)
smoke_radius = 10  # 烟幕有效半径 (m)
smoke_valid_time = 20  # 烟幕有效时长 (s)
v_M1 = 300  # 导弹M1飞行速度 (m/s)
# 3架无人机初始位置
P_FY1_initial = np.array([17800, 0, 1800])  # FY1初始位置 (x,y,z)
P_FY2_initial = np.array([12000, 1400, 1400])  # FY2初始位置 (x,y,z)
P_FY3_initial = np.array([6000, -3000, 700])  # FY3初始位置 (x,y,z)
P_M1_initial = np.array([20000, 0, 2000])  # 导弹M1初始位置 (x,y,z)
TARGET_SAMPLES = None  # 全局缓存真目标采样点（列表）
TARGET_SAMPLES_ARR = None  # 全局缓存真目标采样点（N×3数组）
M1_HIT_TIME = np.linalg.norm(P_M1_initial) / v_M1  # 导弹命中假目标时间≈67.0s
target_params = {
    "center": np.array([0, 200, 0]),  # 真目标下底面圆心 (x,y,z)
    "radius": 7,  # 真目标半径 (m)
    "height": 10  # 真目标高度 (m)
}
# 优化变量范围
OPT_BOUNDS = [
# FY1参数
(70, 140),      # v1: FY1飞行速度 (m/s)
(-np.pi, np.pi),  # theta1: FY1飞行方向角 (rad)
(0, 30),        # t1: FY1投放时间 (s)
(0.1, 15),      # tb1: FY1起爆延迟 (s)
# FY2参数
(70, 140),      # v2: FY2飞行速度 (m/s)
(-np.pi, np.pi),  # theta2: FY2飞行方向角 (rad)
(0, 30),        # t2: FY2投放时间 (s)
(0.1, 15),      # tb2: FY2起爆延迟 (s)
# FY3参数
(70, 140),      # v3: FY3飞行速度 (m/s)
(-np.pi, np.pi),  # theta3: FY3飞行方向角 (rad)
(0, 30),        # t3: FY3投放时间 (s)
(0.1, 15),      # tb3: FY3起爆延迟 (s)
]
#将12维变量归一化到[0,1]区间
NORMALIZATION_PARAMS = []
for i, (low, high) in enumerate(OPT_BOUNDS):
NORMALIZATION_PARAMS.append({
    'min': low,
    'max': high,
```

```
        'range': high - low})
    # 改进的PSO + 局部搜索 + 遗传算法混合参数（增强防局部最优）
PSO_PARAMS = {
    'pop_size': 100,        # 种群大小（进一步增加以提升多样性）
    'max_iter': 500,        # 最大迭代次数
    'w_start': 0.9,         # 初始惯性权重
    'w_end': 0.4,           # 最终惯性权重
    'c1': 2.0,              # 个体学习因子
    'c2': 2.0,              # 社会学习因子
    'local_search_prob': 0.3,  # 局部搜索概率（增加）
    'local_search_iter': 30,    # 局部搜索迭代次数（增加）
    'early_stop_patience': 60,  # 早停耐心值（增加）
    'restart_patience': 40,  # 新增：连续无改进的重启耐心
    # 遗传算法增强参数
    'mutation_rate': 0.2,  # 变异率（增加）
    'crossover_rate': 0.85,  # 交叉率（增加）
    'elite_ratio': 0.15,    # 精英保留比例（减少，增加多样性）
    'diversity_threshold': 0.005,  # 多样性阈值（更严格）
    'restart_ratio': 0.4,    # 重启比例（增加）
    'restart_iterations': 20,  # 重启间隔（减少）
    'genetic_interval': 10,  # 遗传算法执行间隔（条件触发）
    'adaptive_mutation': True,  # 自适应变异
    # 新增：lbest 概率
    'lbest_prob': 0.6,
    # 并行计算参数
    'parallel_batch_size': 64,  # 并行计算批处理大小
    'parallel_workers': None,    # 并行工作进程数（None表示自动检测）
    'use_parallel': True,        # 是否启用并行计算  }

ENHANCED_PSO = {
    'use_time_varying': True,
    'c1_start': 2.5, 'c1_end': 0.5,
    'c2_start': 0.5, 'c2_end': 2.5,
    'vel_frac': 0.2,            # 最大速度占边界范围比例
    'immigrant_period': 8,      # 更频繁的移民
    'immigrant_ratio': 0.15,    # 增大移民比例
    'jitter_period': 6,         # 更频繁的抖动
    'jitter_sigma': 0.04,       # 略增强抖动强度
    # 新增高级防局部最优策略
    'use_chaos_mutation': True,         # 混沌变异
    'chaos_mutation_prob': 0.1,         # 混沌变异概率
    'use_levy_flight': True,            # 莱维飞行
    'levy_flight_prob': 0.15,           # 莱维飞行概率
    'use_opposition_learning': True,    # 对立学习
    'opposition_prob': 0.1,             # 对立学习概率
    'use_quantum_behavior': True,       # 量子行为
    'quantum_prob': 0.08,               # 量子行为概率
```

```
    'use_adaptive_mutation': True,      # 自适应变异
    'adaptive_mutation_rate': 0.2,      # 自适应变异率
    'use_elite_opposition': True,       # 精英对立学习
    'elite_opposition_prob': 0.05,      # 精英对立学习概率
    'use_restart_with_elite': True,     # 精英保留重启
    'elite_retention_ratio': 0.2,       # 精英保留比例
    'use_multi_swarm': True,            # 多群优化
    'swarm_count': 3,                   # 子群数量
    'swarm_migration_prob': 0.1,        # 子群迁移概率
    'use_tabu_search': True,            # 禁忌搜索
    'tabu_tenure': 10,                  # 禁忌期限
    'tabu_list_size': 20,               # 禁忌列表大小
    'use_simulated_annealing': True,    # 模拟退火
    'sa_initial_temp': 100.0,           # 初始温度
    'sa_cooling_rate': 0.95,            # 冷却率
    'use_cultural_algorithm': True,     # 文化算法
    'cultural_influence_prob': 0.1,     # 文化影响概率  }

def normalize_particle(particle: np.ndarray) -> np.ndarray:
    """将12维粒子从原始空间归一化到[0,1]区间"""
    normalized = np.zeros(12)
    for i in range(12):
    param_info = NORMALIZATION_PARAMS[i]
    normalized[i] = (particle[i] - param_info['min']) / param_info['range']
    return normalized

def denormalize_particle(normalized_particle: np.ndarray) -> np.ndarray:
    """将12维粒子从[0,1]区间反归一化到原始空间"""
    denormalized = np.zeros(12)
    for i in range(12):
    param_info = NORMALIZATION_PARAMS[i]
    denormalized[i] = normalized_particle[i] * param_info['range'] +
        param_info['min']
    return denormalized

def generate_target_samples() -> list[np.ndarray]:
    """生成真目标圆柱体的关键采样点"""
    global TARGET_SAMPLES
    if TARGET_SAMPLES is not None:
    return TARGET_SAMPLES
    samples: list[np.ndarray] = []
    center = target_params["center"]
    r = target_params["radius"]
    h = target_params["height"]
    for z in [0, h/4, h/2, 3*h/4, h]:
    samples.append(np.array([center[0], center[1], z]))
    bottom_z = 0
```

```
    for i in range(8):
    ang = i * math.pi / 4
    samples.append(np.array([r * math.cos(ang), center[1] + r * math.sin(ang),
        bottom_z]))
    for i in range(8):
    ang = i * math.pi / 4 + math.pi / 8
    samples.append(np.array([r * math.cos(ang), center[1] + r * math.sin(ang),
        bottom_z]))
    top_z = h
    for i in range(8):
    ang = i * math.pi / 4
    samples.append(np.array([r * math.cos(ang), center[1] + r * math.sin(ang),
        top_z]))
    for i in range(8):
    ang = i * math.pi / 4 + math.pi / 8
    samples.append(np.array([r * math.cos(ang), center[1] + r * math.sin(ang),
        top_z]))
    mid_z = h/2
    for i in range(8):
    ang = i * math.pi / 4
    samples.append(np.array([r * math.cos(ang), center[1] + r * math.sin(ang),
        mid_z]))
    TARGET_SAMPLES = samples
    return samples

def get_target_samples_array() -> np.ndarray:
    """获取真目标采样点数组用于向量化。"""
    global TARGET_SAMPLES_ARR
    if TARGET_SAMPLES_ARR is None:
    TARGET_SAMPLES_ARR = np.array(generate_target_samples(), dtype=float)
    return TARGET_SAMPLES_ARR

def line_segment_sphere_intersect(P1: np.ndarray, P2: np.ndarray,
sphere_center: np.ndarray, sphere_radius: float) -> bool:
    """判断线段是否与球体相交"""
    d = P2 - P1
    d_norm_sq = np.dot(d, d)
    if d_norm_sq < 1e-10:  # 线段退化为点
       return np.linalg.norm(P1 - sphere_center) <= sphere_radius
    t = max(0, min(1, np.dot(sphere_center - P1, d) / d_norm_sq))
    closest_point = P1 + t * d
    distance = np.linalg.norm(closest_point - sphere_center)
    return distance <= sphere_radius

def lhs_init(pop_size: int, bounds: List[Tuple[float, float]]) -> np.ndarray:
    dim = len(bounds)
    X = np.zeros((pop_size, dim))
```

```
        # 分层取值 in (0,1)
        base = (np.arange(pop_size) + np.random.rand(pop_size)) / pop_size
        for j, (low, high) in enumerate(bounds):
        vals = base.copy()
        np.random.shuffle(vals)
        X[:, j] = low + vals * (high - low)
        return X

    def repair_particle_3drones(particle: np.ndarray, is_normalized: bool = False)
        -> np.ndarray:
        Args:
        particle: 12维粒子向量
        is_normalized: 是否为归一化空间中的粒子
        if is_normalized:
            # 归一化空间: 先反归一化, 修复, 再归一化
            denormalized = denormalize_particle(particle)
            repaired = repair_particle_3drones(denormalized, is_normalized=False)
            return normalize_particle(repaired)
            repaired = particle.copy()

        for i, (low, high) in enumerate(OPT_BOUNDS):
            repaired[i] = np.clip(repaired[i], low, high)
            return repaired

    def calculate_diversity(particles: np.ndarray) -> float:
        #计算种群多样性
        if len(particles) <= 1:
            return 0.0
        total_distance = 0.0
        count = 0
        for i in range(len(particles)):
            for j in range(i + 1, len(particles)):
                distance = np.linalg.norm(particles[i] - particles[j])
                total_distance += distance
                count += 1
            return total_distance / count if count > 0 else 0.0

    def crossover_operation(parent1: np.ndarray, parent2: np.ndarray) ->
        Tuple[np.ndarray, np.ndarray]:
        #交叉操作
        alpha = np.random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = (1 - alpha) * parent1 + alpha * parent2
        # 修复到可行域
        child1 = repair_particle_3drones(child1)
        child2 = repair_particle_3drones(child2)
        return child1, child2
```

```
238
239 def mutation_operation(particle: np.ndarray, iteration: int, max_iter: int) ->
        np.ndarray:
240     #变异操作（自适应高斯变异）
241     mutated = particle.copy()
242     # 自适应变异率：随迭代次数增加而减少
243     base_mutation_rate = PSO_PARAMS['mutation_rate']
244     adaptive_rate = base_mutation_rate * (1 - iteration / max_iter)
245     for i in range(len(OPT_BOUNDS)):
246         if np.random.random() < adaptive_rate:
247             low, high = OPT_BOUNDS[i]
248         sigma = (high - low) * 0.1
249         mutation = np.random.normal(0, sigma)
250         mutated[i] = np.clip(mutated[i] + mutation, low, high)
251     return mutated
252
253 def chaos_mutation(particle: np.ndarray, iteration: int, max_iter: int) ->
        np.ndarray:
254     """混沌变异：使用混沌映射生成变异"""
255     mutated = particle.copy()
256     # 使用Logistic混沌映射
257     chaos_param = 3.9
258     chaos_value = np.random.random()
259     for _ in range(10):  # 混沌迭代
260     chaos_value = chaos_param * chaos_value * (1 - chaos_value)
261     # 基于混沌值进行变异
262     for i in range(len(particle)):
263     if np.random.random() < ENHANCED_PSO['chaos_mutation_prob']:
264     # 混沌变异强度随迭代递减
265     mutation_strength = 0.1 * (1 - iteration / max_iter) * chaos_value
266     mutated[i] += np.random.normal(0, mutation_strength)
267     return repair_particle_3drones(mutated)
268
269 def levy_flight(particle: np.ndarray, global_best: np.ndarray) -> np.ndarray:
270     """莱维飞行"""
271     if np.random.random() > ENHANCED_PSO['levy_flight_prob']:
272         return particle
273         # 莱维分布参数
274         alpha = 1.5
275         beta = 1.0
276     # 生成莱维步长
277         u = np.random.normal(0, 1, len(particle))
278         v = np.random.normal(0, 1, len(particle))
279     # 莱维步长计算
280         levy_step = u / (np.abs(v) ** (1/alpha))
281         levy_step = np.clip(levy_step, -1, 1)  # 限制步长范围
282     new_particle = particle + 0.01 * levy_step * (global_best - particle)
```

```
283         return repair_particle_3drones(new_particle)
284
285     def opposition_learning(particle: np.ndarray) -> np.ndarray:
286         """对立学习：生成当前解的对立解"""
287         if np.random.random() > ENHANCED_PSO['opposition_prob']:
288         return particle
289
290         # 计算对立解
291         opposition = np.zeros_like(particle)
292         for i, (low, high) in enumerate(OPT_BOUNDS):
293             opposition[i] = low + high - particle[i]
294             return repair_particle_3drones(opposition)
295
296     def quantum_behavior(particle: np.ndarray, personal_best: np.ndarray,
            global_best: np.ndarray) -> np.ndarray:
297         if np.random.random() > ENHANCED_PSO['quantum_prob']:
298             return particle
299         # 量子半径计算
300         quantum_radius = np.linalg.norm(global_best - personal_best) / 2.0
301         # 量子位置更新
302         quantum_particle = np.zeros_like(particle)
303         for i in range(len(particle)):
304         # 量子不确定性
305         uncertainty = np.random.normal(0, quantum_radius)
306         quantum_particle[i] = global_best[i] + uncertainty
307         return repair_particle_3drones(quantum_particle)
308
309     def adaptive_mutation(particle: np.ndarray, fitness: float, avg_fitness: float,
            iteration: int, max_iter: int) -> np.ndarray:
310         """自适应变异：根据适应度差异调整变异强度"""
311         if np.random.random() > ENHANCED_PSO['adaptive_mutation_rate']:
312             return particle
313         mutated = particle.copy()
314         # 计算适应度差异
315         fitness_diff = abs(fitness - avg_fitness) / (avg_fitness + 1e-8)
316         mutation_strength = 0.1 * fitness_diff * (1 - iteration / max_iter)
317         for i in range(len(particle)):
318         if np.random.random() < 0.3:  # 30%的维度参与变异
319             mutation = np.random.normal(0, mutation_strength)
320             mutated[i] += mutation
321         return repair_particle_3drones(mutated)
322
323     def elite_opposition_learning(particles: np.ndarray, fitness: np.ndarray) ->
            np.ndarray:
324         """精英对立学习：对精英粒子进行对立学习"""
325         if np.random.random() > ENHANCED_PSO['elite_opposition_prob']:
326             return particles
```

```
        elite_count = max(1, int(len(particles) * 0.2))
        elite_indices = np.argsort(fitness)[-elite_count:]
        new_particles = particles.copy()
        for idx in elite_indices:
    # 生成对立解
        opposition = opposition_learning(particles[idx])
        new_particles[idx] = opposition
    return new_particles

    def multi_swarm_optimization(particles: np.ndarray, fitness: np.ndarray) ->
        np.ndarray:
    """将种群分为多个子群独立优化"""
    if not ENHANCED_PSO['use_multi_swarm']:
        return particles

    pop_size = len(particles)
        swarm_count = ENHANCED_PSO['swarm_count']
        swarm_size = pop_size // swarm_count

    new_particles = particles.copy()
    for swarm_id in range(swarm_count):
        start_idx = swarm_id * swarm_size
        end_idx = start_idx + swarm_size if swarm_id < swarm_count - 1 else
            pop_size
        swarm_particles = particles[start_idx:end_idx]
        swarm_fitness = fitness[start_idx:end_idx]
        swarm_best_idx = np.argmax(swarm_fitness)
        swarm_best = swarm_particles[swarm_best_idx]
    for i in range(len(swarm_particles)):
    if np.random.random() < ENHANCED_PSO['swarm_migration_prob']:
    # 向子群最优移动
        direction = swarm_best - swarm_particles[i]
        new_particles[start_idx + i] = swarm_particles[i] + 0.1 * direction
    return new_particles

    class TabuSearch:
    #维护禁忌列表避免重复搜索
def __init__(self, tabu_tenure: int = 10, tabu_list_size: int = 20):
    self.tabu_list = []
    self.tabu_tenure = tabu_tenure
    self.tabu_list_size = tabu_list_size

def is_tabu(self, particle: np.ndarray) -> bool:
"""检查粒子是否在禁忌列表中"""
for tabu_particle, tenure in self.tabu_list:
if np.linalg.norm(particle - tabu_particle) < 1e-6:
    return True
```

```
        return False

    def add_to_tabu(self, particle: np.ndarray):
    """添加粒子到禁忌列表"""
        self.tabu_list.append((particle.copy(), self.tabu_tenure))
        # 限制禁忌列表大小
    if len(self.tabu_list) > self.tabu_list_size:
        self.tabu_list.pop(0)

    def update_tabu(self):
        """更新禁忌列表：减少禁忌期限"""
        self.tabu_list = [(particle, tenure - 1) for particle, tenure in
            self.tabu_list if tenure > 1]

    class SimulatedAnnealing:
    #模拟退火：概率性接受劣解
    def __init__(self, initial_temp: float = 100.0, cooling_rate: float = 0.95):
        self.temperature = initial_temp
        self.cooling_rate = cooling_rate

    def accept_solution(self, old_fitness: float, new_fitness: float) -> bool:
        #判断是否接受新解
        if new_fitness > old_fitness:
            return True
        delta = new_fitness - old_fitness
        if self.temperature > 0:
            accept_prob = np.exp(delta / self.temperature)
            return np.random.random() < accept_prob
            return False

    def cool_down(self):
        self.temperature *= self.cooling_rate

    class CulturalAlgorithm:
    def __init__(self):
        self.belief_space = {
            'situational': None,  # 情境知识
            'normative': None,    # 规范知识
            'topographical': None, # 地形知识    }

    def update_belief_space(self, particles: np.ndarray, fitness: np.ndarray):
        # 更新情境知识（最优解）
        best_idx = np.argmax(fitness)
        self.belief_space['situational'] = particles[best_idx].copy()
        # 更新规范知识
        self.belief_space['normative'] = {
            'lower': np.min(particles, axis=0),
```

```
        'upper': np.max(particles, axis=0)  }

    def influence_particles(self, particles: np.ndarray) -> np.ndarray:
        #信念空间影响粒子
        if np.random.random() > ENHANCED_PSO['cultural_influence_prob']:
        return particles
        influenced_particles = particles.copy()
        # 情境知识影响
        if self.belief_space['situational'] is not None:
        for i in range(len(particles)):
        if np.random.random() < 0.1:  # 10%概率受情境知识影响
        direction = self.belief_space['situational'] - particles[i]
        influenced_particles[i] += 0.05 * direction
        return influenced_particles

    def genetic_enhancement(particles: np.ndarray, fitness: np.ndarray, iteration:
        int) -> np.ndarray:
        #遗传算法增强操作
        pop_size = len(particles)
        elite_size = int(pop_size * PSO_PARAMS['elite_ratio'])
        # 选择精英
        elite_indices = np.argsort(fitness)[-elite_size:]
        elite_particles = particles[elite_indices]
        new_particles = elite_particles.copy()
        while len(new_particles) < pop_size:
        parent1_idx = np.random.choice(elite_size)
        parent2_idx = np.random.choice(elite_size)
        if np.random.random() < PSO_PARAMS['crossover_rate']:
            child1, child2 = crossover_operation(
            elite_particles[parent1_idx],
            elite_particles[parent2_idx]  )
            new_particles = np.vstack([new_particles, child1])
        if len(new_particles) < pop_size:
            new_particles = np.vstack([new_particles, child2])
        else:
            new_particles = np.vstack([new_particles, elite_particles[parent1_idx]])
            for i in range(elite_size, pop_size):  # 精英不参与变异
            new_particles[i] = mutation_operation(new_particles[i], iteration,
                PSO_PARAMS['max_iter'])
        return new_particles[:pop_size]

    def population_restart(particles: np.ndarray, fitness: np.ndarray) ->
        np.ndarray:
        pop_size = len(particles)
        restart_size = int(pop_size * PSO_PARAMS['restart_ratio'])
        elite_size = int(pop_size * PSO_PARAMS['elite_ratio'])
        elite_indices = np.argsort(fitness)[-elite_size:]
```

```
        new_particles = particles[elite_indices].copy()
        # 重新初始化部分粒子
        for i in range(restart_size):
            new_particle = np.zeros(len(OPT_BOUNDS))
        for j, (low, high) in enumerate(OPT_BOUNDS):
            new_particle[j] = np.random.uniform(low, high)
            new_particles = np.vstack([new_particles, new_particle])
            remaining_size = pop_size - len(new_particles)
        if remaining_size > 0:
            for i in range(remaining_size):
                new_particle = np.zeros(len(OPT_BOUNDS))
            for j, (low, high) in enumerate(OPT_BOUNDS):
                new_particle[j] = np.random.uniform(low, high)
                new_particles = np.vstack([new_particles, new_particle])
            return new_particles

    def get_fy_position(fy_initial: np.ndarray, v: float, theta: float, t: float)
        -> np.ndarray:
    #获取无人机在时间t的位置
        return fy_initial + np.array([v * np.cos(theta) * t, v * np.sin(theta) * t,
            0])

    def get_bomb_burst_params_3drones(fy_initial: np.ndarray, v: float, theta:
        float, t_drop: float, t_burst: float) -> Tuple[np.ndarray, np.ndarray,
        float]:
        """获取3架无人机的烟幕弹起爆参数（修正弹道计算）"""
        P_drop = get_fy_position(fy_initial, v, theta, t_drop)

        # 考虑重力和触地约束
        t_flight = t_burst - t_drop
        v0_x = v * np.cos(theta)
        v0_y = v * np.sin(theta)
        # 计算触地时间（从投放点开始）
        # z(t) = P_drop[2] - 0.5 * g * t² = 0
        # 解得: t_ground = sqrt(2 * P_drop[2] / g)
        t_ground = np.sqrt(2 * P_drop[2] / g) if P_drop[2] > 0 else 0
    if t_flight <= t_ground:
        x_burst = P_drop[0] + v0_x * t_flight
        y_burst = P_drop[1] + v0_y * t_flight
        z_burst = P_drop[2] - 0.5 * g * t_flight**2
    else:
        x_burst = P_drop[0] + v0_x * t_ground
        y_burst = P_drop[1] + v0_y * t_ground
        z_burst = 0.0  # 触地点
        P_burst = np.array([x_burst, y_burst, z_burst])
    return P_drop, P_burst, t_burst

```

```
505     def calculate_joint_shelter_duration_3drones_vectorized(particles: np.ndarray)
            -> np.ndarray:
506         #向量化计算多个粒子的联合遮蔽时长
507         Args:
508         particles: (N, 12) 数组，每行包含 [v1, theta1, t1, tb1, v2, theta2, t2, tb2,
                v3, theta3, t3, tb3]
509         Returns:
510         fitness: (N,) 数组，每个粒子的联合遮蔽时长
511         if particles.ndim == 1:
512             particles = particles.reshape(1, -1)
513             N = particles.shape[0]
514             fitness = np.zeros(N)
515         # 预计算目标采样点
516         target_samples_arr = get_target_samples_array()
517         num_targets = target_samples_arr.shape[0]
518         # 预计算导弹方向向量
519         dir_vec = -P_M1_initial
520         dir_unit = dir_vec / np.linalg.norm(dir_vec)
521         batch_size = min(32, N)  # 批处理大小，避免内存过大
522         for batch_start in range(0, N, batch_size):
523             batch_end = min(batch_start + batch_size, N)
524             batch_particles = particles[batch_start:batch_end]
525             batch_size_actual = batch_end - batch_start
526         # 解析批次粒子参数
527         v1_batch = batch_particles[:, 0]
528         theta1_batch = batch_particles[:, 1]
529         t1_batch = batch_particles[:, 2]
530         tb1_batch = batch_particles[:, 3]
531         v2_batch = batch_particles[:, 4]
532         theta2_batch = batch_particles[:, 5]
533         t2_batch = batch_particles[:, 6]
534         tb2_batch = batch_particles[:, 7]
535         v3_batch = batch_particles[:, 8]
536         theta3_batch = batch_particles[:, 9]
537         t3_batch = batch_particles[:, 10]
538         tb3_batch = batch_particles[:, 11]
539         # FY1
540         P_drop1_batch = P_FY1_initial[None, :] + v1_batch[:, None] *
                np.array([np.cos(theta1_batch), np.sin(theta1_batch),
                np.zeros_like(theta1_batch)]).T * t1_batch[:, None]
541         P_burst1_batch = P_drop1_batch.copy()
542         P_burst1_batch[:, 0] += v1_batch * np.cos(theta1_batch) * tb1_batch
543         P_burst1_batch[:, 1] += v1_batch * np.sin(theta1_batch) * tb1_batch
544         P_burst1_batch[:, 2] = np.maximum(P_drop1_batch[:, 2] - 0.5 * g *
                tb1_batch**2, 0.0)
545         t_burst1_batch = t1_batch + tb1_batch
546         # FY2
```

```
        P_drop2_batch = P_FY2_initial[None, :] + v2_batch[:, None] *
            np.array([np.cos(theta2_batch), np.sin(theta2_batch),
            np.zeros_like(theta2_batch)]).T * t2_batch[:, None]
        P_burst2_batch = P_drop2_batch.copy()
        P_burst2_batch[:, 0] += v2_batch * np.cos(theta2_batch) * tb2_batch
        P_burst2_batch[:, 1] += v2_batch * np.sin(theta2_batch) * tb2_batch
        P_burst2_batch[:, 2] = np.maximum(P_drop2_batch[:, 2] - 0.5 * g *
            tb2_batch**2, 0.0)
        t_burst2_batch = t2_batch + tb2_batch
        # FY3
        P_drop3_batch = P_FY3_initial[None, :] + v3_batch[:, None] *
            np.array([np.cos(theta3_batch), np.sin(theta3_batch),
            np.zeros_like(theta3_batch)]).T * t3_batch[:, None]
        P_burst3_batch = P_drop3_batch.copy()
        P_burst3_batch[:, 0] += v3_batch * np.cos(theta3_batch) * tb3_batch
        P_burst3_batch[:, 1] += v3_batch * np.sin(theta3_batch) * tb3_batch
        P_burst3_batch[:, 2] = np.maximum(P_drop3_batch[:, 2] - 0.5 * g *
            tb3_batch**2, 0.0)
        t_burst3_batch = t3_batch + tb3_batch
        time_step = 0.05
        start_times = np.minimum(np.minimum(t_burst1_batch, t_burst2_batch),
            t_burst3_batch)
        end_times = np.minimum(np.maximum(np.maximum(t_burst1_batch +
            smoke_valid_time,
        t_burst2_batch + smoke_valid_time),
        t_burst3_batch + smoke_valid_time), M1_HIT_TIME)
        for i in range(batch_size_actual):
        if start_times[i] >= end_times[i]:
        fitness[batch_start + i] = 0.0
        time_points = np.arange(start_times[i], end_times[i] + time_step, time_step)
        T = len(time_points)
        # 计算导弹位置
        missile_positions = P_M1_initial[None, :] + v_M1 * dir_unit[None, :] *
            time_points[:, None]  # (T, 3)
        smoke_centers_list = []
        valid_masks_list = []
        for P_burst, t_burst in [(P_burst1_batch[i], t_burst1_batch[i]),
        (P_burst2_batch[i], t_burst2_batch[i]),
        (P_burst3_batch[i], t_burst3_batch[i])]:
        smoke_times = time_points - t_burst
        valid_mask = (smoke_times >= 0) & (smoke_times <= smoke_valid_time)
        smoke_centers = np.zeros((T, 3))
        smoke_centers[valid_mask, 0] = P_burst[0]
        smoke_centers[valid_mask, 1] = P_burst[1]
        smoke_centers[valid_mask, 2] = np.maximum(P_burst[2] - v_smoke_sink *
            smoke_times[valid_mask], 0.0)
        smoke_centers_list.append(smoke_centers)
```

```
584         valid_masks_list.append(valid_mask)
585         # 向量化遮蔽检测
586         total_shelter_time = 0.0
587         for t_idx in range(T):
588             missile_pos = missile_positions[t_idx]
589         is_occluded = False
590         for smoke_centers, valid_mask in zip(smoke_centers_list, valid_masks_list):
591         if not valid_mask[t_idx]:
592             continue
593             smoke_center = smoke_centers[t_idx]
594             A = missile_pos
595             Qs = target_samples_arr
596             AB = Qs - A
597             len2_AB = np.sum(AB * AB, axis=1)
598             mask = len2_AB > 1e-12
599         if not np.any(mask):
600             continue
601             AC = (smoke_center - A)[None, :]
602             t_proj = np.zeros_like(len2_AB)
603             t_proj[mask] = np.sum(AB[mask] * AC, axis=1) / len2_AB[mask]
604             t_clamped = np.clip(t_proj, 0.0, 1.0)
605             P_near = A[None, :] + t_clamped[:, None] * AB
606             dist2 = np.sum((P_near - smoke_center[None, :]) ** 2, axis=1)
607             hit = dist2 <= (smoke_radius * smoke_radius + 1e-8)
608             hit = np.logical_and(hit, P_near[:, 2] >= 0.0)
609         if np.any(hit):
610             is_occluded = True
611             break
612         if is_occluded:
613             total_shelter_time += time_step
614             fitness[batch_start + i] = total_shelter_time
615         return fitness if N > 1 else fitness[0]
616     def batch_fitness_calculation(particles: np.ndarray, batch_size: int = 32) ->
        np.ndarray:
617         N = particles.shape[0]
618         if N <= batch_size:
619         return calculate_joint_shelter_duration_3drones_vectorized(particles)
620         fitness = np.zeros(N)
621         for i in range(0, N, batch_size):
622         end_idx = min(i + batch_size, N)
623         batch_particles = particles[i:end_idx]
624         fitness[i:end_idx] =
            calculate_joint_shelter_duration_3drones_vectorized(batch_particles)
625         return fitness
626
627     def _process_particle_chunk(chunk_data: Tuple[np.ndarray, int, int]) ->
        Tuple[np.ndarray, int, int]:
```

```
628         chunk_particles, start_idx, end_idx = chunk_data
629         chunk_fitness =
                calculate_joint_shelter_duration_3drones_vectorized(chunk_particles)
630         return chunk_fitness, start_idx, end_idx
631
632     def parallel_fitness_calculation(particles: np.ndarray,
633         batch_size: int = 32,
634         n_workers: int = None,
635         use_threads: bool = True) -> np.ndarray:
636         #并行计算适应度，支持多线程和多进程
637         N = particles.shape[0]
638         if N <= batch_size:
639             return calculate_joint_shelter_duration_3drones_vectorized(particles)
640         if n_workers is None:
641             n_workers = min(mp.cpu_count(), 8)  # 最多使用8个核心
642             chunks = []
643         for i in range(0, N, batch_size):
644             end_idx = min(i + batch_size, N)
645             chunk_particles = particles[i:end_idx]
646             chunks.append((chunk_particles, i, end_idx))
647             fitness = np.zeros(N)
648         if use_threads:
649             # 使用多线程（适合I/O密集型任务）
650             with ThreadPoolExecutor(max_workers=n_workers) as executor:
651             results = list(executor.map(_process_particle_chunk, chunks))
652         else:
653             with ProcessPoolExecutor(max_workers=n_workers) as executor:
654             results = list(executor.map(_process_particle_chunk, chunks))
655             for chunk_fitness, start_idx, end_idx in results:
656             fitness[start_idx:end_idx] = chunk_fitness
657         return fitness
658
659     def adaptive_parallel_fitness_calculation(particles: np.ndarray,
660         batch_size: int = 32,
661         n_workers: int = None) -> np.ndarray:
662         #自适应并行计算适应度，根据数据规模选择最优策略
663         N = particles.shape[0]
664         if N <= batch_size:
665             return calculate_joint_shelter_duration_3drones_vectorized(particles)
666         elif N <= batch_size * 4:
667             return parallel_fitness_calculation(particles, batch_size, n_workers,
                    use_threads=True)
668         else:
669             return parallel_fitness_calculation(particles, batch_size, n_workers,
                    use_threads=False)
670
671         def compute_fitness_with_config(particles: np.ndarray) -> np.ndarray:
```

```
        if PSO_PARAMS['use_parallel']:
            return adaptive_parallel_fitness_calculation(
            particles,
            batch_size=PSO_PARAMS['parallel_batch_size'],
            n_workers=PSO_PARAMS['parallel_workers']
        )
        else:
            return batch_fitness_calculation(particles,
                batch_size=PSO_PARAMS['parallel_batch_size'])

def benchmark_parallel_performance(n_particles: int = 100, n_trials: int = 3)
    -> Dict[str, float]:
    test_particles = np.random.rand(n_particles, 12)
    for i in range(12):
    low, high = OPT_BOUNDS[i]
    test_particles[:, i] = low + test_particles[:, i] * (high - low)
    serial_times = []
    for _ in range(n_trials):
    start_time = time.time()
    _ = batch_fitness_calculation(test_particles, batch_size=32)
    serial_times.append(time.time() - start_time)
    parallel_times = []
    for _ in range(n_trials):
    start_time = time.time()
    _ = adaptive_parallel_fitness_calculation(test_particles, batch_size=64)
    parallel_times.append(time.time() - start_time)
    avg_serial_time = np.mean(serial_times)
    avg_parallel_time = np.mean(parallel_times)
    speedup = avg_serial_time / avg_parallel_time
        results = {
        'serial_time': avg_serial_time,
        'parallel_time': avg_parallel_time,
        'speedup': speedup,
        'efficiency': speedup / min(mp.cpu_count(), 8) * 100  }
        return results

def calculate_joint_shelter_duration_3drones(v1: float, theta1: float, t1:
    float, tb1: float,
    v2: float, theta2: float, t2: float, tb2: float,
    v3: float, theta3: float, t3: float, tb3: float) -> float:
    # 获取起爆参数
    P_drop1, P_burst1, t_burst1 = get_bomb_burst_params_3drones(P_FY1_initial,
        v1, theta1, t1, t1 + tb1)
    P_drop2, P_burst2, t_burst2 = get_bomb_burst_params_3drones(P_FY2_initial,
        v2, theta2, t2, t2 + tb2)
    P_drop3, P_burst3, t_burst3 = get_bomb_burst_params_3drones(P_FY3_initial,
        v3, theta3, t3, t3 + tb3)
```

```
        # 从最早起爆到三弹有效期并集末端，且不超过导弹命中
        time_step = 0.05
        start_time = min(t_burst1, t_burst2, t_burst3)
        end_time = min(max(t_burst1 + smoke_valid_time, t_burst2 + smoke_valid_time,
            t_burst3 + smoke_valid_time), M1_HIT_TIME)
        if start_time >= end_time:
        return 0.0
        time_points = np.arange(start_time, end_time + time_step, time_step)
        # 获取目标采样点）
        target_samples_arr = get_target_samples_array()
        total_shelter_time = 0.0
        prev_eff: bool | None = None
        for idx, t in enumerate(time_points):
        # 计算导弹位置（导弹从初始位置飞向原点）
            dir_vec = -P_M1_initial  # 指向假目标（原点）的方向向量
            dir_unit = dir_vec / np.linalg.norm(dir_vec)  # 单位向量
            missile_pos = P_M1_initial + v_M1 * dir_unit * t
        # OR口径：任一采样点被任一烟幕遮蔽即有效
        A = missile_pos  # (3,)
        Qs = target_samples_arr  # (N,3)
        AB = Qs - A  # (N,3)
        len2_AB = np.sum(AB * AB, axis=1)
        mask = len2_AB > 1e-12

    def check_occluded(P_burst: np.ndarray, t_burst: float) -> bool:
        if not (t_burst <= t <= t_burst + smoke_valid_time):
            return False
        smoke_z = max(P_burst[2] - v_smoke_sink * (t - t_burst), 0.0)
        C = np.array([P_burst[0], P_burst[1], smoke_z])
        R = smoke_radius
        AC = (C - A)[None, :]
        t_proj = np.zeros_like(len2_AB)
        t_proj[mask] = np.sum(AB[mask] * AC, axis=1) / len2_AB[mask]
        t_clamped = np.clip(t_proj, 0.0, 1.0)
        P_near = A[None, :] + t_clamped[:, None] * AB
        dist2 = np.sum((P_near - C[None, :]) ** 2, axis=1)
        hit = dist2 <= (R * R + 1e-8)
        # 半空间裁剪：仅地面以上的最近点有效（开关可扩展）
        hit = np.logical_and(hit, P_near[:, 2] >= 0.0)
        return np.any(hit)
        eff = (check_occluded(P_burst1, t_burst1) or
        check_occluded(P_burst2, t_burst2) or
        check_occluded(P_burst3, t_burst3))
        if eff:
            total_shelter_time += time_step
        # 边界二分细化：当相邻采样点有效性发生变化时进行边界定位
        if idx > 0 and prev_eff is not None and (eff != prev_eff):
```

```
            lo = time_points[idx - 1]
            hi = t
        for _ in range(5):  # 细化次数
            mid = 0.5 * (lo + hi)
        # 重新计算导弹遮蔽状态
        A_mid = P_M1_initial + v_M1 * dir_unit * mid
        Qs_mid = target_samples_arr
        AB_mid = Qs_mid - A_mid
        len2_mid = np.sum(AB_mid * AB_mid, axis=1)
        mask_mid = len2_mid > 1e-12
        def occluded_mid(P_burst: np.ndarray, t_burst: float) -> bool:
        if not (t_burst <= mid <= t_burst + smoke_valid_time):
        return False
        smoke_z = max(P_burst[2] - v_smoke_sink * (mid - t_burst), 0.0)
        C = np.array([P_burst[0], P_burst[1], smoke_z])
        R = smoke_radius
        ACm = (C - A_mid)[None, :]
        tproj = np.zeros_like(len2_mid)
        tproj[mask_mid] = np.sum(AB_mid[mask_mid] * ACm, axis=1) / len2_mid[mask_mid]
        tcl = np.clip(tproj, 0.0, 1.0)
        Pn = A_mid[None, :] + tcl[:, None] * AB_mid
        d2 = np.sum((Pn - C[None, :]) ** 2, axis=1)
        hit = d2 <= (R * R + 1e-8)
        hit = np.logical_and(hit, Pn[:, 2] >= 0.0)
        return np.any(hit)
        cur = (occluded_mid(P_burst1, t_burst1) or
        occluded_mid(P_burst2, t_burst2) or
        occluded_mid(P_burst3, t_burst3))
        if cur == prev_eff:
            lo = mid
        else:
            hi = mid
            prev_eff = eff
        return total_shelter_time
    def local_search_optimization(particle: np.ndarray) -> np.ndarray:
        def objective(x):
        return -calculate_joint_shelter_duration_3drones(*x)
        # 使用L-BFGS-B方法进行局部搜索
        result = minimize(objective, particle, method='L-BFGS-B', bounds=OPT_BOUNDS,
        options={'maxiter': PSO_PARAMS['local_search_iter']})
        if result.success:
            return repair_particle_3drones(result.x)
        else:
            return particle

    def hybrid_pso_local_search_ultra_enhanced() -> Tuple[Dict[str, float],
        List[Tuple[int, float, float]]]:
```

```
    if PSO_PARAMS['use_parallel']:
        n_workers = PSO_PARAMS['parallel_workers'] or min(mp.cpu_count(), 8)
    # 初始化种群
    pop_size = PSO_PARAMS['pop_size']
    dim = 12
    # 在归一化空间[0,1]^12中初始化粒子位置
    particles = np.random.rand(pop_size, dim)
    velocities = np.random.randn(pop_size, dim) * 0.1
    # 初始化个体最优和全局最优
    personal_best = particles.copy()
    # 批量计算初始适应度
    denormalized_particles = np.array([denormalize_particle(p) for p in
        particles])
    personal_best_fitness = compute_fitness_with_config(denormalized_particles)
    global_best_idx = np.argmax(personal_best_fitness)
    global_best = personal_best[global_best_idx].copy()
    global_best_fitness = personal_best_fitness[global_best_idx]
    # 初始化策略组件
    tabu_search = TabuSearch(ENHANCED_PSO['tabu_tenure'],
        ENHANCED_PSO['tabu_list_size'])
    simulated_annealing = SimulatedAnnealing(ENHANCED_PSO['sa_initial_temp'],
        ENHANCED_PSO['sa_cooling_rate'])
    cultural_algorithm = CulturalAlgorithm()
    optimization_history = []
    no_improvement_count = 0
    last_restart_iter = 0
    # PSO主循环（集成所有高级策略）
    for iteration in range(PSO_PARAMS['max_iter']):
    # 计算惯性权重
    w = PSO_PARAMS['w_start'] - (PSO_PARAMS['w_start'] - PSO_PARAMS['w_end']) *
        iteration / PSO_PARAMS['max_iter']
    if ENHANCED_PSO['use_time_varying']:
        ratio = iteration / max(1, PSO_PARAMS['max_iter'])
        c1_tv = ENHANCED_PSO['c1_start'] + (ENHANCED_PSO['c1_end'] -
            ENHANCED_PSO['c1_start']) * ratio
        c2_tv = ENHANCED_PSO['c2_start'] + (ENHANCED_PSO['c2_end'] -
            ENHANCED_PSO['c2_start']) * ratio
    else:
        c1_tv, c2_tv = PSO_PARAMS['c1'], PSO_PARAMS['c2']
    generation_improved = False
    avg_fitness = np.mean(personal_best_fitness)
    idx_left = (np.arange(pop_size) - 1) % pop_size
    idx_right = (np.arange(pop_size) + 1) % pop_size
    for i in range(pop_size):
    # lbest邻域选择
        lbest_idx = i
    if personal_best_fitness[idx_left[i]] > personal_best_fitness[lbest_idx]:
```

```
            lbest_idx = idx_left[i]
        if personal_best_fitness[idx_right[i]] > personal_best_fitness[lbest_idx]:
            lbest_idx = idx_right[i]
            social_target = personal_best[lbest_idx] if (np.random.rand() <
                PSO_PARAMS['lbest_prob']) else global_best
        # 标准PSO更新
        r1, r2 = np.random.random(2)
        velocities[i] = (w * velocities[i] +
        c1_tv * r1 * (personal_best[i] - particles[i]) +
        c2_tv * r2 * (social_target - particles[i]))
        vel_cap = np.full(dim, 0.1)
        velocities[i] = np.clip(velocities[i], -vel_cap, vel_cap)
        particles[i] += velocities[i]
        particles[i] = np.clip(particles[i], 0.0, 1.0)
        particles[i] = repair_particle_3drones(particles[i], is_normalized=True)
        original_particle = particles[i].copy()
        if ENHANCED_PSO['use_chaos_mutation']:
            particles[i] = chaos_mutation(particles[i], iteration,
                PSO_PARAMS['max_iter'])
        # 莱维飞行
        if ENHANCED_PSO['use_levy_flight']:
            particles[i] = levy_flight(particles[i], global_best)
        # 量子行为
        if ENHANCED_PSO['use_quantum_behavior']:
            particles[i] = quantum_behavior(particles[i], personal_best[i],
                global_best)
        # 自适应变异
        if ENHANCED_PSO['use_adaptive_mutation']:
            particles[i] = adaptive_mutation(particles[i], personal_best_fitness[i],
                avg_fitness, iteration, PSO_PARAMS['max_iter'])
        # 禁忌搜索检查
        if ENHANCED_PSO['use_tabu_search'] and tabu_search.is_tabu(particles[i]):
            particles[i] = original_particle  # 恢复原粒子
        # 计算适应度
        denormalized_particle = denormalize_particle(particles[i])
        fitness = compute_fitness_with_config(denormalized_particle.reshape(1, -1))
        if np.isscalar(fitness):
            fitness = float(fitness)
        else:
            fitness = fitness[0]
        # 模拟退火接受机制
        if ENHANCED_PSO['use_simulated_annealing']:
        if not simulated_annealing.accept_solution(personal_best_fitness[i],
            fitness):
            particles[i] = original_particle
        fitness = personal_best_fitness[i]
        if fitness > personal_best_fitness[i]:
```

```
            personal_best[i] = particles[i].copy()
            personal_best_fitness[i] = fitness
            generation_improved = True
        # 添加到禁忌列表
        if ENHANCED_PSO['use_tabu_search']:
            tabu_search.add_to_tabu(particles[i])
        # 更新全局最优
        if fitness > global_best_fitness:
            global_best = particles[i].copy()
            global_best_fitness = fitness
        # 更新高级策略组件
            tabu_search.update_tabu()
            simulated_annealing.cool_down()
            cultural_algorithm.update_belief_space(particles, personal_best_fitness)
        # 7. 文化算法影响
        if ENHANCED_PSO['use_cultural_algorithm']:
            particles = cultural_algorithm.influence_particles(particles)
        #精英对立学习
        if ENHANCED_PSO['use_elite_opposition']:
            particles = elite_opposition_learning(particles, personal_best_fitness)
        if ENHANCED_PSO['use_multi_swarm']:
            particles = multi_swarm_optimization(particles, personal_best_fitness)
        if np.random.random() < PSO_PARAMS['local_search_prob']:
            idx = np.random.randint(0, pop_size)
            denormalized_particle = denormalize_particle(particles[idx])
            improved_particle = local_search_optimization(denormalized_particle)
            improved_particle_normalized = normalize_particle(improved_particle)
        improved_fitness = compute_fitness_with_config(improved_particle.reshape(1,
            -1))
        if np.isscalar(improved_fitness):
            improved_fitness = float(improved_fitness)
        else:
            improved_fitness = improved_fitness[0]
        if improved_fitness > personal_best_fitness[idx]:
            particles[idx] = improved_particle_normalized
            personal_best[idx] = improved_particle_normalized
            personal_best_fitness[idx] = improved_fitness
            generation_improved = True
        if improved_fitness > global_best_fitness:
            global_best = improved_particle_normalized
            global_best_fitness = improved_fitness
        # 遗传增强
        if (iteration + 1) % PSO_PARAMS['genetic_interval'] == 0:
            diversity = calculate_diversity(particles)
        if (diversity < PSO_PARAMS['diversity_threshold']) or (no_improvement_count
            >= 10):
```

```
            denormalized_particles = np.array([denormalize_particle(p) for p in
                particles])
            current_fitness = compute_fitness_with_config(denormalized_particles)
            enhanced_particles = genetic_enhancement(denormalized_particles,
                current_fitness, iteration)
            particles = np.array([normalize_particle(p) for p in enhanced_particles])
        # 重新计算适应度
        denormalized_particles = np.array([denormalize_particle(p) for p in
            particles])
        new_fitness = compute_fitness_with_config(denormalized_particles)
        for i in range(pop_size):
            if new_fitness[i] > personal_best_fitness[i]:
                personal_best[i] = particles[i].copy()
                personal_best_fitness[i] = new_fitness[i]
            generation_improved = True
            if new_fitness[i] > global_best_fitness:
                global_best = particles[i].copy()
                global_best_fitness = new_fitness[i]

        # 移民与抖动
        if (iteration + 1) % ENHANCED_PSO['immigrant_period'] == 0:
            k = max(1, int(pop_size * ENHANCED_PSO['immigrant_ratio']))
            worst_idx = np.argsort(personal_best_fitness)[:k]
        for wi in worst_idx:
            particles[wi] = np.random.rand(dim)
            particles[wi] = repair_particle_3drones(particles[wi], is_normalized=True)
            personal_best[wi] = particles[wi].copy()
            denormalized_particle = denormalize_particle(particles[wi])
            fitness_wi = compute_fitness_with_config(denormalized_particle.reshape(1,
                -1))
        if np.isscalar(fitness_wi):
        personal_best_fitness[wi] = float(fitness_wi)
        else:
            personal_best_fitness[wi] = fitness_wi[0]
            generation_improved = True
        if (iteration + 1) % ENHANCED_PSO['jitter_period'] == 0:
        velocities += np.random.randn(*velocities.shape) *
            ENHANCED_PSO['jitter_sigma']
        # 多样性监控和种群重启
        diversity = calculate_diversity(particles)
        if ((diversity < PSO_PARAMS['diversity_threshold']) and
        (iteration - last_restart_iter > PSO_PARAMS['restart_iterations']) or
        (no_improvement_count >= PSO_PARAMS['restart_patience'])):
        # 精英保留重启
        if ENHANCED_PSO['use_restart_with_elite']:
            elite_count = max(1, int(pop_size *
                ENHANCED_PSO['elite_retention_ratio']))
```

```
            elite_indices = np.argsort(personal_best_fitness)[-elite_count:]
            elite_particles = particles[elite_indices]
            # 重新初始化其余粒子
        new_particles = elite_particles.copy()
        for _ in range(pop_size - elite_count):
            new_particle = np.random.rand(dim)
            new_particle = repair_particle_3drones(new_particle, is_normalized=True)
            new_particles = np.vstack([new_particles, new_particle])

            particles = new_particles
        else:
            denormalized_particles = np.array([denormalize_particle(p) for p in
                particles])
            restarted_particles = population_restart(denormalized_particles,
                personal_best_fitness)
            particles = np.array([normalize_particle(p) for p in restarted_particles])
                # 重新计算适应度
            denormalized_particles = np.array([denormalize_particle(p) for p in
                particles])
            personal_best_fitness =
                compute_fitness_with_config(denormalized_particles)
            last_restart_iter = iteration
            generation_improved = True
            no_improvement_count = 0
        if generation_improved:
            no_improvement_count = 0
        else:
            no_improvement_count += 1
            # 早停机制
        if no_improvement_count > PSO_PARAMS['early_stop_patience']:
            print(f"第{iteration+1}代: 早停 (连续{no_improvement_count}代无改进)")
        break
            # 记录历史
        avg_fitness = np.mean(personal_best_fitness)
        optimization_history.append((iteration + 1, global_best_fitness,
            avg_fitness))
            # 每10代输出一次
        if (iteration + 1) % 10 == 0:
            print(f"第{iteration+1}代: 最优解={global_best_fitness:.4f}s, 平均解={avg_fitness:.4f}s,
            # 构建结果
        global_best_denormalized = denormalize_particle(global_best)
        v1_opt, theta1_opt, t1_opt, tb1_opt, v2_opt, theta2_opt, t2_opt, tb2_opt,
            v3_opt, theta3_opt, t3_opt, tb3_opt = global_best_denormalized
            # 计算投放点和起爆点位置
            P_drop1, P_burst1, t_burst1 =
                get_bomb_burst_params_3drones(P_FY1_initial, v1_opt, theta1_opt,
                t1_opt, t1_opt + tb1_opt)
```

```
        P_drop2, P_burst2, t_burst2 =
            get_bomb_burst_params_3drones(P_FY2_initial, v2_opt, theta2_opt,
            t2_opt, t2_opt + tb2_opt)
        P_drop3, P_burst3, t_burst3 =
            get_bomb_burst_params_3drones(P_FY3_initial, v3_opt, theta3_opt,
            t3_opt, t3_opt + tb3_opt)
        result = {
            'v1': v1_opt, 'theta1': theta1_opt, 't_drop1': t1_opt, 't_burst1':
                tb1_opt,
            'v2': v2_opt, 'theta2': theta2_opt, 't_drop2': t2_opt, 't_burst2':
                tb2_opt,
            'v3': v3_opt, 'theta3': theta3_opt, 't_drop3': t3_opt, 't_burst3':
                tb3_opt,
            'P_drop1': P_drop1, 'P_burst1': P_burst1,
            'P_drop2': P_drop2, 'P_burst2': P_burst2,
            'P_drop3': P_drop3, 'P_burst3': P_burst3,
            'shelter_duration': global_best_fitness   }

    return result, optimization_history
    def _q4_run_segment_combo(v1_range: Tuple[float, float],
        v2_range: Tuple[float, float],
        v3_range: Tuple[float, float]) -> Dict[str, Any]:
        #在给定三架速度分段范围内运行一次混合PSO。
        orig_v1, orig_v2, orig_v3 = OPT_BOUNDS[0], OPT_BOUNDS[4], OPT_BOUNDS[8]
        try:
            OPT_BOUNDS[0] = v1_range
            OPT_BOUNDS[4] = v2_range
            OPT_BOUNDS[8] = v3_range
        result, _ = hybrid_pso_local_search_ultra_enhanced()
        # 附带记录使用的分段
        result['segments'] = {'v1': v1_range, 'v2': v2_range, 'v3': v3_range}
        return result
        finally:
            OPT_BOUNDS[0] = orig_v1
            OPT_BOUNDS[4] = orig_v2
            OPT_BOUNDS[8] = orig_v3

    def segmented_optimize_q4(
        v1_segments: List[Tuple[float, float]] = [(70, 92), (88, 112), (108, 140)],
        v2_segments: List[Tuple[float, float]] = [(70, 92), (88, 112), (108, 140)],
        v3_segments: List[Tuple[float, float]] = [(70, 92), (88, 112), (108, 140)],
        mode: str = 'full',  # 'full' | 'topk' | 'rotate'
        top_k: int = 2,
        max_workers: int | None = None,
        rotate_rounds: int = 1,
        ) -> Dict[str, Any]:
        #三无人机速度分段并行优化。
```

```
    results: Dict[str, Any] = {'mode': mode, 'combos': []}
    global_best: Dict[str, Any] = None

def submit_combos(combos: List[Tuple[Tuple[float, float], Tuple[float, float],
    Tuple[float, float]]]) -> None:
        nonlocal global_best
        workers = max_workers or max(1, os.cpu_count() or 1)
        with concurrent.futures.ProcessPoolExecutor(max_workers=workers) as ex:
        futs = [ex.submit(_q4_run_segment_combo, c[0], c[1], c[2]) for c in
            combos]
        for fut in concurrent.futures.as_completed(futs):
    try:
        res = fut.result()
        results['combos'].append(res)
    if (global_best is None) or (res['shelter_duration'] >
        global_best['shelter_duration']):
        global_best = res
        except Exception as e:
        if mode == 'full':
    # 全组合
    combos = []
        for s1 in v1_segments:
            for s2 in v2_segments:
                for s3 in v3_segments:
                    combos.append((s1, s2, s3))
                    print(f"分段(full): 共 {len(combos)} 组")
                    submit_combos(combos)
        elif mode == 'topk':
    # 改进的TopK筛选: 考虑协同效应
def select_topk_safe(segs1: List[Tuple[float, float]],
        segs2: List[Tuple[float, float]],
        segs3: List[Tuple[float, float]]) -> List[Tuple[Tuple[float, float],
            Tuple[float, float], Tuple[float, float]]]:
    #评估所有组合, 避免筛掉最优解
    all_combos = []
    scores = []
    # 评估所有分段组合
    for i, s1 in enumerate(segs1):
        for j, s2 in enumerate(segs2):
            for k, s3 in enumerate(segs3):
                combo = (s1, s2, s3)
        print(f"  评估组合 {i+1}-{j+1}-{k+1}: v1{s1}, v2{s2}, v3{s3}")
        res = _q4_run_segment_combo(s1, s2, s3)
        all_combos.append(combo)
        scores.append(res['shelter_duration'])
        print(f"    结果: {res['shelter_duration']:.4f}s")
        # 按分数排序, 选择Top-K组合
```

```
            sorted_indices = np.argsort(scores)[::-1]  # 降序
            top_combos = [all_combos[i] for i in sorted_indices[:max(1, top_k)]]
        for i, combo in enumerate(top_combos):
            score = scores[sorted_indices[i]]
            print(f"  Top-{i+1}: v1{combo[0]}, v2{combo[1]}, v3{combo[2]} =
                {score:.4f}s")
        return top_combos
        # 使用安全筛选
        top_combos = select_topk_safe(v1_segments, v2_segments, v3_segments)
        print(f"分段(topk): 安全筛选出 {len(top_combos)} 个最优组合")
        submit_combos(top_combos)
        else:  # rotate
            sel_v1, sel_v2, sel_v3 = None, None, None
        for r in range(max(1, rotate_rounds)):
        # 优化 v1 段
        best_pair = None
        for seg in v1_segments:
            v1r = seg
            v2r = sel_v2 if sel_v2 else OPT_BOUNDS[4]
            v3r = sel_v3 if sel_v3 else OPT_BOUNDS[8]
            res = _q4_run_segment_combo(v1r, v2r, v3r)
        if (best_pair is None) or (res['shelter_duration'] > best_pair[0]):
            best_pair = (res['shelter_duration'], seg)
            sel_v1 = best_pair[1]
        # 优化 v2 段
            best_pair = None
        for seg in v2_segments:
            v1r = sel_v1 if sel_v1 else OPT_BOUNDS[0]
            v2r = seg
            v3r = sel_v3 if sel_v3 else OPT_BOUNDS[8]
            res = _q4_run_segment_combo(v1r, v2r, v3r)
        if (best_pair is None) or (res['shelter_duration'] > best_pair[0]):
            best_pair = (res['shelter_duration'], seg)
            sel_v2 = best_pair[1]
        # 优化 v3 段
            best_pair = None
        for seg in v3_segments:
            v1r = sel_v1 if sel_v1 else OPT_BOUNDS[0]
            v2r = sel_v2 if sel_v2 else OPT_BOUNDS[4]
            v3r = seg
            res = _q4_run_segment_combo(v1r, v2r, v3r)
        if (best_pair is None) or (res['shelter_duration'] > best_pair[0]):
            best_pair = (res['shelter_duration'], seg)
            sel_v3 = best_pair[1]
        # 用选择的段组合精跑一次
        final_res = _q4_run_segment_combo(sel_v1 if isinstance(sel_v1, tuple) else
            OPT_BOUNDS[0],
```

```
        sel_v2 if isinstance(sel_v2, tuple) else OPT_BOUNDS[4],
        sel_v3 if isinstance(sel_v3, tuple) else OPT_BOUNDS[8])
        results['combos'].append(final_res)
        global_best = final_res

        results['global_best'] = global_best
        print("分段并行完成。总体最优联合遮蔽=", global_best['shelter_duration'])
        return results

    def save_results_to_excel(optimal_params: Dict[str, float]) -> None:
            result_data = []
            # 为3架无人机创建数据行
        for drone_id in range(3):
            drone_key = f'FY{drone_id + 1}'
            v_key = f'v{drone_id + 1}'
            theta_key = f'theta{drone_id + 1}'
            t_drop_key = f't_drop{drone_id + 1}'
            t_burst_key = f't_burst{drone_id + 1}'
            # 计算运动方向
            direction = np.degrees(optimal_params[theta_key])
        if direction < 0:
            direction += 360
                result_data.append({
                '无人机编号': drone_key,
                '无人机运动方向': f"{direction:.2f}",
                '无人机运动速度 (m/s)': f"{optimal_params[v_key]:.2f}",
                '烟幕干扰弹投放点的x坐标 (m)': f"{optimal_params[f'P_drop{drone_id +
                    1}'][0]:.2f}",
                '烟幕干扰弹投放点的y坐标 (m)': f"{optimal_params[f'P_drop{drone_id +
                    1}'][1]:.2f}",
                '烟幕干扰弹投放点的z坐标 (m)': f"{optimal_params[f'P_drop{drone_id +
                    1}'][2]:.2f}",
                '烟幕干扰弹起爆点的x坐标 (m)': f"{optimal_params[f'P_burst{drone_id +
                    1}'][0]:.2f}",
                '烟幕干扰弹起爆点的y坐标 (m)': f"{optimal_params[f'P_burst{drone_id +
                    1}'][1]:.2f}",
                '烟幕干扰弹起爆点的z坐标 (m)': f"{optimal_params[f'P_burst{drone_id +
                    1}'][2]:.2f}",
                '有效干扰时长 (s)': f"{optimal_params['shelter_duration']:.4f}"})


        df = pd.DataFrame(result_data)
            with pd.ExcelWriter('result2.xlsx', engine='openpyxl') as writer:
        df.to_excel(writer, sheet_name='result2', index=False)
            note_df = pd.DataFrame(note_data)
        note_df.to_excel(writer, sheet_name='result2', startrow=len(df)+2,
            index=False)
```

```
1180
1181 if __name__ == "__main__":
1182     import sys
1183     if len(sys.argv) > 1 and sys.argv[1] == "--benchmark":
1184         print("=" * 60)
1185         print("q4.py 并行计算性能基准测试")
1186         print("=" * 60)
1187         benchmark_parallel_performance(n_particles=100, n_trials=3)
1188         sys.exit(0)
1189         print("=" * 60)
1190         print("问题4: 三段式分段并行优化 (topk → full → 精跑)")
1191         print("=" * 60)
1192     # 计算总组合数, 自动选择策略
1193     v1_segments = [(70, 92), (88, 112), (108, 140)]
1194     v2_segments = [(70, 92), (88, 112), (108, 140)]
1195     v3_segments = [(70, 92), (88, 112), (108, 140)]
1196     total_combos = len(v1_segments) * len(v2_segments) * len(v3_segments)
1197         print(f"分段配置: v1有{len(v1_segments)}段, v2有{len(v2_segments)}段, v3有{len(v3_segments)
1198     print(f"总组合数: {total_combos}个")
1199     if total_combos <= 27:  # 3×3×3 = 27
1200         # 组合数较少, 使用安全TopK筛选
1201         topk_results = segmented_optimize_q4(mode='topk', top_k=2)
1202         best_topk = topk_results['global_best']
1203         seg_v1 = best_topk['segments']['v1']
1204         seg_v2 = best_topk['segments']['v2']
1205         seg_v3 = best_topk['segments']['v3']
1206         print(f"安全Top-K 最优分段: v1{seg_v1}, v2{seg_v2},
                v3{seg_v3}, 联合遮蔽={best_topk['shelter_duration']:.4f}s")
1207     else:
1208         full_results = segmented_optimize_q4(mode='full')
1209         best_full = full_results['global_best']
1210         print(f"全组合最优: 联合遮蔽={best_full['shelter_duration']:.4f}s")
1211         seg_v1 = best_full['segments']['v1']
1212         seg_v2 = best_full['segments']['v2']
1213         seg_v3 = best_full['segments']['v3']
1214
1215     # 在筛选的分段集合上做全组合并行
1216     if total_combos <= 27:
1217         v1_segments_final = [seg_v1]
1218         v2_segments_final = [seg_v2]
1219         v3_segments_final = [seg_v3]
1220         full_results = segmented_optimize_q4(v1_segments=v1_segments_final,
                v2_segments=v2_segments_final, v3_segments=v3_segments_final, mode='full')
1221     best_full = full_results['global_best']
1222     print(f"Full 组合最优: 联合遮蔽={best_full['shelter_duration']:.4f}s")
1223     else:
1224         # 如果第一阶段已经用了全组合, 直接使用第一阶段结果
```

```
        best_full = full_results

    # 第二阶段最优为初值做一次精跑
    print("[阶段3] 精跑局部细化（超强防局部最优版本，提高迭代次数）...")
        old_max_iter = PSO_PARAMS['max_iter']
    try:
        PSO_PARAMS['max_iter'] = max(old_max_iter, 450)
        optimal_params, optimization_history =
            hybrid_pso_local_search_ultra_enhanced()
    finally:
        PSO_PARAMS['max_iter'] = old_max_iter

    # 输出最优结果
    print(f"联合遮蔽时长: {optimal_params['shelter_duration']:.4f}秒")
    print(f"FY1: 速度={optimal_params['v1']:.2f}
        m/s, 方向角={np.degrees(optimal_params['theta1']):.2f}°, 投放时间={optimal_params['t_drop1
        s, 起爆延迟={optimal_params['t_burst1']:.2f} s")
    print(f"FY2: 速度={optimal_params['v2']:.2f}
        m/s, 方向角={np.degrees(optimal_params['theta2']):.2f}°, 投放时间={optimal_params['t_drop2
        s, 起爆延迟={optimal_params['t_burst2']:.2f} s")
    print(f"FY3: 速度={optimal_params['v3']:.2f}
        m/s, 方向角={np.degrees(optimal_params['theta3']):.2f}°, 投放时间={optimal_params['t_drop3
        s, 起爆延迟={optimal_params['t_burst3']:.2f} s")

    # 保存结果
    print("\n保存结果...")
    save_results_to_excel(optimal_params)
    print("完成。")
```

附录 E　自适应双坐标系差分进化算法——问题五的 Python 程序

```
import numpy as np
import pandas as pd
import math
import time

# ---------------------------- 一、核心配置（固定参数，无动态调整）
    ----------------------------
# 1. 目标与载体初始位置（原文件核心坐标，无修改）
TGT = np.array([0.0, 200.0, 0.0])  # 真目标位置
MISSILE_INITS = [  # 3枚导弹初始位置（M1~M3）
np.array([20000.0, 0.0, 2000.0]),
np.array([19000.0, 600.0, 2100.0]),
np.array([18000.0, -600.0, 1900.0]),
]
```

```
UAV_INITS = [  # 5架无人机初始位置（FY1~FY5）
np.array([17800.0, 0.0, 1800.0]),
np.array([12000.0, 1400.0, 1400.0]),
np.array([6000.0, -3000.0, 700.0]),
np.array([11000.0, 2000.0, 1800.0]),
np.array([13000.0, -2000.0, 1300.0]),
]
UAV_NAMES = ["FY1", "FY2", "FY3", "FY4", "FY5"]
MISSILE_NAMES = ["M1", "M2", "M3"]

# 2. 物理常数（原文件核心参数，固定无自适应）
MISSILE_SPEED = 300.0  # 导弹速度(m/s)
R_CLOUD = 10.0  # 烟幕遮蔽判定半径(m)
CLOUD_DURATION = 20.0  # 烟幕有效时间(s)
CLOUD_SINK_V = 3.0  # 烟幕下沉速度(m/s)
G = 9.8  # 重力加速度(m/s²)

# 3. 无人机约束（固定范围，无动态调整）
UAV_SPEED_MIN, UAV_SPEED_MAX = 70.0, 140.0  # 无人机速度范围
NUM_UAV = 5  # 无人机数量
SLOTS_PER_UAV = 3  # 每架无人机投弹槽位数量
TOTAL_SLOTS = NUM_UAV * SLOTS_PER_UAV  # 总投弹槽位（5*3=15）

# 4. 仿真与优化参数（固定，无AI化动态调整）
DT = 0.1  # 仿真时间步长(s)
SIM_T_END = max([np.linalg.norm(p-TGT)/MISSILE_SPEED for p in MISSILE_INITS])
    # 仿真结束时间（最晚导弹命中）
DE_NP = 100  # 差分进化（DE）种群大小
DE_MAX_GEN = 300  # DE迭代次数
DE_F = 0.8  # DE变异缩放因子（固定）
DE_CR = 0.5  # DE交叉概率（固定）
OUTPUT_XLSX = "result3_core.xlsx"  # 结果输出路径

def clip_angle(angle):
"""角度归一到[0, 2π)，确保方向合法"""
return angle % (2 * math.pi)

def missile_pos(missile_idx, t):
"""计算t时刻导弹位置（原文件逻辑：匀速向真目标飞行）"""
init_pos = MISSILE_INITS[missile_idx]
dir_vec = (TGT - init_pos) / np.linalg.norm(TGT - init_pos)  #
    导弹指向真目标的单位向量
return init_pos + MISSILE_SPEED * dir_vec * t

def uav_pos(uav_idx, theta, speed, t):
"""计算t时刻无人机位置（原文件逻辑：等高度匀速飞行）"""
init_pos = UAV_INITS[uav_idx]
```

```
    dir_xy = np.array([math.cos(theta), math.sin(theta), 0.0])  # 水平飞行方向向量
    return init_pos + dir_xy * speed * t

def cloud_blast_pos(uav_idx, theta, speed, t_drop, t_blast_delay):
    """计算烟幕起爆点（原文件逻辑：平抛运动+地面约束，避免钻地）"""
    # 步骤1：计算烟幕投放时的无人机位置（投放点）
    drop_pos = uav_pos(uav_idx, theta, speed, t_drop)
    # 步骤2：烟幕弹平抛运动（水平速度=无人机速度，垂直受重力）
    horizontal_speed = speed * np.array([math.cos(theta), math.sin(theta), 0.0])
    # 触地时间（烟幕弹落地前必须起爆，避免无效）
    t_ground = math.sqrt(2 * drop_pos[2] / G) if drop_pos[2] > 0 else 0.0
    actual_delay = min(t_blast_delay, t_ground)  # 实际起爆延迟（不超过触地时间）
    # 步骤3：计算起爆点坐标
    blast_x = drop_pos[0] + horizontal_speed[0] * actual_delay
    blast_y = drop_pos[1] + horizontal_speed[1] * actual_delay
    blast_z = max(0.0, drop_pos[2] - 0.5 * G * (actual_delay ** 2))  # 垂直位移（不低于地面）
    return np.array([blast_x, blast_y, blast_z]), t_drop + actual_delay  # 返回起爆点+总起爆时间

def cloud_center(blast_pos, blast_time, t_current):
    """计算t_current时刻烟幕中心（原文件逻辑：起爆后下沉，地面约束）"""
    if t_current < blast_time:
        return blast_pos  # 未起爆：烟幕在起爆点不动
    # 已起爆：按固定速度下沉
    sink_time = t_current - blast_time
    sink_z = max(0.0, blast_pos[2] - CLOUD_SINK_V * sink_time)  # 不低于地面
    return np.array([blast_pos[0], blast_pos[1], sink_z])

def is_missile_sheltered(missile_idx, t_current, clouds):
    """判断t时刻导弹是否被遮蔽（原文件判定规则：烟幕到导弹-真目标连线距离≤10m）"""
    # 步骤1：获取当前导弹位置和飞行方向
    missile_p = missile_pos(missile_idx, t_current)
    missile_dir = (TGT - missile_p) / np.linalg.norm(TGT - missile_p)  # 导弹指向真目标的单位向量
    # 步骤2：遍历所有烟幕，判断是否有有效遮蔽
    for cloud in clouds:
        blast_time = cloud["blast_time"]
        blast_pos = cloud["blast_pos"]
        # 烟幕有效条件：t在[起爆时间，起爆时间+20s]
        if not (blast_time - 1e-6 <= t_current <= blast_time + CLOUD_DURATION + 1e-6):
            continue
        # 计算当前烟幕中心位置
        cloud_p = cloud_center(blast_pos, blast_time, t_current)
        # 计算烟幕中心到导弹-真目标连线的距离（向量叉积公式，原文件核心判定）
        vec_missile_cloud = cloud_p - missile_p
        dist_to_line = np.linalg.norm(np.cross(vec_missile_cloud, missile_dir))
```

```
    if dist_to_line <= R_CLOUD + 1e-6:
    return True  # 有一个烟幕满足即判定遮蔽
    return False


    def decode_solution(sol):
    """解码优化变量（原文件变量布局，无复杂修复逻辑）"""
    idx = 0
    # 1. 无人机飞行角度（弧度，归一化）
    thetas = np.array([clip_angle(sol[idx+i]) for i in range(NUM_UAV)])
    idx += NUM_UAV
    # 2. 无人机飞行速度（裁剪到合法范围[70,140]）
    speeds = np.clip(sol[idx:idx+NUM_UAV], UAV_SPEED_MIN, UAV_SPEED_MAX)
    idx += NUM_UAV
    # 3. 投弹槽位启用标志（0=禁用，1=启用，阈值0.5）
    s_flags = np.where(sol[idx:idx+TOTAL_SLOTS] > 0.5, 1, 0)
    idx += TOTAL_SLOTS
    # 4. 烟幕投放时间（裁剪到[0, 仿真结束时间]）
    drop_times = np.clip(sol[idx:idx+TOTAL_SLOTS], 0.0, SIM_T_END)
    idx += TOTAL_SLOTS
    # 5. 烟幕起爆延迟（≥0，避免负延迟）
    blast_delays = np.maximum(sol[idx:idx+TOTAL_SLOTS], 0.0)
    idx += TOTAL_SLOTS
    return thetas, speeds, s_flags, drop_times, blast_delays

    def evaluate_solution(sol):
    thetas, speeds, s_flags, drop_times, blast_delays = decode_solution(sol)
    clouds = []  # 存储有效烟幕信息

    # 步骤1：生成有效烟幕（仅启用的槽位，过滤无效投弹）
    for slot_global in range(TOTAL_SLOTS):
    if s_flags[slot_global] == 0:
    continue  # 跳过禁用的槽位
    # 确定槽位所属无人机
    uav_idx = slot_global // SLOTS_PER_UAV
    # 计算烟幕起爆点和起爆时间
    blast_pos, blast_time = cloud_blast_pos(
    uav_idx=uav_idx,
    theta=thetas[uav_idx],
    speed=speeds[uav_idx],
    t_drop=drop_times[slot_global],
    t_blast_delay=blast_delays[slot_global]
    )
    # 过滤无效烟幕（起爆时间超过最晚导弹命中+5s，视为浪费）
    if blast_time > SIM_T_END + 5.0:
    continue
    clouds.append({
```

```
        "blast_pos": blast_pos,
        "blast_time": blast_time,
        "uav_idx": uav_idx
    })

    # 步骤2：仿真计算每枚导弹的遮蔽时长
    total_shelter = 0.0  # 总遮蔽时长（适应度核心指标）
    for missile_idx in range(len(MISSILE_NAMES)):
    t = 0.0  # 当前仿真时间
    missile_shelter = 0.0  # 该导弹的遮蔽时长
    missile_hit_time = np.linalg.norm(MISSILE_INITS[missile_idx]-TGT)/MISSILE_SPEED
        # 导弹命中时间
    # 遍历每个时间步，判断遮蔽状态
    while t <= missile_hit_time + 1e-6:
    if is_missile_sheltered(missile_idx, t, clouds):
    missile_shelter += DT  # 遮蔽则累加时长
    t += DT
    total_shelter += missile_shelter

    # 步骤3：简单惩罚
    penalty = 0.0
    if len(clouds) > TOTAL_SLOTS:  # 理论不会触发，槽位数量限制
    penalty += (len(clouds) - TOTAL_SLOTS) * 10.0
    # 适应度 = 总遮蔽时长 - 惩罚（原文件核心逻辑）
    fitness = total_shelter - penalty
    return fitness, {
        "total_shelter": total_shelter,
        "clouds": clouds,
        "penalty": penalty    }


    def create_de_bounds():
    """创建优化变量边界（原文件变量范围，固定无动态调整）"""
    bounds = []
    # 1. 无人机角度（0~2π）
    for _ in range(NUM_UAV):
    bounds.append((0.0, 2 * math.pi))
    # 2. 无人机速度（70~140）
    for _ in range(NUM_UAV):
    bounds.append((UAV_SPEED_MIN, UAV_SPEED_MAX))
    # 3. 槽位启用标志（0~1）
    for _ in range(TOTAL_SLOTS):
    bounds.append((0.0, 1.0))
    # 4. 投放时间（0~SIM_T_END）
    for _ in range(TOTAL_SLOTS):
    bounds.append((0.0, SIM_T_END))
    # 5. 起爆延迟（0~10，避免过长延迟）
```

```
    for _ in range(TOTAL_SLOTS):
    bounds.append((0.0, 10.0))
    return bounds

    def init_de_population(bounds, pop_size):
    dim = len(bounds)
    pop = np.zeros((pop_size, dim))
    for i in range(dim):
    low, high = bounds[i]
    pop[:, i] = np.random.uniform(low, high, pop_size)  # 均匀随机初始化
    return pop

    def de_mutation(pop, target_idx, F):
    pop_size = pop.shape[0]
    # 选择3个与目标个体不同的随机个体
    while True:
    r1, r2, r3 = np.random.choice(pop_size, 3, replace=False)
    if r1 != target_idx and r2 != target_idx and r3 != target_idx:
    break
    # 变异公式: v = r1 + F*(r2 - r3)（原文件基础公式）
    return pop[r1] + F * (pop[r2] - pop[r3])

    def de_crossover(target, mutant, CR):
    dim = len(target)
    trial = target.copy()
    # 随机选择一个维度强制交叉（确保trial与target不同）
    j_rand = np.random.randint(dim)
    for j in range(dim):
    if np.random.rand() <= CR or j == j_rand:
    trial[j] = mutant[j]
    return trial

    def de_selection(target, trial, target_fitness):
    trial_fitness, trial_info = evaluate_solution(trial)
    if trial_fitness >= target_fitness:
    return trial, trial_fitness, trial_info  # trial更优: 替换目标个体
    else:
    return target, target_fitness, None  # target更优: 保留原个体

    def run_basic_de():

    bounds = create_de_bounds()
    dim = len(bounds)
    # 步骤1: 初始化种群
    pop = init_de_population(bounds, DE_NP)
    # 步骤2: 初始评估种群适应度
    fitness = np.zeros(DE_NP)
```

```
    pop_info = [None] * DE_NP
    for i in range(DE_NP):
    fitness[i], pop_info[i] = evaluate_solution(pop[i])
    # 步骤3：初始化全局最优
    gbest_idx = np.argmax(fitness)
    gbest_val = fitness[gbest_idx]
    gbest_pos = pop[gbest_idx].copy()
    gbest_info = pop_info[gbest_idx]

    for gen in range(DE_MAX_GEN):
    for i in range(DE_NP):
    # 变异（rand/1策略）
    mutant = de_mutation(pop, i, DE_F)
    # 交叉（二项式交叉）
    trial = de_crossover(pop[i], mutant, DE_CR)
    # 边界修复
    for j in range(dim):
    trial[j] = np.clip(trial[j], bounds[j][0], bounds[j][1])
    # 选择（贪心选择）
    pop[i], fitness[i], new_info = de_selection(pop[i], trial, fitness[i])
    # 更新全局最优
    if fitness[i] > gbest_val:
    gbest_val = fitness[i]
    gbest_pos = pop[i].copy()
    gbest_info = new_info if new_info is not None else pop_info[i]
    # 每50代打印进度
    if (gen + 1) % 50 == 0:
    print(f"DE迭代第{gen+1}/{DE_MAX_GEN}代 |
        全局最优遮蔽时长：{gbest_info['total_shelter']:.2f}s")
```