# 图覆盖项目算法

## 📌 内容概述

1.算法实现的详细过程

2.参数介绍

3.参数调整的对比结果，包括图和数据

# 1. 算法描述

## 1.1 问题描述

> **以竞赛的问题描述为例，当然也可以小改问题条件**
>
> 每个目标由个子目标区域组成，每个子目标区域为不规则多边形。子目标区域可以为矩形、圆形，或带孔多边形。用DD对目标进行打击。每发DD的实际弹着点与打击点存在随机性偏差。DD命中后，向弹着点周边一定范围的圆内对子目标区域造成破坏。选手考虑DD命中的随机性因素，以及DD对不同子目标区域的破坏覆盖面积，规划打击点坐标，使得对目标的实际破坏覆盖面积最大，提升把握程度，并尽可能降低耗弹量。
>
> **关于覆盖面积比、把握程度**
>
> 覆盖面积比是指，多枚DD命中后，其破坏范围的并集，与子目标区域交汇部分的面积与子目标区域面积之比。
>
> 把握程度是指，在指定次仿真试验中，至少有次试验达到覆盖面积比要求。

## 1.2 模块划分

| 类别 | 模块 | 文件 | 函数 | 描述 |
| --- | --- | --- | --- | --- |
| 基本算法（纯计算） | 轰炸偏差函数CEP | CoverageRatioWithError | func0 | 题目中已经给定公式，直接使用即可 |
| | 根据DD打击位置计算覆盖比例 | area | draw_small_circle | 目标区域和所有轰炸圆的交集 |
| | 根据DD打击位置计算把握程度 | CoverageRatioWithError | Min_Value | 模拟打击N次，有K次满足覆盖比，则把握程度为K/N |
| | | | | |

| | | | | |
|---|---|---|---|---|
| | 画结果图 | draw | draw_pic | 画出目标区域和当次实验中的轰炸圆 |
| 粒子群 | 粒子群算法 | PSO、PSO2 | pso、pso2 | pso的目标函数是覆盖比、pso2的目标函数是平均覆盖几率大于给定把握程度的像素点数量 |
| 概率圆 | 概率圆的初始化计算 | calculating_result | get_P_circle | 概率圆是带权重的圆，需计算并将其存入二维数组 |
| | 根据DD打击位置计算平均覆盖几率数组 | area | get_P_result | 计算目标区域中所有点的平均覆盖几率 |
| | 根据DD打击位置计算平均覆盖几率大于给定把握程度的像素点数量 | area | get_P_num | 计算平均覆盖几率大于给定把握程度的点的数量，若数量大于覆盖比*目标区域面积，认为能够满足 |
| 无关圆检测 | 第一步检测 | area | first_check | 去除该轰炸圆后覆盖比例变化很小（默认0.1%），且去除该圆后覆盖比依然满足条件的DD |
| | 第二步检测 | area | second_check | 找到单独删除该圆，对覆盖比例影响最小的DD，若删除它仍满足把握程度，则删除 |

💡 概率圆是本项目中为了解决轰炸偏差造成的不确定性提出的概念。由于存在偏差时，可能出现覆盖比已经100%，目标函数无法继续有效优化，但是把握程度依然不满足给定条件。因此，为了继续优化，提升把握程度，提出了基于概率圆的算法。

如下图所示，如果以二维数组的形式表示概率圆，则中心区域坐标为1或趋近于1，距离圆心越远，数值越小，直到为0。

### 1.2.1 基本算法（纯计算）

1. 轰炸偏差函数CEP

```
def func0(Init_solution):
    solution = []
    # 定位误差
    length = random.randint(0, drawInit.locate_error)
    x_change = random.randint(int(np.ceil(-length)), int(np.floor(length)))
    y_1 = [int(np.floor(math.sqrt(length ** 2 - x_change ** 2))),
           int(np.ceil(-math.sqrt(length ** 2 - x_change ** 2)))]
    y_change = random.choice(y_1)
    coverageRatio = []

    # Init_solution表示无偏差的DD打击位置坐标（偶数是横坐标），x_trans和y_trans是CEP偏差之后的坐标
    for num0 in range(len(Init_solution)):
        r1 = random.random()
        r2 = random.random()
        if (num0 % 2) == 0:
            x_coor = Init_solution[num0]
            x_trans = x_coor + m * drawInit.CEP * math.sqrt(-2 *math.log(r1)) * math.cos(2 * math.pi * r2) + x_change
            solution.append(x_trans)
        else:
            y_coor = Init_solution[num0]
```

```
            y_trans = y_coor + m * drawInit.CEP * math.sqrt(-2 *math.log(r1))
* math.sin(2 * math.pi * r2) + y_change
            solution.append(y_trans)
    # area用于求覆盖比
    coverageArea = area(small_circles_location=solution + [0])
    coverageRatio.append(coverageArea)
    return coverageRatio
```

2. 根据DD打击位置计算覆盖比例

```
# small_circles表示DD打击位置的横纵坐标，pic在多个目标区域中代表第几个子区域（若目标区域是个整体不用考虑）
def draw_small_circle(small_circles, pic):
    n = 0
    # border是整张图的边界，semi_border是半张图（将图切成四个象限）
    picture_small = np.zeros((border, border))
    for num0 in range(circle_num * 2):
        small_circles[num0] = round(small_circles[num0])
    semi_border = int(border / 2)
    for circles_num in range(circle_num):
        x_left = int(small_circles[circles_num * 2] - small_r + semi_border)
        x_right = int(small_circles[circles_num * 2] + small_r + semi_border)
        y_left = int(small_circles[circles_num * 2 + 1] - small_r +
semi_border)
        y_right = int(small_circles[circles_num * 2 + 1] + small_r +
semi_border)
        # 循环遍历整张图的所有像素点，若该点是目标区域且被DD打中则自增1
        for x in range(x_left, x_right):
            for y in range(y_left, y_right):
                if 0 <= x < border and 0 <= y < border:
                    if drawInit.small_circle_shape[x - x_left][y - y_left] ==
1:
                        if picture_small[x][y] == 0 and
drawInit.picture_big[pic][x][y] == 1:
                            picture_small[x][y] = 1
                            n += 1
    # 覆盖比=击中面积/目标区域面积
    return n / drawInit.big_area[pic] * 100
```

3. 根据DD打击位置计算把握程度

```
# 通过多进程的方式实现，具体不需要管，总之就是模拟了N次，看有多少次满足覆盖比
def Min_Value(Init_solution, number, w):
```

```
    time1 = time.time()
    args = ()  # 多进程
    kwargs = {}  # 多进程
    mp_pool = multiprocessing.Pool(drawInit.thread_num)  # 进程数
    obj = partial(_obj_wrapper0, args, kwargs)  # 多进程
    x = np.random.rand(number, len(Init_solution))
    for i in range(number):
        x[i] = Init_solution

    coverageRatio = np.array(mp_pool.map(obj, x))
    ratio1 = w * 100
    ratiovalue = []
    togather_w = [0] * number
    sum_togather = 0
    for pic in range(len(drawInit.big_area)):
        sum0 = 0
        sum1 = 0
        for num in range(len(coverageRatio)):
            sum1 += coverageRatio[num][0][pic]
            if coverageRatio[num][0][pic] >= ratio1:
                togather_w[num] += 1
                sum0 += 1
        ratiovalue.append(sum0 / number)
        avg = round(sum1 / number, 3)
        print('图形: ', pic, '的平均覆盖比例: ', avg)

    for epoch in range(number):
        if togather_w[epoch] == len(drawInit.big_area):
            sum_togather += 1

    time2 = time.time()
    hours, minutes, seconds = c_time(time2 - time1)
    print('计算把握程度用时为:  ', hours, 'h, ', minutes, 'min ,', seconds, 's')

    return ratiovalue, sum_togather / number
```

4. 画结果图：根据DD打击位置和半径画图，覆盖目标区域的图

```
# 生成结果图，保存到本地，名称包含时间
def draw_pic(circles_location, flag, image_path, fgbl, bwcd):
    color = [255, 0, 0]

    if flag == 1:
        circle_sum = int(len(circles_location) / 2)
        img = mp.imread(image_path)
```

```
        np.array(img, np.int32)
        plt.title('覆盖不比例: ' + str(fgbl) + '%, 把握程度: ' + str(bwcd) + '%')
        dir = os.getcwd()
        pic_dir = os.path.join(dir, 'picture')
        if os.path.exists(pic_dir):
            os.listdir(pic_dir)
        else:
            os.mkdir(pic_dir)
        all_pic = os.listdir(pic_dir)
        pic_name = str(datetime.now().strftime("%Y-%m-%d_%H-%M-%S_%f")[:-3]) +
'.png'
        pic_path = os.path.join(pic_dir, pic_name)
        cor_trans(circles_location)
        for current in range(circle_sum):
            rr, cc = draw.circle_perimeter(round(circles_location[current *
2]), round(circles_location[current * 2 + 1]), circle_r)
            draw.set_color(img, [cc, rr], color=color)
            plt.imshow((img * 255).astype(np.uint8))
            img = np.clip(img, 0.0, 1.0)
        mp.imsave(pic_path, img)
    if flag == 2:
        img = np.clip(img, 0.0, 1.0)
        out_path = 'picture/' + str(datetime.now().strftime("%Y-%m-%d_%H-%M-
%S_%f")[:-3]) + '.png'
        mp.imsave(out_path, img)
```

### 1.2.2 粒子群算法（PSO、PSO2）

💡 两个文件几乎一样，就是目标函数的区别，pso2目标函数为覆盖比，pso2目标函数为平均覆盖几率大于把握程度的点个数。

```
# 也是采用多进程的方式
def pso2(l, u, small_num, swarmsize=40, maxiter=100, debug=False):
    time1 = time.time()
    args = ()  # 多进程
    kwargs = {}  # 多进程
    obj = partial(_obj_wrapper, args, kwargs)  # 多进程
    mp_pool = multiprocessing.Pool(drawInit.thread_num)  # 进程数

    D = small_num * 2  # 小圆个数*2是维度数，是所有轰炸圆横纵坐标加在一起
    lb = l * np.ones(D)
    ub = u * np.ones(D)
    interval = np.abs(ub - lb)
```

```
    vhigh = 0.2 * interval  # 粒子移动速度下限
    vlow = -vhigh  # 粒子移动速度下限

    # 下面这些参数都是英文表示，都是粒子群算法中的概念
    # Initialize objective function
    ite = np.linspace(1, maxiter, maxiter)
    Weight = 0.9 - ite * 0.7 / maxiter
    c1 = 1
    c2 = 1
    x = np.random.rand(swarmsize, D)  # particle positions
    fx = np.zeros(swarmsize)  # current particle function values适应度
    p = np.zeros_like(x)  # best particle positions **pbest**
    fp = np.ones(swarmsize) * 0  # best particle function values **pbest value**
    fg = 0  # best swarm position starting value   **gbest value**

    for i in range(swarmsize):  # particle positions
        for j in range(D):
            x[i][j] = np.random.uniform(lb[j], ub[j])
    # fx为目标函数的得到的结果
    fx = np.array(mp_pool.map(obj, x))

    # Initialize the particle's velocity
    v = vlow + np.random.rand(swarmsize, D) * (vhigh - vlow)  # vlow = -vhigh vhigh = np.abs(ub - lb)#限定
    # Calculate objective and constraints for each particle

    for i in range(swarmsize):
        if fx[i] > fp[i]:
            p[i, :] = x[i, :]
            fp[i] = fx[i]
    # Update swarm's best position
    i_max = np.argmax(fp)
    if fp[i_max] > fg:
        fg = fp[i_max]  # best swarm position starting value
        g = p[i_max, :].copy()  # best swarm position
    else:
        g = x[0, :].copy()

    # Iterate until termination criterion met ##################################
    fri_best = np.ones([swarmsize, D])
    for i in range(swarmsize):
        fri_best[i:] = i

    it = 0
    vel = np.zeros_like(v)
```

```
    while it <= maxiter - 1:
        for i in range(swarmsize):
            vel[i, :] = Weight[it] * v[i, :] + (c1 * np.random.rand(1, D) *
(p[i, :] - x[i, :])) + (
                    c2 * np.random.rand(1, D) * (g - x[i, :]))
        maskl = vel < vlow
        masku = vel > vhigh
        vel = vel * (~np.logical_or(maskl, masku)) + vlow * maskl + vhigh *
masku
        pos = x + vel
        x = pos
        v = vel
        # Correct for bound violations限制边界
        maskl = x < lb
        masku = x > ub
        x = x * (~np.logical_or(maskl, masku)) + lb * maskl + ub * masku
        # fx为目标函数的得到的结果
        fx = np.array(mp_pool.map(obj, x))
        for i in range(swarmsize):
            if fx[i] > fp[i]:
                p[i, :] = x[i, :]
                fp[i] = fx[i]
        i_max = np.argmax(fp)
        if fp[i_max] > fg:
            g = p[i_max, :].copy()
            fg = fp[i_max]

        if debug:
            print('New best for swarm at iteration {:}: 目标区域像素点平均覆盖几率
为: {:}%'.format(it, fg * 100 / drawInit.big_area[0]))

        it += 1
    time2 = time.time()
    f = area(g)
    print('Stopping search: maximum iterations reached -->
{:}'.format(maxiter))
    hours, minutes, seconds = c_time(time2 - time1)
    print('运行时间为:  ', hours, 'h, ', minutes, 'min ,', seconds, 's')
    return g, f, fg * 100 / drawInit.big_area[0]
```

### 1.2.3 概率圆

1. 初始化采样、计算

这里代码量有点大，不具体展示了。周老师提出的通过计算的方式初始化概率圆。因为最开始为了得到概率圆，采用随机N次，通过大量的采样数据模拟出概率分布，后来为了提速并精确化，采用计算的

方式。

2. 根据DD打击位置计算平均覆盖几率矩阵

```
# 平均覆盖几率矩阵的的计算，其实就是计算每个目标区域中的点被打击到的概率
def get_P_result(small_circles):
    border = drawInit.border
    w = 1 - drawInit.w2[0]
    print("大小:")
    print(drawInit.P_circle.size)
    picture_P = np.ones((border, border))
    circle_num = int(len(small_circles) / 2)
    for num0 in range(circle_num * 2):
        small_circles[num0] = round(small_circles[num0])
    semi_border = int(border / 2)
    rr = drawInit.small_r * 2
    for circles_num in range(circle_num):
        x_left = int(small_circles[circles_num * 2] - rr + semi_border)
        x_right = int(small_circles[circles_num * 2] + rr + semi_border)
        y_left = int(small_circles[circles_num * 2 + 1] - rr + semi_border)
        y_right = int(small_circles[circles_num * 2 + 1] + rr + semi_border)
        for x in range(max(x_left, 0), min(x_right, border)):
            for y in range(max(y_left, 0), min(y_right, border)):
                picture_P[x][y] *= 1 - drawInit.P_circle[x - x_left][y -
y_left]
    for i in range(border):
        for j in range(border):
            if drawInit.picture_big[0][i][j] == 0:
                picture_P[i][j] = -1
            else:
                picture_P[i][j] = 1 - picture_P[i][j]
    return picture_P
```

3. 根据DD打击位置计算平均覆盖几率大于给定把握程度的像素点数量

```
# 计算平均覆盖几率大于把握程度的像素点数量，若大于把握程度的像素点大于覆盖比，认为满足条件
def get_P_num(small_circles):
    r = 0
    border = drawInit.border
    picture_P = np.ones((border, border))
    circle_num = int(len(small_circles) / 2)
    for num0 in range(circle_num * 2):
        small_circles[num0] = round(small_circles[num0])
    semi_border = int(border / 2)
    rr = drawInit.small_r * 2
```

```
    for circles_num in range(circle_num):
        x_left = int(small_circles[circles_num * 2] - rr + semi_border)
        x_right = int(small_circles[circles_num * 2] + rr + semi_border)
        y_left = int(small_circles[circles_num * 2 + 1] - rr + semi_border)
        y_right = int(small_circles[circles_num * 2 + 1] + rr + semi_border)
        for x in range(max(x_left, 0), min(x_right, border)):
            for y in range(max(y_left, 0), min(y_right, border)):
                picture_P[x][y] *= 1 - drawInit.P_circle[x - x_left][y -
y_left]
    for i in range(border):
        for j in range(border):
            if drawInit.picture_big[0][i][j] == 1:
                r += 1 - picture_P[i][j]
    return r
```

### 1.2.4 无关圆检测（area）

1. 第一步检测：遍历所有DD打击位置，若去除该圆后覆盖比例变化很小（当前默认0.1%），且去除该圆后覆盖比例依然满足条件，则尝试删除它

```
def first_check(xopt1, mini_num, start_area, w):
    print("第一步遍历删去圆之前的坐标为：{}".format(xopt1))
    useless_min_circle = []
    for mini_circles_num in range(mini_num):
        index = [mini_circles_num * 2, mini_circles_num * 2 + 1]
        xopt2 = np.delete(xopt1, index)
pt2)
        # 若删除该圆对覆盖比影响小于0.1%，则删除
        if start_area - current_area < 0.001 and current_area > w:
            useless_min_circle.append(index)
    print("第一步遍历得到的无关圆个数为：{}".format(len(useless_min_circle)))
    return useless_min_circle
```

2. 第二步检测：遍历所有DD打击位置，找到删除该圆对覆盖比例影响最小的DD，尝试删除它

```
def second_check(xopt1, mini_num, max_area):
    print("第二步遍历删去圆之前的坐标为：{}".format(xopt1))
    max_xopt2 = xopt1
    # 依次尝试删除每个圆，若删除后依然满足把握程度，则删除这个圆
    for mini_circles_num in range(mini_num):
        index = [mini_circles_num * 2, mini_circles_num * 2 + 1]
        xopt2 = np.delete(xopt1, index)
        current_area = area(xopt2)
```

```
        if current_area > max_area:
            max_area = current_area
            max_xopt2 = xopt2
    print("第二步遍历删去圆之后的坐标为：{}".format(max_xopt2))
    return max_xopt2, max_area
```

### 1.3 执行步骤

1. xml文件读取，或自定义赋值
2. 变量初始化
3. 通过粒子群算法得到所有DD打击位置
4. 判断当前所有DD，能否满足给定的覆盖比例和把握程度，若不满足，小圆数量+1，返回第3步，若满足，执行第5步
5. 执行第一步检测，尝试删去无关圆1，删除后判断当前所有DD能否满足给定的覆盖比例和把握程度，不满足则不删，满足则删去
6. 执行第二步检测，尝试删去无关圆2，删除后判断当前所有DD能否满足给定的覆盖比例和把握程度，若不满足则不删，执行第7步，若满足则删去，重复执行第6步
7. 画结果图

## 2. 初始化参数

```
# 计算轰炸圆二维数组
def draw_small_circle(small_r):
    small_circle_shape = np.zeros((small_r * 2, small_r * 2))
    small = 0
    for x0 in range(0, small_r * 2):
        for y0 in range(0, small_r * 2):
            if (x0 - small_r) ** 2 + (y0 - small_r) ** 2 <= small_r ** 2:
                small_circle_shape[x0][y0] = 1
                small += 1
    return small, small_circle_shape

# 默认初始化轰炸圆个数（估计值）
def init_circle_num(big_area, small_area):
    circles = math.ceil(big_area[0] / small_area)
    return circles

# 默认初始化轰炸圆圆心移动范围（目前是目标区域的外接正方形）
def init_big_range(picture_big):
    semi_border = int(border / 2)
    x_l = y_l = 999
```

```
    x_u = y_u = -1
    for x in range(border):
        for y in range(border):
            if picture_big[0][x][y] == 1:
                if y < y_l:
                    y_l = y
                if y > y_u:
                    y_u = y
                if x < x_l:
                    x_l = x
                if x > x_u:
                    x_u = x
    if y_l < x_l:
        x_l = y_l
    if y_u > x_u:
        x_u = y_u
    return x_l - semi_border, x_u - semi_border

class drawInit:
    def __init__(self):
        self = self
    border = border
    m = 0.84932180
    color = [255, 255, 255]
    image = Image.new('RGB', (border, border), (color[0], color[1], color[2]))
    image.save("init.png")
    xml_path = "3.xml"
    pic_path = "round/1.png"
    w1 = 0.8  # 覆盖比
    w2 = 0.8  # 把握程度
    CEP = 0.1  # 轰炸偏差常数
    small_r = 60  # 轰炸圆半径
    all_area, all_pic = get_pic_area(pic_path)  # 输入为图片，而不是xml
    big_area = [all_area]  # 目标区域面积
    picture_big = [all_pic]  # 目标区域二维数组
    P_circle = get_P_circle(m, CEP, small_r)  # 概率圆二维数组
    swarmsize = 30  # 粒子群的粒子个数
    maxiter = 30  # 粒子群的迭代次数
    thread_num = 8  # 线程数
    debug = True  # 不用管
    w1 = [w1]  # 覆盖比例
    w2 = [w2]  # 把握程度
    l = -border / 2  # 粒子的上下界
    u = border / 2
    epochs = 1000  # 迭代次数
    locate_error = 0  # 定位偏差

```

```
    # print("目标区域像素:" + str(all_area))
    small_area, small_circle_shape = draw_small_circle(small_r)
    # print("使用图片: ", pic_path)
    l, u = init_big_range(picture_big)
    # print("初始化位置上下限: ", l, ',', u)
    small_num = init_circle_num(big_area, small_area)
```

# 3. 参数调整与对比

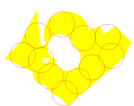
💡 覆盖比、把握程度、CEP、轰炸圆半径、输入图片、粒子个数、迭代次数都可以调整。

## 3.1 参数值建议范围

| 参数 | 范围 | 过大 | 过小 |
| --- | --- | --- | --- |
| 覆盖比 | 0.5～1.0 | 1.0已经是最大了 | 太容易实现，算法没意义 |
| 把握程度 | 0.5～0.95 | 如果存在误差（CEP>0），把握程度无法接近1，永远存在偏差导致无法满足覆盖比的情况 | 太容易实现，算法没意义 |
| CEP | 0～20 | 轰炸偏差太严重，把握程度太低 | 0表示没有偏差（指哪打哪） |
| 轰炸圆半径 | 20～80 | 只需很少的圆就能满足条件 | 需要太多的轰炸圆（接近小圆点） |
| 粒子个数 | 30～100 | 运算速度慢，没有其他缺点（效果也有上限） | 粒子太少，效果不好 |
| 粒子迭代次数 | 30～120 | 已经收敛了，无法继续优化 | 迭代次数少，还没有收敛 |

## 3.2 对比示例

1. 不规则图形round/1.png(粒子个数80，迭代次数90)

| 序号 | 给定覆盖比 | 给定把握程度 | CEP | 轰炸圆半径 | DD个数 | 覆盖比 | 把握程度 | 图 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 0.8 | 0.8 | 4 | 60 | 11 | 83.4 | 1.0 |  |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 2 | 0.8 | 0.8 | 16 | 60 | 13 | 87.2 | 0.92 |  |
| 3 | 0.8 | 0.8 | 16 | 45 | 24 | 89.7 | 84.9 |  |
| 4 | 0.8 | 0.6 | 16 | 45 | 23 | 89.3 | 75.4 |  |
| 5 | 0.7 | 0.8 | 16 | 45 | 18 | 80.2 | 87.2 |  |

2. 图形round/eight_220_120.png(粒子个数80，迭代次数90)

| 序号 | 给定覆盖比 | 给定把握程度 | CEP | 轰炸圆半径 | DD个数 | 覆盖比 | 把握程度 | 图 |
|---|---|---|---|---|---|---|---|---|
| 1 | 0.8 | 0.8 | 4 | 30 | 8 | 85.4 | 1.0 |  |
| 2 | 0.8 | 0.8 | 16 | 30 | 13 | 99.4 | 86.8 |  |
| 3 | 0.8 | 0.8 | 16 | 20 | 33 | 97.6 | 0.8 |  |
| 5 | 0.8 | 0.6 | 16 | 20 | 32 | 96.1 | 69.4 |  |
| 4 | 0.7 | 0.8 | 16 | 20 | 25 | 89.5 | 86.8 |  |