# 生产过程中决策问题的优化改进

## 摘要

优化生产过程中的决策是制造业企业长足发展的核心问题之一。本文以企业生产过程中的几种常见情况为背景，建立了**序贯抽样模型、基于递推逼近的遍历优化模型、基于分治算法的遗传退火模型、基于贝叶斯统计方法的蒙特卡洛模拟**等模型，来求解出企业生产过程中的最优决策。

**对于问题一：**为使抽验检测的信度到达要求值，需要同时减小**第一类错误**与**第二类错误的发生率**，这就会导致最小样本量上升，因此该问题实质上就是样本量与两类错误的平衡过程。本文开始采用了**简单抽样模型**，在单侧置信区间下得出最大允许误差为 0.02 时，为使 $\alpha$ 达到 95% 与 90% 的置信区间，所需最小样本量分别为 609 件与 370 件。分析后发现，当真实次品率出现偏差后该模型所得结果较差，因此引入**功效分析**与**自适应误差**来改进，建立了**序贯抽样模型**。该模型具有良好的信度，并且可以根据实施抽样灵活调整抽样策略，以达成最小化抽样个数效果。

**对于问题二：**本文以最终稳定**生产出一件产品利润的数学期望**作为决策指标提出了**基于递推逼近的遍历模型**。首先写出生产不同批次的递推表达式，并在此基础上由上一批次参数递推出下一批次参数进一步求解利润，在最终利润变化值达到阈值后认为稳定，该利润作为最稳定利润值，在此基础上**遍历决策变量**所有情况得到最优指标以及决策，并将最终结果与第三问中的子问题求解结果相对比互相验证。最终，得到的 6 种情况的利润期望分别为 **15.80 元，8.60 元，14.89 元，14.25 元，11.86 元，19.84 元**。

**对于问题三：**本文以最终稳定**生产出一件产品利润的数学期望**作为决策指标以及依据提出了**基于分治算法的遗传退火模型**，运用**分治算法**将多工序利润计算拆分为多个类似问题二的子问题，并运用方程求解出**每个子问题最终稳定输出次品率，生产成本，作为下一级子问题的进货次品率以及进货成本**逐步求解到成品，最终求得总利润，在此基础上建立遍历模型求解出全局最优解在此基础上选择合适模型最终确定建立**遗传退火混合模型**以应对大可行域最优决策求解，模型表现良好，最终得到生产出一件成品的利润的数学期望为 **57.03 元**。

**对于问题四：**本文通过**贝叶斯统计**，以 **Beta 分布**拟合先验与后验概率，得到相应概率密度函数来更新二三问中的次品率。同时基于概率密度函数，在置信域内均匀取值，对每一取得的概率组合，计算获取候选策略。最后利用**蒙特卡洛模拟**的方法产生随机值检验不同候选策略，以利润期望与方差为两个评判标准，选出非劣决策策略构成 **Pareto 解集**供决策企业参考。

最后，本文对模型进行了检验，并评估优缺点。

**关键字：** 序贯抽样　分治算法　遗传退火混合模型　蒙特卡洛模拟

# 一、 问题重述

## 1.1 问题背景

如何优化生产过程中的决策是企业实现利润最大化、我国制造业转型升级的重要命题。尤其是在精密性电子产品的生产制造中，零配件的质量、复杂多样的装配环节等都会给企业带来挑战。本文问题背景是在零配件供应商声称的零配件次品率标称值、企业的检测成本、装配成本、拆解成本、市场售价、调换损失等量已知的情况下，通过不同的检测方案，对企业产品的生产进行决策优化，以求达到利润最大化。

## 1.2 问题信息

- 该企业生产电子产品，需购入两种 (若干种) 零配件进行装配；
- 若两种（若干种）零配件不合格，则成品也不合格；
- 若两种（若干种）零配件都合格，装配出的成品也不一定合格；
- 可以对两种（若干种）零配件以及成品进行检测，企业需要支付检测成本
- 装配后，企业检测或被用户退回的不合格的成品可以选择报废或者拆解，拆解可以获得原零配件，但需要支付拆解费用
- 用户若购买到不合格成品，企业需要给予调换并支付调换费用
- 企业利润 = 合格品个数 $*$ 市场售价 $-$ 总购买成本 $-$ 装配成本 $-$ 调换损失 $-$ 拆解费用 $-$ 检测成本

## 1.3 待求解问题

**问题 1**：在尽可能减少检测次数，降低检测成本的情况下，设计高效的抽样检测方案，并针对不同信度条件，判断是否接收零配件。

**问题 2**：根据题中表 1 中的不同情况，为企业生产过程的各个阶段（零配件检测、成品检测、不合格品处理）制定决策方案，使得在该决策方案下企业可以实现利润最大化，并给出具体的决策依据和指标结果。

**问题 3**：在若干道工序和若干多个零配件的生产环境下，重复问题 2 的决策流程，优化各生产阶段的决策来增大利润，问题 3 可认为是问题 2 的一般化推广。同时需要给出在题中图 1、表 2 的情况下的具体决策方案。

**问题 4**：假设问题 2 和问题 3 中涉及的零配件、半成品和成品的次品率都是通过抽样检测方法得出的，参考问题 1 中的抽样方法，重新制定问题 2 和问题 3 中的生产决策方案。

# 二、 问题分析

## 2.1 问题 1 的分析

在问题 1 中，企业需要通过抽样检测判断供应商提供的零配件的次品率是否超过 10%。为此，使用统计学中的假设检验方法可以有效解决该问题。首先，在题干中的两个情形下，设定零假设，然后通过抽样来检验这一假设。在 95% 的置信水平下，要求检测出的次品率显著高于 10% 时拒收零配件；而在 90% 的置信水平下，如果检测表明次品率不超过 10%，则可以接收该批零配件。该问题可以用简单随机抽样、序贯抽样等不同的抽样方法来设计抽样方案。

## 2.2 问题 2 的分析

在问题 2 中，企业已知的量有：购入的零配件 1 和零配件 2 的总体次品率、购买单价及检测成本，装配成品时的次品率、成品的装配成本、成品的检测成本和市场售价，不合格品的调换损失与调节费用等。企业需要基于以上未知量制定生产过程中的决策，核心在于优化问题并求解出最大利润值。可以考虑用计算数学期望的思想，逐步推导出装配一件成品的利润期望计算公式，结合遍历模型，遍历出所有决策方案下的利润期望值，从而进行优化。

## 2.3 问题 3 的分析

在问题 3 中，企业已知 $m$ 种工序和 $n$ 种零配件的相应次品率，要对运用了这些工序和零配件的成品进行生产上的决策，需要不断优化决策方案，并且需要求出当 $m = 2$ 且 $n = 8$ 的具体情况下的具体决策方案以及指标结果。问题 3 是一个典型的多阶段生产过程决策优化问题，涉及多个零件、半成品、最终产品的组装、检测和拆解等决策。每个阶段的决策会影响生产成本、次品率以及最终利润。问题 3 可以考虑用分治算法来解决，将复杂的问题不断简单化，将整个大的过程按照问题 2 中的模式进行拆解，再组合起来计算总利润，再用遗传算法，模拟退火，遍历等方法进行具体求解。

## 2.4 问题 4 的分析

问题 4 在问题 2 和问题 3 的基础上做了更加符合实际情况的变化，即企业已知的零配件、装配半成品和成品的次品率并非是其真实值，而是根据抽样检测的方法估计所得。该问题可以根据贝叶斯公式进行贝叶斯统计检验，结合先验概率与后验概率的数学关系，在置信区间内求取相对应的可能的真实次品率值，将新的次品率组合分别带入到原问题 2、问题 3 的求解过程中去，求出新的决策方案，再进行优化挑选。

# 三、 模型假设

假设 1：企业装配的成品进入到市场后都可以找到买家，不需要考虑滞销问题。

假设 2：买家买到不合格成品后，一定可以将该不合格成品检测出并要求商家进行调换处理。

假设 3：该成品的市场售价、装配该成品的各种成本基本稳定，短时间内不会发生明显改变。

# 四、 符号说明

表 **1**　符号与说明

| 符号 | 说明 | 符号 | 说明 |
|---|---|---|---|
| $H_0$ | 零假设 | $p_i$ | 某种零配件的次品率或某工序中单纯由装配产生的次品率 |
| $\alpha$ | 假设检验中的拒真错误（第一类错误）概率 | $d_i$ | 决策变量，代表是否执行某种决策 |
| $\beta$ | 假设检验中的纳伪错误（第二类错误）概率 | $Profit$ | 企业获得的利润（单位：元） |
| $n_i$ | 某次抽样检测的样本量（单位：件） | $p_0$ | 某种零配件或成品的声称次品率（标称值） |
| $\epsilon$ | 允许误差范围 | $C_i$ | 各种成本（单位：元） |
| $k_{\text{reject}}$ | 拒收标准（单位：件） | $k_{\text{accept}}$ | 接收标准（单位：件数） |
| $Price$ | 单个成品的市场售价（单位：元） | $X_i$ | 某种情况下零配件的数量 |
| $d_i$ | 问题 2 中某种决策的决策变量 | $x_i$ | 问题 3 中，与检测决策相应的决策变量 |
| $Ch_i$ | 某种零配件或者成品的检测成本（单位：元） | $P_i$ | 某种零配件或者成品的实际次品率或成品率 |

# 五、 问题 1 的模型建立与求解

## 5.1 问题背景下的假设检验

在本问题情景中，本文引入统计学中常见的假设检验 [2] 来解决。在假设检验中，$H_0$ 为零假设，$\alpha$ 为拒真错误（概率），$\beta$ 为纳伪错误（概率）。实际情况与假设检验结果的如表 2。

表 **2**　假设检验中的决策与实际情况

| $H_0$ 检验 | | |
|---|---|---|
| **决策** | $H_0$ **为真** | $H_0$ **为假** |
| 接受 $H_0$ | $1-\alpha$ | 纳伪（第二类）错误概率 $(\beta)$ |
| 拒绝 $H_0$ | 拒真（第一类）错误概率 $(\alpha)$ | 检验功效 $(1-\beta)$ |

本题情境中，供应商给了企业该批零配件次品率不超过的标称值 $p_0 = 10\%$，本文假设实际上该批零配件总体的真实次品率为 $p$。

在第一种情形（在 95% 的信度下认定零配件次品率超过标称值，则拒收这批零配件）下，零假设 $H_0$ 为：该批零配件的实际次品率 $p > p_0$。假设检验的四种可能的情况及实际意义如下:

- 正确决策 $1-\alpha$ ：$H_0$ 为真, 且企业接受了正确的假设 $H_0$。即这批零配件实际上是不好的，且企业拒绝了。
- 拒真错误 $\alpha$：$H_0$ 为真，但是企业拒绝了该正确的假设 $H_0$。即这批零配件实际上是不好的，但企业接收了。
- 纳伪错误 $\beta$：$H_0$ 为假，但是企业接受了该错误的假设 $H_0$。即这批零配件实际上是好的，但企业拒收了。
- 检验功效 $(1-\beta)$：$H_0$ 为假，且企业拒绝了错误的假设 $H_0$。即这批零配件实际上是好的，且企业接收了。

在第二种情形（在 90% 的信度下认定零配件次品率不超过标称值，则接收这批零配件）下，零假设 $H_0$ 为：该批零配件的实际次品率 $p \leq p_0$。假设检验的四种可能的情况及实际意义如下:

- 正确决策 $1-\alpha$ ：$H_0$ 为真, 且企业接受了正确的假设 $H_0$。即这批零配件实际上是好的，而且企业接收了。
- 拒真错误 $\alpha$：$H_0$ 为真。但是企业拒绝了该正确的假设 $H_0$。即这批零件实际上是好的，单企业拒收了。
- 纳伪错误 $\beta$: $H_0$ 为假，但是企业接受了该错误的假设 $H_0$。即这批零件实际上是不好的，但是企业接收了。
- 检验功效 $(1-\beta)$：$H_0$ 为假，且企业拒绝了错误的假设 $H_0$。即这批零件实际上是不好的，且企业拒收了。

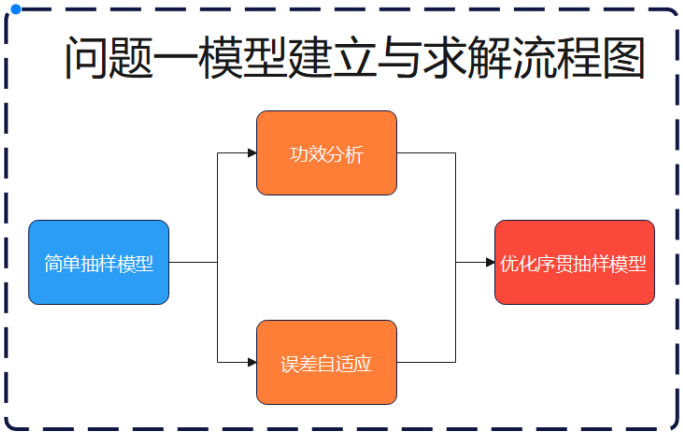



图 **1**　问题 **1** 求解流程图

在本问题中，考虑使用假设检验相关的统计学知识来求解本题。问题 1 的求解流程图如图 1。

### 5.2 基于简单随机抽样的方法的模型建立与求解

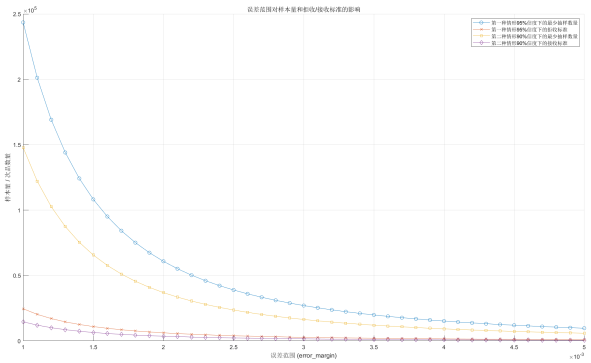
考虑用简单随机抽样的方法 [4] 来解决，由于只需要考虑单侧检验，单侧检验最小样本量计算公式 (1) 和接收标准、拒收标准计算公式 (2)。

$$n = \frac{z_\alpha^2 \cdot p_{true}(1 - p_{true})}{\epsilon^2} \tag{1}$$

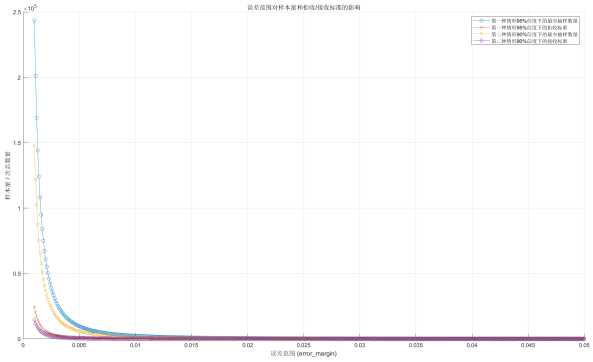
$$\begin{aligned} k_{\text{reject}} &= \inf\{k \in \mathbb{N} \mid P(X \leq k \mid n, p) \geq 1 - \alpha\} \\ k_{\text{accept}} &= \sup\{k \in \mathbb{N} \mid P(X \leq k \mid n, p) \leq \alpha\} \end{aligned} \tag{2}$$

表 **3**　简单随机抽样方法中不同误差范围的抽样方案

| 允许误差范围 | **0.001** | **0.002** | **0.003** | **0.004** | **0.005** | **0.006** | **0.007** | **0.008** | **0.009** | **0.01** | **0.02** | **0.03** | **0.04** | **0.05** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 第一种情形下的最小抽样数量 | 243499 | 60875 | 27056 | 15219 | 9740 | 6764 | 4970 | 3805 | 3007 | 2435 | 609 | 271 | 153 | 98 |
| 第一种情形下的拒收标准 | 24594 | 6209 | 2787 | 1583 | 1023 | 717 | 532 | 411 | 328 | 268 | 73 | 35 | 22 | 15 |
| 第二种情形下的最小抽样数量 | 147814 | 36954 | 16424 | 9239 | 5913 | 4106 | 3017 | 2310 | 1825 | 1479 | 370 | 165 | 93 | 60 |
| 第二种情形下的接收标准 | 14634 | 3622 | 1593 | 887 | 562 | 281 | 281 | 213 | 166 | 133 | 30 | 12 | 6 | 3 |



**(a)** 允许误差范围 $\epsilon$ 在 **0.001** 到 **0.005** 之间



**(b)** 允许误差范围 $\epsilon$ 在 **0.001** 到 **0.05** 之间

图 **2**　误差范围对最小检测量、接收/拒收标准的影响图

在 (1) 中，$n$ 为简单随机抽样方法下的所需要检测的最小检测量，$z_\alpha$ 为正态分布分位数对应单侧显著性水平 $\alpha$ 下的标准正态分布的临界值，$p_{true}$ 为二项分布中“成功”的概率，在本题情境下可以理解为真实次品率。$\epsilon$ 为最大误差范围。

在此种简单随机抽样的方法下，得到的抽样方案如表表 3所示。抽样方案中，四个指标量与允许误差范围的变化图如图 2所示。由图 2可知，各项指标量在允许误差范围在 0.001 到 0.002 时变化非常明显；当允许误差范围大于 0.002 时，各项指标量随允许误差范围的变化趋于平缓。抽样方案为：在第一种情形下，企业根据实际所允许的误差范围，选择不同的最小抽样数量，当样本中的次品数量达到拒收标准以上时，拒收这一批零配件；在第二种情形下，当样本中的次品数量在接收标准以内时，接收这一批零配件。

### 5.3 简单随机抽样方法的问题和优化思路

在本题的情境下，简单随机抽样方法的主要优点是易于理解和操作并适用于总体较小的数据，但也存在以下问题：

- 在公式 (1) 中，$p$ 应该为总体中的次品率，只有当 $p$ 为总体中的次品率时，所得的最

小检测量 $n$ 才是正确的。但是在本题中，$p$ 是未知的，容易出现高估或者低估的状况。如图 3所示，假如真实次品率 $p > 0.1$，608 个抽样并不能达到最小样本量的要求，从而导致对 $p$ 的估计出现较大偏差；假如真实次品率 $p < 0.1$，其所需的最小样本量又未达到 608 个，出现了抽样浪费。

- 简单随机抽样方法中，最小检测数量完全由抽样大小计算公式 (1) 得到，不能即时地对当前抽样结果进行实时反馈 [1]，从而加剧企业的检测成本。

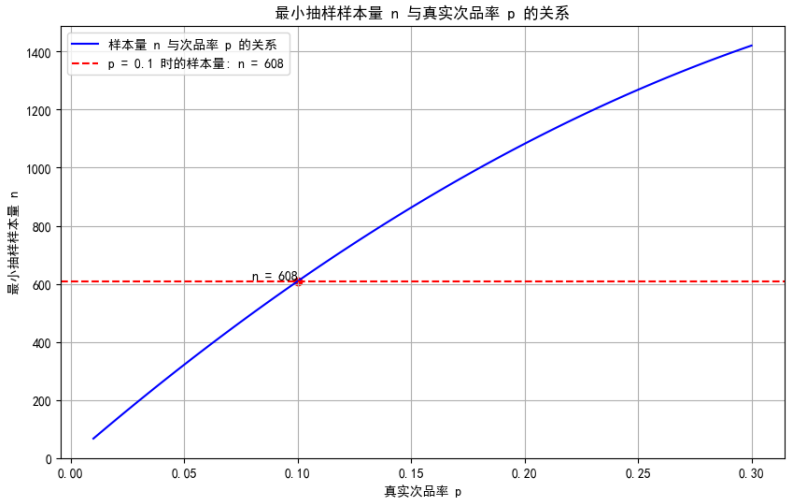



图 **3**　最小抽样样本量 $n$ 与真实次品率 $p$ 的关系

优化思路：针对样本量不足或样本量过大的问题，本文需要引入功效检验方法，同时降低第二类错误 $\beta$ 概率。此外，引入序贯抽样机制，并非一次性选取全部样本，而是分批逐步进行，根据每一步检测所获得的信息来决定是否停止抽样并接受或拒绝假设，从而达到更好的效果。

### 5.4 优化后的序贯抽样方法的模型建立

按照上述两步优化思路开始建立优化序贯抽样模型。第一步：引入功效检验 [5]，从而达到更好的最小样本估计效果，在上文中仅仅考虑了显著性水平 $\alpha$，即第一类错误发生的情况，而对第二类错误的发生没有做任何考虑，从而导致了在 $p$ 较大的情况下出现了样本取样不足的情况，这样的样本取样不足就是会导致第二类错误发生的概率大大增加。因此，此处引入功效检验，对公式（1）进行第一次修正：将 $z_\alpha$ 改为 $(z_\alpha + z_\beta)$，如式 (3) 所示。这样虽然会使样本抽样量增加，但却大大增加其第二类可信度。

$$n = \frac{(z_\alpha + z_\beta)^2 \cdot p_0(1 - p_0)}{\epsilon^2} \tag{3}$$

第二步：引入序贯抽样机制，对公式 (1) 进行第二次修正：将允许误差范围 $\epsilon$ 用真实的总体次品率 $p_{true}$ 与 $p_0$ 的差值进行替换，在抽样检测中，我们通常关注的并非一个

固定的误差 $\epsilon$，这种固定的允许误差范围可能无法反映真实的差异需求，而样本中实际次品率与标称次品率的差值才是被广泛需要的。因此调整公式，将 $\epsilon$ 修正为 $|p_{true} - p_0|$ 的形式，如 (4) 所示

$$n = \frac{(z_\alpha + z_\beta)^2 \cdot p_0(1-p_0)}{(p_{\text{true}} - p_0)^2} \tag{4}$$

这样它就会成为一个自适应误差范围 [3]，其自身会根据次品率的差异进行动态调整，如果标称次品率 $p_0$ 与实际次品率 $p_{true}$ 非常接近，那么误差范围会很小，这种情况下需要较大的样本量来精确估计这个差异。而如果两者差异很大，误差范围增大，所需样本量相应减少，从而提高检验效率。做出以上两点修正以后，本文的优化序贯抽样模型就已经初步建立，下面来简单介绍其整体流程：

### 5.4.1 贯序抽样方法的算法思路

贯序抽样方法的算法流程图如图 4所示。

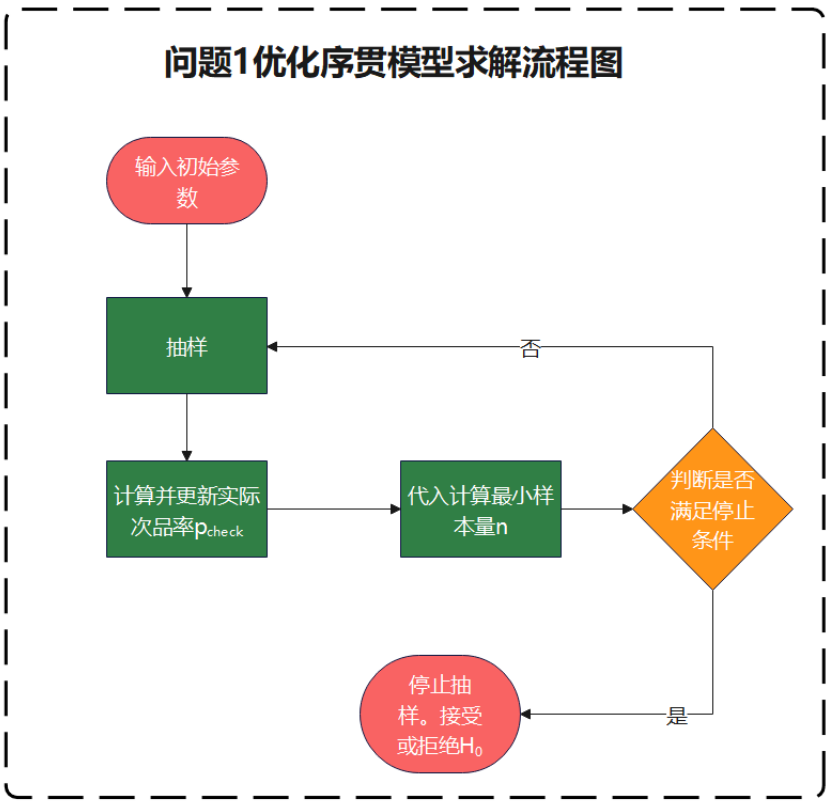



图 **4**　$\alpha$ 和 $\beta$ 不变情况下真实次品率 $p_{true}$ 与样本量 $n$ 的关系图

首先，输入初始参数 $\alpha$、$\beta$、初始抽样检测数量 $n_{start}$ 和次品率的标称值 $p_0$。抽样：进行单个抽样。计算并更新实际样本次品率当作 $p_{true}$，代入到公式 (4) 中的 $p_{true}$，计算 $p_{true}$ 所对应最小样本量，再把现有已抽样本量与 $p_{true}$ 所对应最小样本量进行对比，

判断是否满足停止条件，若满足则停止抽样并接受或拒绝假设 $H_0$；若不满足停止条件，则继续抽样

### 5.4.2 问题 1 两种情形的求解

固定 $\alpha = 0.05$、$\beta = 0.05$，可得达到置信度所需抽测样本量 $n$ 与真实次品率 $p_{true}$ 的关系图如图 5所示。固定 $\alpha = 0.1$、$\beta = 0.1$，可得达到置信度所需抽测样本量 $n$ 与真实次品率 $p_{true}$ 的关系图如图 6所示。

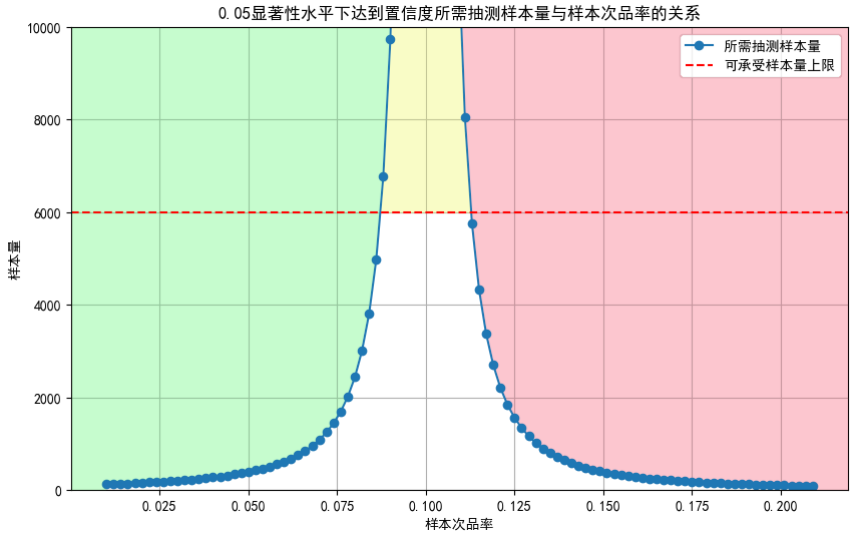



图 **5**　$\alpha$ 和 $\beta$ 都为 **0.05** 情况下（第一种情形）达到置信度所需抽测样本量与样本真实次品率 $p_{true}$ 的关系图

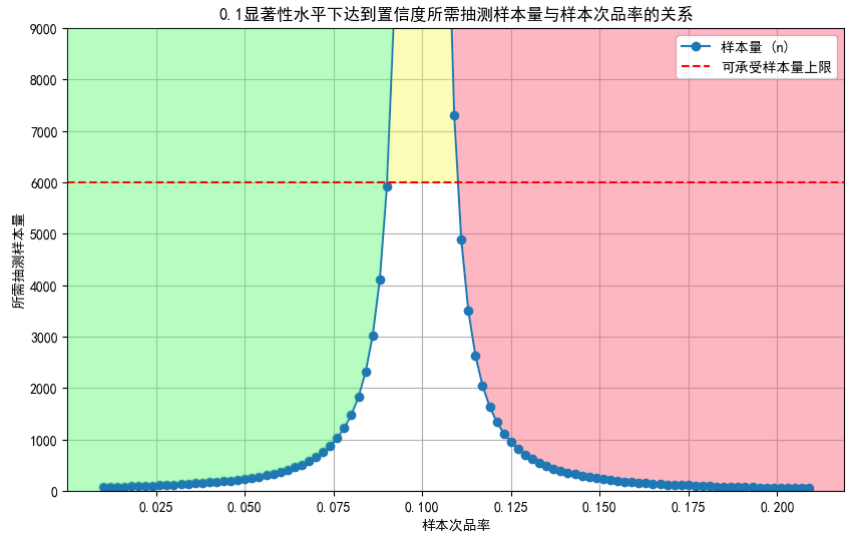



图 **6**　$\alpha$ 和 $\beta$ 都为 **0.1** 情况下（第二种情形）达到置信度所需抽测样本量与样本真实次品率 $p_{true}$ 的关系图

基于图 5和图 6本文做出以下抽样策略：为避免偶然性，首次抽样抽取 30 个样本，之后抽样运用贯序抽样的方法逐个抽样，并在每次抽样接受后都计算其样本次品率，如果以 (样本次品率, 样本量) 为坐标的点位于绿色区域则表示在相应置信度下接受其产品次品率小于 0.1，如果在红色区域则表示在相应置信度下拒绝接受其产品次品率小于 0.1，然而在要保证其置信度的情况下，其产品次品率及其接近 0.1 时，其所需抽取样本量会趋于无穷，这显然不符合公司利益，因此可以依据具体情况设置样本承受上限，以两图中 6000 为例，如果以 (样本次品率, 样本量) 为坐标的点进入黄色区域则停止抽样并依据具体情况做出具体决策，比如：为最大化公司利益黄色区域内均拒绝接受其产品次品率小于 0.1。

# 六、 问题 2 的模型建立与求解

## 6.1 不同决策方案对企业周期化生产的影响

在问题 2 的背景下，实际企业的生产运营周期中，可以将企业的生产周期简化为以下周期性过程：

1. 成品的第一个装配过程（涉及到零配件 1 和零配件 2 的检测决策）
2. 成品的第一个售卖过程（涉及到装配后成品的检测决策）
3. 成品的第一个售后过程（涉及到回收次品的拆解决策）
4. 成品的第二个装配过程（涉及到利用售后过程中回收到的零配件）
5. .....

企业需要做出的决策主要有以下四项：

1. 是否检测零配件 1
2. 是否检测零配件 2
3. 是否检测成品
4. 是否拆解回收次品

企业每装配一件成品，该成品的结局有：

- 该成品为非次品，被以市场售价售出，企业获利。
- 该成品为次品，被检验后废弃，企业受损。
- 该成品为次品，被检验后拆解，废物利用。
- 该成品为次品，未被检验就被售出，被买方调换后被企业废弃，企业受损。
- 该成品为次品，未被检验就被售出，被买方调换后被企业拆解，废物利用。

企业装配出的成品的以上几种结局均会影响企业的利润水平，可以采用求数学期望的思想，以企业装配出的单个成品的利润期望为目标函数，将企业装配出的单个成品的利润期望尽可能地优化上升。

企业在实际的周期性生产过程中，前一个生产周期的售后过程中的拆解次品环节会影响到下一个生产周期。同时，企业需要做出四项相互独立的决策，这四项决策会周期性地影响到企业的各个生产周期，进而影响企业利润。假设检测零配件 1 的决策变量为 $d_0$（$d_0 = 1$ 表检测，$d_0 = 0$ 表不检测），检测零配件 2 的决策变量为 $d_1$（$d_1 = 1$ 表检测，$d_1 = 0$ 表不检测），检测装配后成品的决策变量为 $d_2$（$d_2 = 1$ 表检测，$d_2 = 0$ 表不检测），拆解回收次品的决策变量为 $d_3$（$d_3 = 1$ 表拆解，$d_3 = 0$ 表不拆解）。

本文将这些相互独立的决策项数字化为了四个不同的决策变量，先逐步推导出企业装配出的单个成品的利润期望与上述决策变量的关系式，再遍历 16 种不同的决策方案的利润期望值，对比求出企业装配出的单个成品的利润期望最高的决策方案。

### 6.2 基于递推逼近的遍历优化模型的建立

装配成品时，只有零配件 1 和零配件 2 都合格时，才有可能生成合格成品。但即使零配件都合格，成品仍有概率是次品。假设 $P_{1n}$ 是在第 $n$ 个生产周期的装配过程中零配件 1 的实际总体次品率（即零配件 1 总体次品的比率）；$P_{2n}$ 是在第 $n$ 个生产周期的装配过程中零配件 2 的实际总体次品率；$P_{3n}$ 是在第 $n$ 个生产周期完成装配过程后，装配后成品的总体次品率；$P_{com}$ 是在零配件 1 和零配件 2 都正常的情况下，装配成品时仍不幸获得的次品的概率。则在任意第 $n$ 个生产周期中，有关系式 (5)。特殊地，在第 1 个生产过程中，有式 (6)。

$$P_{3n} = (1 - P_{com}) \times (1 - (1 - d_0) \times P_{1n}) \times (1 - (1 - d_1) \times P_{2n}) \tag{5}$$

$$P_{30} = (1 - P_{com}) \times (1 - (1 - d_0) \times P_{10}) \times (1 - (1 - d_1) \times P_{20}) \tag{6}$$

假设 $P_{success1n}$ 为：在第 $n$ 个生产周期中，装配后所得成品的总体成品率。其计算公式为 (7)。假设 $P_{success0}$ 为：在前后生产过程不相互影响的情况下（即无拆解次品的环节），装配过程后所得成品的总体成品率。$P_{success0}$ 计算公式为式 (8)。

$$P_{success1n} = (1 - P_{\text{com}}) \times (1 - (1 - d_0) \times P_{1n}) \times (1 - (1 - d_1) \times P_{2n}) \tag{7}$$

$$P_{success0} = (1 - P_{\text{com}}) \times (1 - (1 - d_0) \times P_{10}) \times (1 - (1 - d_1) \times P_{20}) \tag{8}$$

拆解后不合格成品中的零配件可以重复使用，上一个生产周期中的拆解操作会影响下一个生产周期中零配件的总体次品率。第 $n + 1$ 个生产过程的零配件 1 总体次品率 $P_{1(n+1)}$ 与第 $n$ 个生产周期中零配件 1 的总体次品率 $P_{1n}$ 的递推关系如 (9) 所示，第 $n + 1$ 个生产过程的零配件 2 总体次品率 $P_{2(n+1)}$ 与第 $n$ 个生产过程中零配件 1 的总体次品率 $P_{2n}$ 的递推关系如 (10) 所示。同时，可以得到 $P_{3n}$ 与 $P_{success1n}$ 和 $P_{success0}$ 的关系式 (11)。

$$P_{1(n+1)} = d_3 \times (P_{10} \times P_{\text{success1n}} + (1 - d_0) \times P_{1n}) + (1 - d_3) \times P_{10} \tag{9}$$

$$P_{2(n+1)} = d_3 \times (P_{20} \times P_{\text{success1n}} + (1 - d_1) \times P_{2n}) + (1 - d_3) \times P_{20} \tag{10}$$

$$P_{3n} = d_3 \times (1 - P_{success1n}) + (1 - d_3) \times (1 - P_{success0}) \tag{11}$$

由上述式 (12) 到式 (17)，本文构建出了该企业在任意决策情况下，生产过程中各个量的递推关系公式。虽然在不同的生产过程中，随着 $n$ 的变化，各个量也在变化，但当 $n$ 趋向无穷大时，生产过程中的各个量（包括需要求解的企业装配出的单个成品的利润期望）会趋向稳定，而生产过程中利润期望量趋于稳定的标志就是迭代到某两个生产周期的利润期望差距极小。通过这样的递推逼近，可以得出不同决策方案下利润的稳定值。

购买成本方面，假设第 $n$ 个生产周期中装配一件成品的平均总成本为 $C_{buy(n)}$，零配件 1 和零配件 2 的单个购买成本分别为 $C_1$ 和 $C_2$，则 $C_{buy(n)}$ 的计算公式为式 (12)。

$$C_{buy(n)} = d_3 \times (C_1 \times P_{success0} + C_2 \times P_{success0}) + (1 - d_3) \times (C_1 + C_2) \tag{12}$$

检测成本方面，假设第 $n$ 个生产周期中装配一件成品的平均总检测成本为 $C_{detection(n)}$，对单个零配件 1、单个零配件 2、单个成品的检测成本分别记为 $Ch_1$、$Ch_2$、$Ch_3$。则总的检测成本 $C_{detection(n)}$ 的计算公式为式 (13)。

$$C_{\text{detection}(n)} = Ch_1 \times d_0 + Ch_2 \times d_1 + \frac{P_{1n}}{1 - P_{10}} \times (C_1 + Ch_1) \times d_0 + \frac{P_{2n}}{1 - P_{20}} \times (C_2 + Ch_2) \times d_1 \tag{13}$$

调换成本方面，假设第 $n$ 个生产周期中装配一件成品的平均总调换成本 $C_{det_{ret}(n)}$，其中单个不合格成品的调换费用为 $C_{change}$。则总调换成本 $C_{det_{ret}(n)}$ 的计算公式为式 (14)。

$$C_{det_{ret}(n)} = Ch_3 \times d_2 + (1 - d_2) \times C_{change} \times P_{3n} \tag{14}$$

假设第 $n$ 个生产周期中，装配一件成品的平均总的拆解费用为 $C_{totaldis(n)}$，单个次品的拆解成本为 $C_{dis}$，那么 $C_{totaldis(n)}$ 的计算公式为式 (15)。

$$C_{totaldis(n)} = d_3 \times P_{3n} \times C_{dis} \tag{15}$$

假设第 $n$ 个生产周期中，装配一件成品的平均总成本为 $C_{total(n)}$，装配一个成品所需要的成本为 $C_{com}$，则总成本的计算公式为 (16)。

$$C_{total(n)} = C_{det_{ret}(n)} + C_{buy(n)} + C_{detection(n)} + C_{com} + C_{totaldis(n)} \tag{16}$$

$Profit(n)$ 第 $n$ 个生产周期中装配出一个成品企业所获得利润的数学期望。$Price$ 为单个成品的市场售价，装配出一个成品的利润期望计算公式为式 (17)。

$$Profit(n) = Price \times P_{success1n} - C_{total(n)} \tag{17}$$

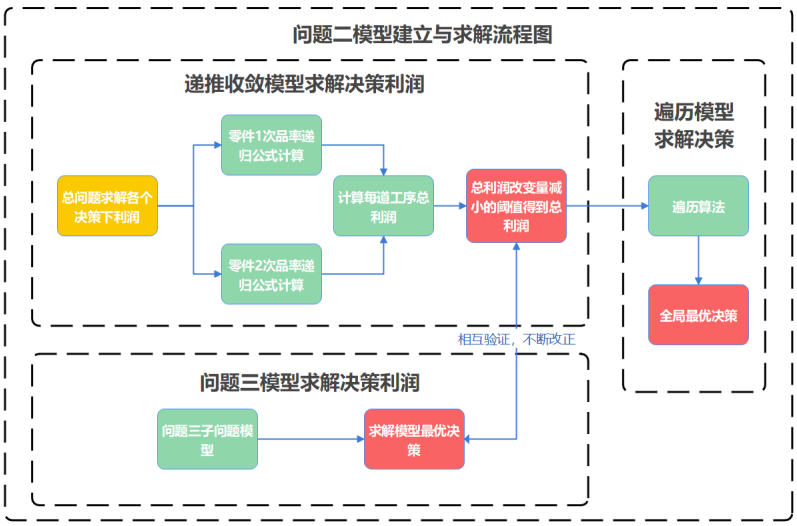



图 **7**　问题 **2** 模型的建立与求解流程图

当 $n \to \infty$ 时，$Profit(n) \to A$，其中 $A$ 是一个常数，即在该决策方案下企业利润的稳定值。每一种情况下，都有 16 种不同的决策方案，本文可以计算出这 16 种不同的决策方案的利润稳定值，通过对比求取出利润最高时的决策方案。问题 2 模型的建立与求解流程图如图 7 所示。

### 6.3 问题求解

已知 $P_{10}$、$C_1$、$Ch_1$、$P_{20}$、$C_2$、$Ch_2$、$P_{com}$、$C_{com}$、$Ch_3$、$Price$、$C_{dis}$ 等量，将其带入到基于迭代逼近的遍历优化模型中进行遍历优化求解，求解结果如表 4所示。

本文也用了问题 3 中的子问题求解小模型进行验证，得到的结果与本问题种基于爹太逼近的遍历优化模型结果一致，互为验证。

表 **4**　最佳决策与最大利润

| 情况 | 是否检测零配件 1 | 是否检测零配件 2 | 是否检测零成品 | 是否对次品进行拆解 | 最大单个成品的利润期望 |
|---|---|---|---|---|---|
| 1 | **1（是）** | **1（是）** | **0（否）** | **1（是）** | **15.80** |
| 2 | **1（是）** | **1（是）** | **0（否）** | **1（是）** | **8.60** |
| 3 | **1（是）** | **1（是）** | **1（是）** | **1（是）** | **14.89** |
| 4 | **1（是）** | **1（是）** | **1（是）** | **1（是）** | **14.25** |
| 5 | **0（否）** | **1（是）** | **1（是）** | **0（否）** | **11.86** |
| 6 | **0（否）** | **0（否）** | **1（是）** | **0（否）** | **19.84** |

# 七、 问题 3 的模型建立与求解

## 7.1 分治算法思想的运用与处理

当企业的生产用到了 $m$ 道工序、$n$ 种零配件时，那么相应的决策变量主要是在生产周期的不同过程中，是否对零配件、半成品、成品进行检测或拆解。每个决策变量都会影响生产成本、次品率以及最终利润。同时，从零配件到成品，要经过：各种零配件 $\rightarrow$ 各种半成品 $\rightarrow$ 最终成品 。此时，可以运用分治算法的思想，将从零配件到成品的生产大问题分解为多个子问题，每种子问题可以视为一种类似问题 2 的单个装配过程（如用零配件生产出某个半成品、用半成品生产出成品）。对每种子问题分别求解后再合并子问题的解，进而求出目标函数———企业装配一件成品所能获得的利润期望 $Profit(n)$ 与各个决策变量、成本量、次品率的关系。

在有 2 道工序、8 个零配件且零配件到成品的关系如题目中图 1 所示的具体情况下，企业需要做出的决策一共有 16 项：是否对某种零配件进行检测（8 项）、是否对某种半成品进行检测（3 项）、是否对某种不合格半成品进行拆解（3 项）、是否对成品进行检测（1 项）、是否对不合格成品进行拆解（1 项）。利用问题 2 的方法，假设是否检测第 $i$ 种零配件的决策变量为 $x_i$（$x_i = 1$ 记为检测，$x_i = 0$ 记为不检测），是否检测第 $i$ 中半成品的决策变量为 $y_i$（$y_i = 1$ 记为检测，$y_i = 0$ 记为不检测），是否拆解第 $i$ 种不合格半成品的决策变量为 $y_i'$（$y_i' = 1$ 记为拆解，$y_i' = 0$ 记为不拆解），是否对成品进行检测的决策变量为 $z$（$z = 1$ 记为检测，$z = 0$ 记为不检测），是否对不合格成品进行拆解的决策变量为 $z'$（$z' = 1$ 记为拆解，$z' = 0$ 记为不拆解）。那么，决策方案可以用 [$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8), (y_1, y_2, y_3), (y_1', y_2', y_3'), (z), (z')$]

由题中所给信息，零配件 1 到零配件 8 的次品率记为 $p_1$、$p_2$、$p_3$ ... $p_8$，当零配件没

有问题时，装配半成品 1、半成品 2、半成品 3 时得到不合格半成品的次品率分别记为 $p_9$、$p_{10}$、$p_{11}$，当半成品没有问题时，装配成品时得到不合格成品的次品率记为 $p_{12}$。

## 7.2 基于分治算法的遗传退火模型的建立

### 7.2.1 分治算法子问题模型建立

本问题经分治后，可以分为以下四个子问题：

- 用零配件 1、零配件 2、零配件 3 装配半成品 1
- 用零配件 4、零配件 5、零配件 6 装配半成品 2
- 用零配件 7、零配件 8 装配半成品 3
- 用半成品 1、半成品 2、半成品 3 装配成品

因为这四个子问题的过程是相似的，本文重点讨论用零配件 1、零配件 2、零配件 3 装配半成品 1 这一子问题。本文的目标函数是企业装配出一件成品后所获得的利润期望，在子问题中，各量都是假设企业产出单位 1 的子问题中成品（如本文所重点讨论的子问题中成品就是半成品 1）所需要的量。

由于每个装配过程中，拆解环节都会影响下一个装配过程中相应零配件的次品率。由是设:

- $P_{X_i\text{ru}(n)}$ 为零配件 $i$ 在第 $n$ 个装配过程中检测之前的次品率
- $P'_{X_i\text{bu}(n)}$ 为零配件 $i$ 在第 $n$ 个装配过程中拆解情况下的 $P_{X_i\text{ru}(n)}$
- $P_{X_i\text{bu}(n)}$ 为零配件 $i$ 在第 $n$ 个装配过程中补货的次品率
- $P_{X_i\text{chu}(n)}$ 是零配件 $i$ 在第 $n$ 个装配过程中经过检测环节后零配件的次品率
- $P_{Y_i\text{ru}(n)}$ 是各个子问题中组装体 $i$ 在第 $n$ 个装配过程中检测之前的次品率
- $P_{Y_i\text{zhuang}}$ 是组装体 $i$ 的组装失败率
- $X_{i\text{ruzong}(n)}$ 为零配件 $i$ 在第 $n$ 个装配过程的装配需要数
- $X_{i\text{chuzong}(n)}$ 为零配件 $i$ 在第 $n$ 个装配过程中经检测状态后的数量

其中，补货指的是在每个装配过程中需要的零配件来满足装配需要，其中补货可以源自于上一个装配过程中的拆解也可以来自于供应商。对于 $P_{X_i\text{chu}(n)}$，若有检测，则 $P_{X_i\text{chu}(n)}=0$；若无检测，则 $P_{X_i\text{chu}(n)}=P_{X_i\text{ru}(n)}$。可以得到 $P_{X_i\text{chu}(n)}$ 的表达式 (21):

$$P_{X_i\text{chu}(n)}=(1-x_i)P_{X_i\text{ru}(n)} \tag{18}$$

对于 $X_{i\text{chuzong}(n)}$，若没有检测环节，$X_{i\text{ruzong}(n)}=X_{i\text{chuzong}(n)}$；若有检测环节，则 $X_{i\text{chuzong}(n)}$ 就是 $X_{i\text{ruzong}(n)}$ 中的合格数量。可以得到 $X_{i\text{ruzong}(n)}$ 的表达式 (19):

$$X_{i\text{ruzong}(n)}=\frac{X_{i\text{chuzong}(n)}}{x_i(1-P_{X_i\text{bu}(n)})+(1-x_i)} \tag{19}$$

对于 $P'_{X_i\text{bu}(n)}$，在进行拆解时第 n+1 个装配过程时其零配件的次品来自于两个部分：一是进货里边的次品，二是上第 n 次装配过程中拆解下来的次品且拆解下来的次品数就等于上第 n 次装配过程中的相应零配件检测后流向装配体的次品总数。由是可以得到 $P'_{X_i\text{bu}(n)}$ 的表达式 (20)。

$$P'_{X_i\text{bu}(n+1)}=\frac{\left[P_{X_i\text{bu}(n+1)}\left(X_{i\text{ruzong}(n+1)}-(X_{i\text{chuzong}(n)}-1)\right)+(1-x_i)P_{X_i\text{chu}(n)}X_{i\text{chuzong}(n)}\right]}{X_{i\text{ruzong}(n+1)}} \tag{20}$$

对于 $P_{X_i\text{ru}(n+1)}$，在拆解的情况下其等于 $P'_{X_i\text{bu}(n+1)}$，而在拆解的情况下其应该等于 $P_{X_i\text{bu}(n+1)}$ 即，由此可以得到 $P_{X_i\text{ru}(n+1)}$ 的表达式 (21)：

$$P_{X_i\text{ru}(n+1)}=(1-y'_1)P_{X_i\text{bu}(n+1)}+y'_1P'_{X_i\text{bu}(n+1)} \tag{21}$$

对于 $P_{Y_i\text{ru}(n)}$，其就应该等于 1 减去组装成功率与各个对应零配件输出的非次品率，由此可以得到 $P_{Y_i\text{zhuang}}$ 的表达式 (22)，其中 X 是子问题中零配件的编号集合:

$$P_{Y_i\text{ru}(n)}=1-(1-P_{Y_i\text{zhuang}})\prod_{j\in X}\left[1-(1-x_j)P_{X_j\text{chu}(n+1)}\right] \tag{22}$$

对于 $X_{i\text{chuzong}(n+1)}$，在装配体不检测的情况下其应该等于需要输出的组装体的量即为单位 1，在黄佩提检测的情况下其应等于单位 1 与装配体检测前的非次品率的比值，由此可以得到 $X_{i\text{chuzong}(n+1)}$ 的表达式 (23)：

$$X_{i\text{chuzong}(n+1)}=1-y_i+\frac{y_i}{1-P_{Y_i\text{ru}(n)}} \tag{23}$$

由式 (20) 到式 (23)，可以得到第 n 个装配过程到第 n+1 个装配过程的推导，由于在 n 趋于无穷时其一定会区域定值以达到稳态，该稳态的各个参数与最终成本最终决定了该决策优劣，且在稳态时各个参数量不再改变，本文直接另式中第 n+1 个装配过程中的量等于第 n 个装配过程中的量，这样就得到了方程组 (24)：

$$\begin{cases}P'_{X_i\text{bu}(n)}=\frac{\left[P_{X_i\text{bu}(n)}\left(X_{i\text{ruzong}(n)}-(X_{i\text{chuzong}(n)}-1)\right)+(1-x_i)P_{X_i\text{chu}(n)}X_{i\text{chuzong}(n)}\right]}{X_{i\text{ruzong}(n)}},\\ P_{X_i\text{ru}(n)}=(1-y'_1)P_{X_i\text{bu}(n)}+y'_1P'_{X_i\text{bu}(n)},\\ P_{Y_i\text{ru}(n)}=1-(1-P_{Y_i\text{zhuang}})\prod_{j\in X}\left[1-(1-x_j)P_{X_j\text{chu}(n)}\right],\\ X_{i\text{chuzong}(n)}=1-y_i+\frac{y_i}{1-P_{Y_i\text{ru}(n)}}.\end{cases} \tag{24}$$

同一子问题下要想成功组装则其 $X_{i\text{chuzong}(n)}$ 均应相等，由此得到式子 (25)

$$X_{1\text{chuzong}(n)}=X_{2\text{chuzong}(n)}=...=X_{k\text{chuzong}(n)} \tag{25}$$

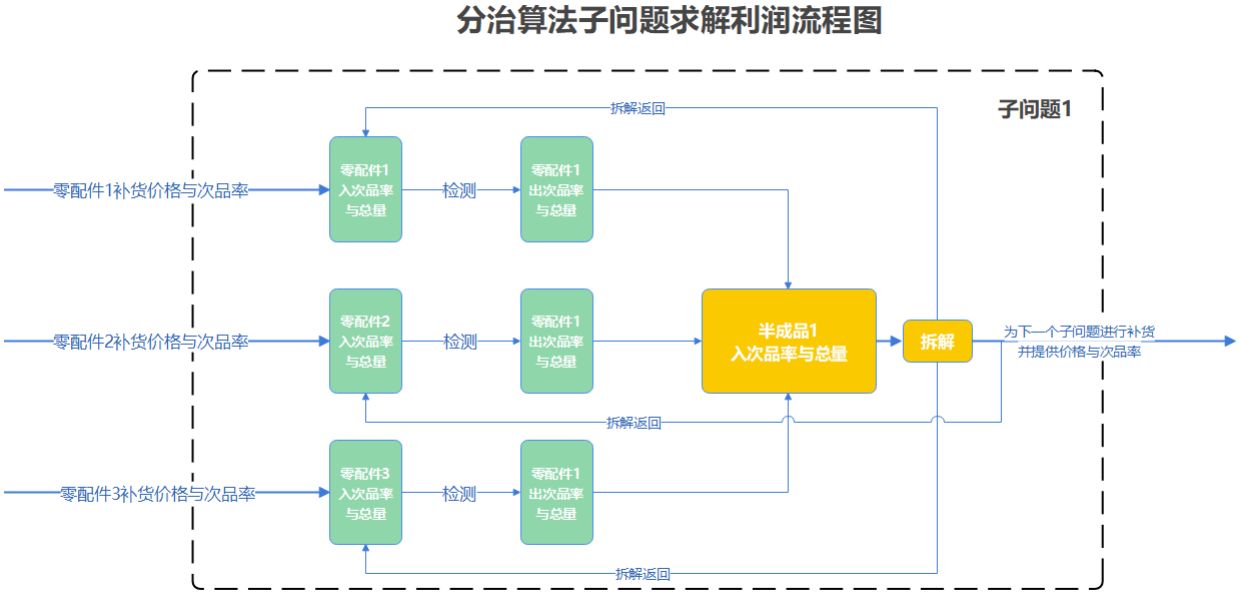



图 **8**　分治算法求解子问题流程图

最终联立方程 (25) 与子问题中所有零配件的方程组 (24), 求解该即得到本子问题组合最终稳态参数。子问题求解流程图见图 8。

### 7.2.2 分治算法求解总问题的模型和遗传退火算法的应用

记生产半成品 1、半成品 2、半成品 3、成品的子问题分别为子问题 1、子问题 2、子问题 3、子问题 4。其中子问题 1、子问题 2、子问题 3 为 1 级子问题、子问题 4 为 2 级子问题。子问题分配图如图 9 所示。

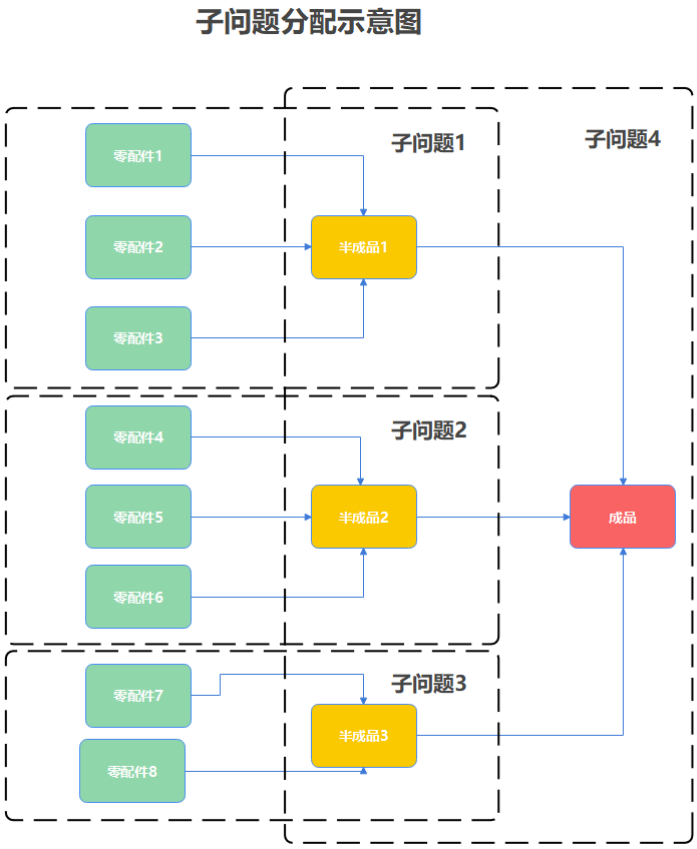



图 **9**　子问题的分配图

对于任意子问题，不妨归一化认为该子问题的关系是零件装配成成品，现假设 $y'$ 为该子问题下的成品拆解与否的决策变量，$X_i$ 为该子问题下所需要的某种零配件，$Y$ 为该子问题下的成品。有以下量：

- $X_{i(zongru)}$ 表示某种零件总共需要的量
- $X_{(zongchu)}$ 表示某种零件经检测后输出的量
- $X_{i(gou)}$ 表示某种零件的购买单价
- $X_{i(cost)}$ 为生产某种成品一件的量所需要的某种零件的总花费（包括购买和检测的费用）
- $X_{chai}$ 某种零件拆解返回的数量
- $X_{i(jian)}$ 表示某种零件的检测成本
- $Y_{i(zhuang)}$ 表示某种零件的装配成本
- $Y_{(chai)}$ 表示某种零件的拆解成本
- $Y_{(cost)}$ 表示该决策下生产一件成品所需要的总花费

$$X_{chai} = y'(X_{(zongchu)} - 1) \tag{26}$$

$$X_{i(cost)} = X_{i(gou)}(X_{i(zongru)} - X_{chai}) + x_i X_{i(zongru)} X_{i(jian)} \tag{27}$$

$$Y_{(cost)} = Y_{i(zhuang)} X_{(zongchu)} + y' Y_{(chai)} X_{chai} + \sum_{i=k} X_{i(\text{cost})} \tag{28}$$

最终算出 $Y_{i(cost)}$，作为下一级子问题的相应零件的进货单价。如此递推，最终递推到成品（最后一个子问题），解决最后的子问题得到成品的次品率 $P_{final}$、成品的生产成本 $C_{final}$、再利用最终成品的检测成本 $Ch_{final}$ 与调回成本 $C_{recall}$，记 $Price$ 为单个成品的市场售价、$z_1$ 为最终成品的检验的决策变量。

$$Profit = Price(1 - P_{final}) - z_1 * Ch_{final} - (1 - z_1) * (C_{recall} * P_{final}) \tag{29}$$

分治算法的核心在于利润计算时子问题之间的结合，，即在上一级子问题的基础上求解下一级子问题，上一级子问题的输出将作为下一级子问题相应零件的供货，因此在上一级子问题解决的基础上，可以求出下一级子问题相应零件的供货单价以及供货次品率，在确定了下一级子问题的供货单价和供货次品率后，整个系统的利润计算可以通过逐步递推的方式完成。具体而言，每一级子问题的解决都会影响最终成品的成本和质量，从而影响总利润。以分治算法模型求解利润的流程图如图 10 所示。

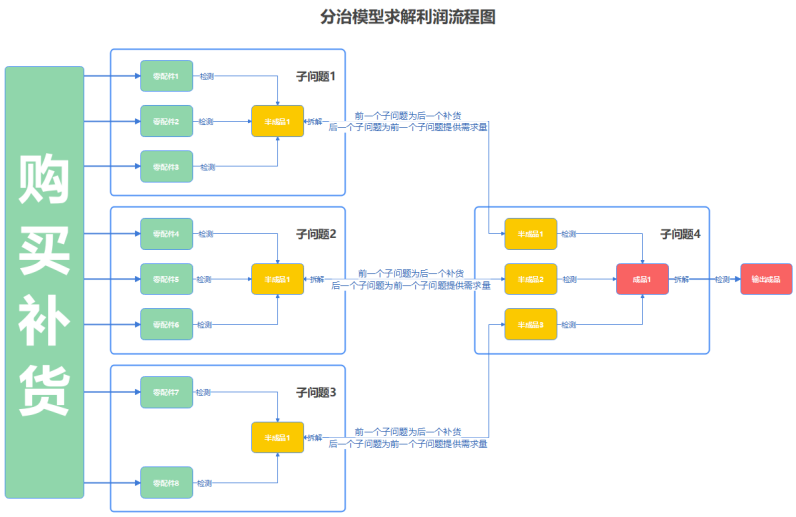



图 **10**　分治算法模型求解利润的流程图

分治模型将总问题拆解成多个子问题，计算各自的成本和利润。随后，这些子问题的求解结果进入遍历模型，通过穷举所有可能的决策组合，找到全局最优解。然而单靠遍历算法无法应对复杂度，因此引入了遗传算法与模拟退火混合算法。该算法通过退火改善初始解，再利用遗传算法逐步优化解集，最终在相互对比中找到最稳定的最优

决策。整个过程从局部优化到全局搜索，再到智能调整，确保在复杂问题中得到最佳方案。问题 3 基于分治算法的遗传退火模型求解的流程图如图 11 所示。

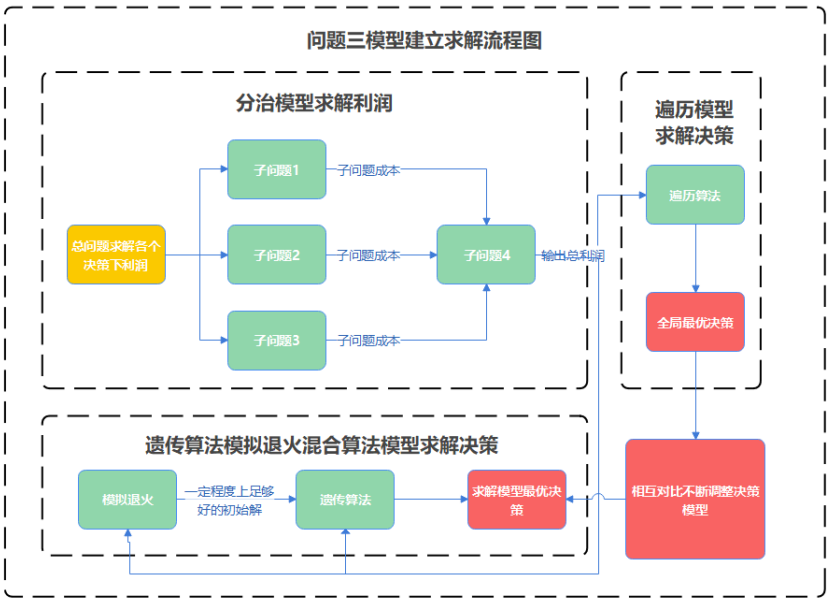



图 **11**　问题 **3** 基于分治算法的遗传退火模型求解的流程图

### 7.3 问题求解

最终得到问题 3 的最优决策为 [(1,1,0,0,1,0,1,1),(0,1,1),(0,1,1),(1),(0)]。在该种决策方案的情况下，企业每装配一个成品所获得的最大利润期望为：57.032 元。

最后的决策方案为：检测零配件 1、检测零配件 2，不检测零配件 3，不检测零配件 4，检测零配件 5，不检测零配件 6，检测零配件 7，检测零配件 8，不检测半成品 1，检测半成品 2，检测半成品 3，不拆解不合格的半成品 1，拆解不合格的半成品 2，拆解不合格的半成品 3，检测最终成品，不拆解不合格的成品。在该最优决策方案下，企业每生产一件成品，最后可以获得的最大利润期望约为 57.032 元。

## 八、 问题 4 的模型建立与求解

问题 4 在第二与第三问的基础上做了进一步的变化，结合了第一问。在第四问中，零配件，半成品与成品的次品率并非是其真实值，而是其通过抽样方法估计所得，本文考虑用贝叶斯公式和蒙特卡洛模拟的方法来求解本问题。求解流程图如图 12。

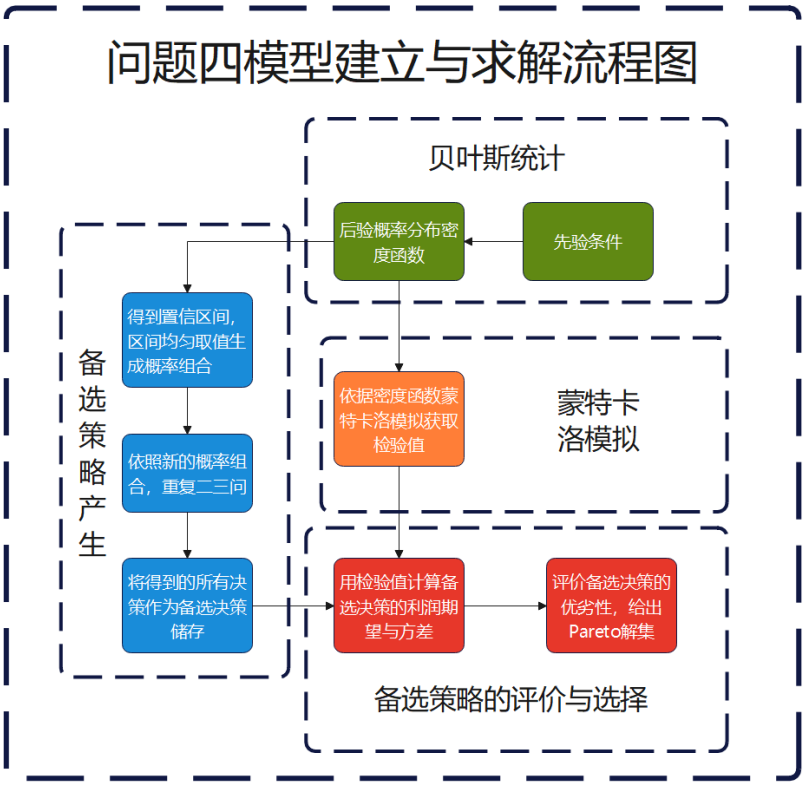



图 **12**　问题 **4** 求解流程图

### 8.1 基于贝叶斯公式的模型修改思想

本文在此基础上根据贝叶斯公式 (30) 做了贝叶斯统计检验。抽样所得的次品率作为先验概率 $P(A)$，通过贝叶斯估计的方式来计算其后验概率 $P(A|B)$。

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{30}$$

由于原分布属于二项分布，其先验概率的分布假设应当满足与二项分布的共轭分布，即式 (31) 贝塔（Beta）分布。，这样可以使得后验分布也满足于贝塔（Beta）分布，从此本文推得了后验分布的概率，该概率可用作两个步骤：（1）估计后验分布的 95% 置信区间，以此来获取可能决策选择；（2）产生随机样品用于蒙特卡洛模拟，以此获取不同决策的期望与方差进行最终决策的确定。

$$f(\theta;\alpha,\beta) = \frac{\theta^{\alpha-1}(1-\theta)^{\beta-1}}{B(\alpha,\beta)} \tag{31}$$

### 8.2 问题 2 的模型修正和重新求解

#### 8.2.1 通过贝叶斯公式获取“候选策略”

本文通过贝叶斯统计估计后验分布的 95% 置信区间，以此来获取“候选策略”，次品率的后验分布图如图 13。

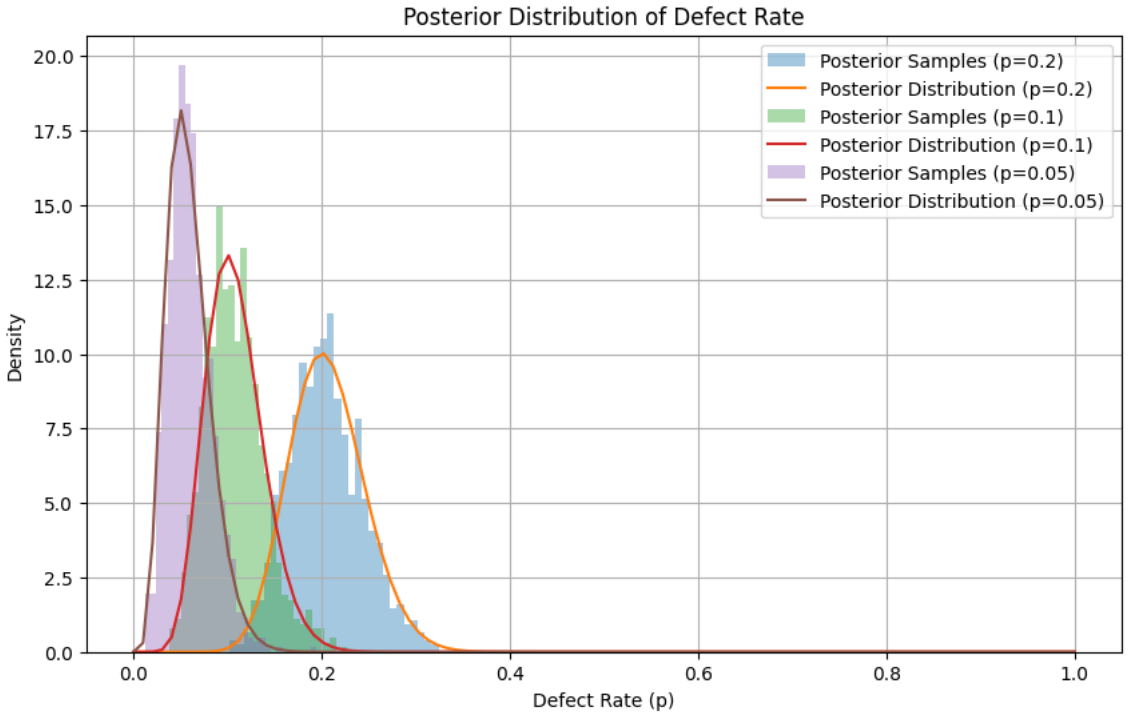



图 **13**　次品率的后验分布图

本文为控制变量，将先验样本值固定为 100，该样本值是低于第一问中大多数所需样本值的，这样可以获得更大的置信区间，这样讨论所得的鲁棒性也会更加可靠。由于题目中出现了三类次品率：0.05,0.1,0.2，在此也仅以这三个次品率举例：

1. 次品率为 0.05 的 95.0% 置信区间: [0.0221, 0.1118]
2. 次品率为 0.1 的 95.0% 置信区间: [0.0556, 0.1746]
3. 次品率为 0.2 的 95.0% 置信区间: [0.1336, 0.2891]

本文将这三个置信区间等距离划分后取值（实现代码中每个区间等间距取了 20 个值），这样对零配件一、零配件二与成品，都会获得与自身检验概率（0.05 或 0.1 或 0.2）相对应的可能真实次品率值，将这 $20 \times 20 \times 20$ 共构成的 8000 种组合分别带入第二问的计算中得到最优化的决策方案，可以发现：

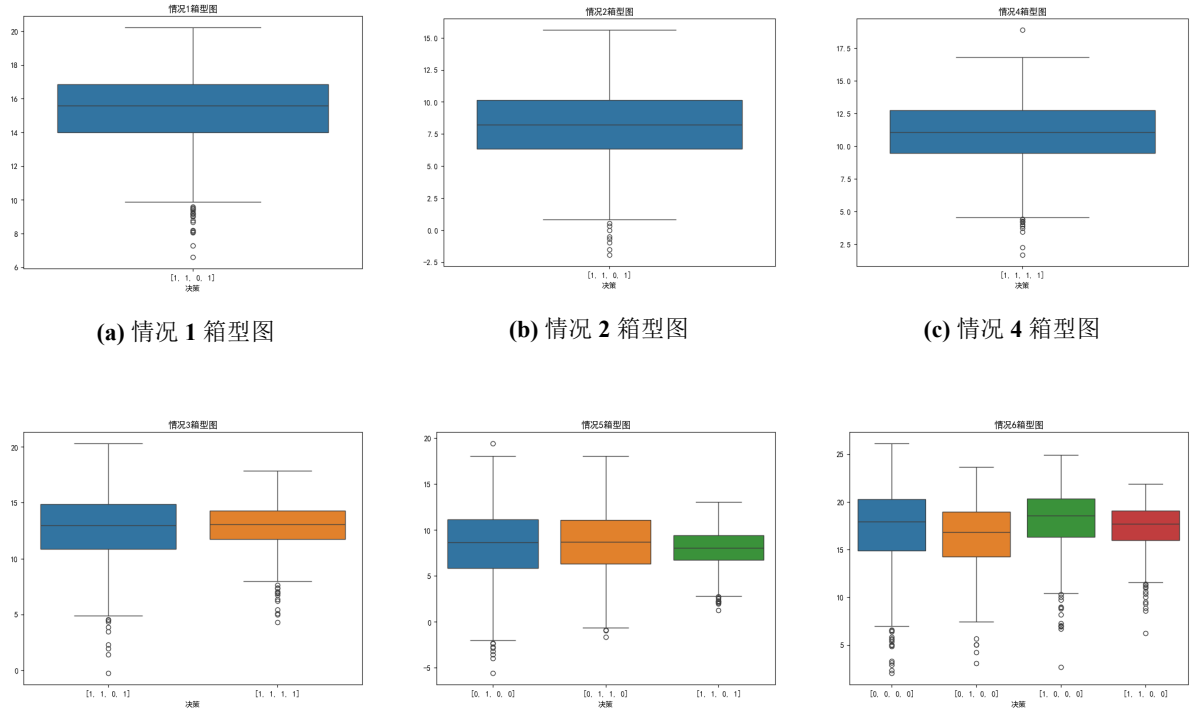
- 在情况一、情况二、情况四下：情况一：决策方案为 $[1,1,0,1]$（即决策方案为：检验零配件 1、检测零配件 2、不检测成品、对次品进行拆解）；情况二：决策方案为 $[1,1,0,1]$（即决策方案为：检验零配件 1、检测零配件 2、不检测成品、对次品进行拆解）；情况四：决策方案为 $[1,1,1,1]$（即决策方案为：检验零配件 1、检测零配件 2、检测成品、对次品进行拆解）。次品率的改变不会影响最优决策，仅有利润会随次品率的改变而一定程度的身高或者下降，这可以说明当下的最优决策在该情况的条件下具有相当好的鲁棒性。
- 在情况三、情况五、情况六中，次品率的变化引起了最优决策的改变，即认为最优决

策出现了多个“候选策略”。情况三：决策 $[1,1,1,1]$，决策 $[1,1,0,1]$；情况五：决策 $[0,1,1,0]$，决策 $[0,1,0,0]$ 和决策 $[1,1,0,1]$；情况六：决策 $[0,0,0,0]$、决策 $[0,1,0,0]$、决策 $[1,0,0,0]$、决策 $[1,1,0,0]$。

然而，在生产过程当中，需要我们在一开始就选择一个最合适的策略，因此，对这些“候选策略”进行一定程度上的评估是重要的。本文主要从两个方面展开评估：该决策对于所有情况的利润期望、该决策在所有情况下的综合方差。综合考虑这两个值，在实际意义上，就是综合考虑企业获利的大小与稳定性。这两点毫无疑问都是企业决策的重中之重。

### 8.2.2 基于蒙特卡洛模拟的最终决策确定

本文用蒙特卡洛模拟产生随机样品，以此获取不同决策的期望与方差进行最终决策的确定。为了更接近于实际情况，利用了上述所得到的 Beta 后验概率密度函数来随机生成 1000 个待测样本。对于每一种情况，本文从上一步得到的“候选策略”中选取其中一个，对生成的 1000 个待测样本进行计算，从而获得相应的利润。综合所有的利润，我们就可以获得利润期望与方差，并绘出箱型图如图 14所示。




**(a)** 情况 **1** 箱型图　**(b)** 情况 **2** 箱型图　**(c)** 情况 **4** 箱型图

**(d)** 情况 **3** 箱型图　**(e)** 情况 **5** 箱型图　**(f)** 情况 **6** 箱型图

图 **14**　问题 **2** 不同情况下的利润期望与方差和决策方案相关的箱型图

可以看到，不同的“候选策略”之间也会有优劣之分。例如在情况 3 下：决策 $[1,1,1,1]$ 的利润期望 12.786 大于决策 $[1,1,0,1]$ 的利润期望 12.666，但同时决策 $[1,1,1,1]$ 的方

差 4.211 却又小于决策 $[1,1,0,1]$ 的方差 10.015。这表明：在大规模制造的情况下，决策 $[1,1,1,1]$ 在企业获利方面无论是大小还是稳定性都是比决策 $[1,1,0,1]$ 更优的一方，决策 $[1,1,0,1]$ 仅可能在小规模的制造中占于上风，观察箱型图又发现，决策 $[1,1,0,1]$ 的最低获利远低于决策 $[1,1,1,1]$ 的最低获利，其值甚至降低到了负数，这又表明了：在小规模制造中决策 $[1,1,0,1]$ 需要承担大量的风险，甚至出现企业亏损的行为。由此可见，在情况三下决策 $[1,1,1,1]$ 对决策 $[1,1,0,1]$ 的优越性是支配性的。

然而，有些解之间的支配关系则并不存在，若将此处看做一个多目标优化，则这几个非劣解就可以组成其 Pareto 解集，例如情况六中的决策 $[1,0,0,0]$ 与决策 $[1,1,0,0]$，前者具有更高的利润期望，但相应的方差较大，最低值也较小，是牺牲了稳定性从而获取更高利润，属于高风险高回报，而后者虽然利润期望较小，但胜在稳定，利于企业的持续性发展，这两者的决策可供企业以自身目标与偏好选用。在问题 4 的新背景下，不同情况下可供选择的决策及其可以获得的利润期望与利润方差如表 5所示。

表 **5**　不同情况下的决策数据表

| 情况 | 是否检验零配件 1 | 是否检验零配件 2 | 是否检验成品 | 是否对不合格成品进行拆解 | 最佳决策收益期望 | 最佳决策收益方差 |
|---|---|---|---|---|---|---|
| 1 | **1（是）** | **1（是）** | **0（否）** | **1（是）** | **15.163** | **4.704** |
| 2 | **1（是）** | **1（是）** | **0（否）** | **1（是）** | **8.076** | **7.768** |
| 3 | **1（是）** | **1（是）** | **1（是）** | **1（是）** | **12.786** | **4.211** |
| 4 | **1（是）** | **1（是）** | **1（是）** | **1（是）** | **10.979** | **6.134** |
| 5 | **0（否）** | **1（是）** | **1（是）** | **0（否）** | **8.684** | **10.691** |
| | **1（是）** | **1（是）** | **0（否）** | **1（是）** | **8.056** | **4.124** |
| 6 | **1（是）** | **0（否）** | **0（否）** | **0（否）** | **18.115** | **9.355** |
| | **1（是）** | **1（是）** | **0（否）** | **0（否）** | **17.405** | **4.979** |

由表 5和图 14可知，对于原本问题 2 的情况 1、情况 2、情况 4 的最优决策，问题 4 将真实次品率改为抽样检测所得次品率的改变并没有引起最优决策的改变，这是因为原有的最优决策的鲁棒性较好。情况 3 中出现了另外一个“候选决策”$[1,1,1,1]$，且该候选决策对原本最优决策的优越是具有优越性的，因此调整情况 3 的最优决策为 $[1,1,1,1]$。情况 5 和情况 6 也出现了“候选决策”，真实次品率改为抽样检测所得次品率后情况 5 的决策方案为：若企业追求高风险高回报，那么决策方案为 $[0,1,1,0]$；若企业追求稳健收益，那么决策方案为 $[1,1,0,1]$。情况 6 的决策方案为：若企业追求高风险高回报，那么决策方案为 $[1,0,0,0]$；若企业追求稳健收益，那么决策方案为 $[1,1,0,0]$。

### 8.3 问题 3 的模型修正与重新求解

问题 3 模型的修正步骤与上文所叙完全相同，均是先通过对其置信区间的划分取值求得“候选策略”，再通过蒙特卡洛模拟，由利润期望与方差等条件来判断各个策略的优劣性。此处不多赘述，重点讲述结果的分析。（此处可附上结果）。

从结果中发现：大多数在某一特定次品率下所寻得的具有高利润的最优策略在鲁棒性上的表现都很差，反倒是一些利润较低的策略却呈现出相当好的鲁棒性，经过对所有候选策略的检验，发现仅有三个决策方案是满足利润期望为正值的，分别是**决策 1 [(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1), (1)]**、**决策 2[(1, 1, 1, 1, 1, 1, 1, 0), (1, 1, 1), (1, 1, 0), (1), (1)]** 与**决策 3[(1, 1, 1, 1, 1, 0, 1, 1), (1, 1, 1), (1, 0, 1), (1), (1)]**，这三种决策方案的收益箱型图如图 15所示。这三者情况十分相近，无论是哪种决策方案均具有较好的性质，其中**决策 1[(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1), (1)]** 实现了每一步的检测与拆解，是三者中最稳定可靠的方案。每一步的检测与拆解，是三者中最稳定可靠的方案。

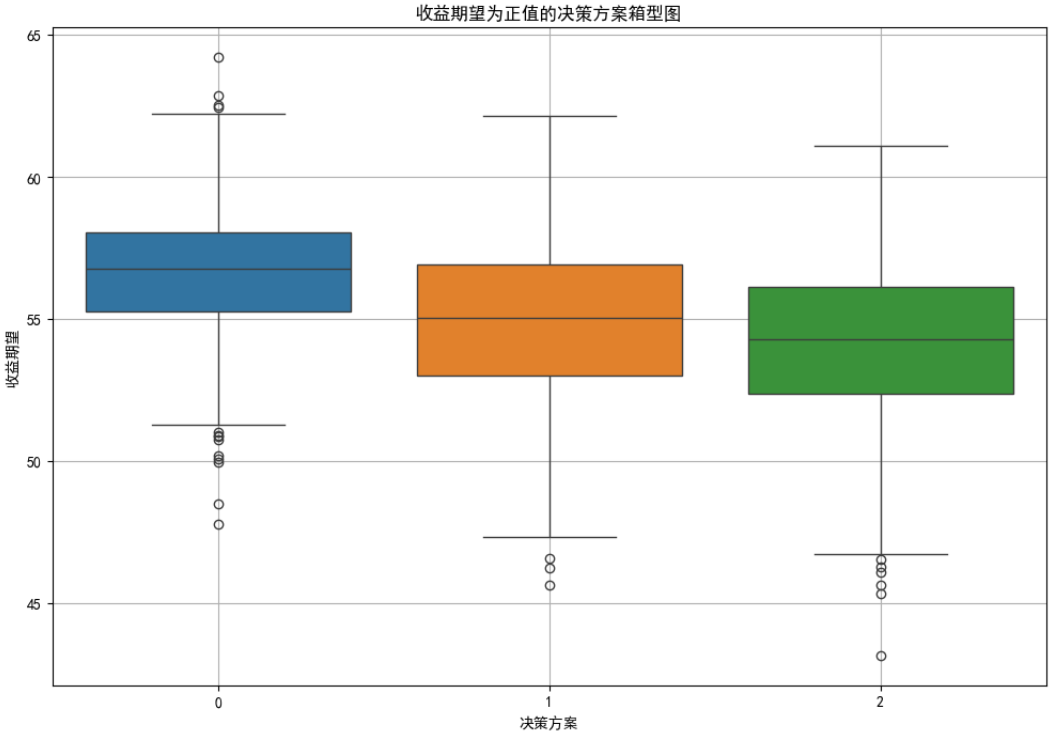



图 **15**　问题 **4** 背景下问题 **3** 相应决策的箱型图

对鲁棒性巨大差异的定性假想分析：多层网络的“次品率”存在严重的“传递”现象，一旦有一件次品流入了制造，其产生的影响是更为深远的，产生的损失也会在一级一级的传递中越来越大，那些利润巨大的方法往往是大量的省去了其中的检测步骤以节省成本，因此在某一个特定的次品率区间内表现会极为优异，但一旦次品率发生变化，由于它的“不检验”，产生的后续损失就会快速加大，影响数学期望。再回到前面给出的三种可行决策，无一不是经过了大量检测，即使是没检测的零件在下一层级就会

立马被检验出并及时处理，从而稳定性与利润期望大大上升。定性结论：在网络结构中稳定性往往强于一时利润

### 8.4 基于随机森林模型对先验样本量 $n_0$ 的讨论

在两问的重新计算中，本文都用到了贝叶斯统计的知识来计算后验的概率密度分布，而其中密切相关的一个量就是先验样本量 $n_0$。一旦先验样本量提高了，其后验概率就会更向中心聚拢，使得置信区间变窄，该样品的次品率波动就会下降。在这里以问题 4 背景下的问题 2 为例：我们建立随机森林模型，通过特征重要性来反映每一个自变量对最后的决策变量的影响程度如图 16所示，如果一个自变量对最后决策变量的影响很大，那么其波动性就会影响最后做出的决策；反过来看，最后做出的决策对于该自变量的“适应性”更弱，难以适应较大的波动、鲁棒性较差。因此，对相关性高的样本进行更多的先验取样，降低波动大的大小，对于全决策鲁棒性具有良好的帮助。

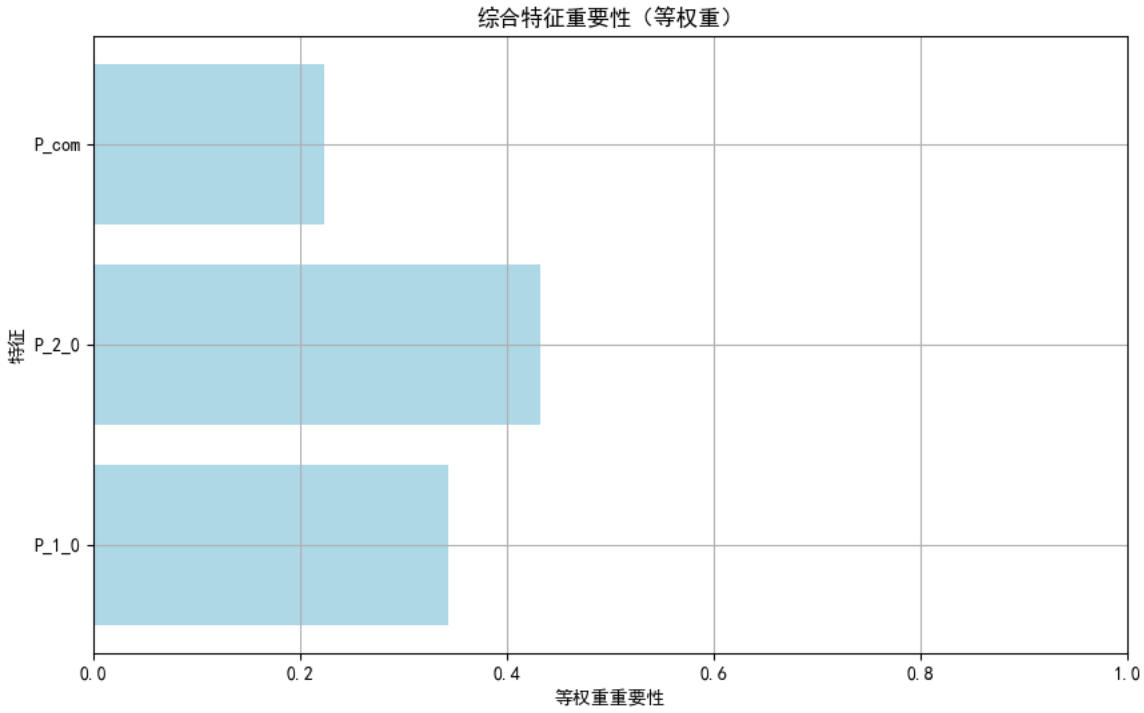



图 **16**　问题 **4** 背景下问题 **2** 综合特征重要性图

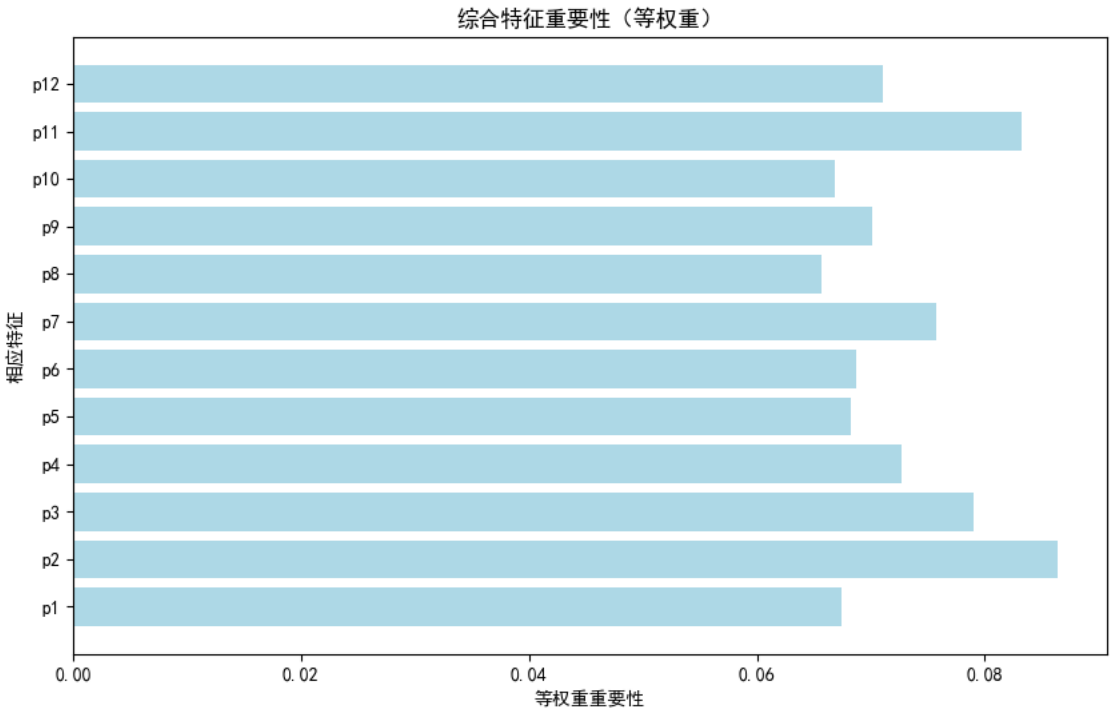



图 **17**　问题 **4** 背景下问题 **3** 综合特征重要性图

如图 16所示，在问题 2 的重新解决中 $P_{20}$ 这一个次品率值会对总决策变量有着更高的重要性，以此为判断依据，可以增加对零件二的样品抽测率，从而可以做出更好的决策选择。

而对于具有更多自变量的情况，如问题 4 背景下的问题 3 也可以使用该方法进行选择，问题 4 背景下问题 3 综合特征重要性图如图 17所示；此外，不仅能用于在整体决策值下的判断，对其中的任一决策变量也可以给出该关系图进行分析。如图 18是问题 4 背景下问题 3 相应决策位下的特征重要性的热力图，由图可知，其中决策量 15、16 对任何特征都不具备重要性，表明该决策量具有强鲁棒性，一旦做出正确决策就不需要任何调整，也不会与初始样本量的选择产生相关。

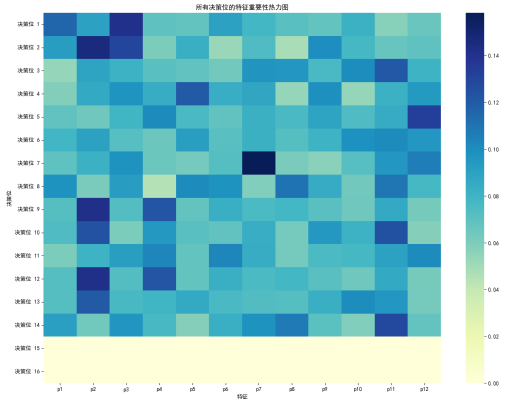


图 **18**　问题 **4** 背景下问题 **3** 相应决策位下的特征重要性热力图图

# 九、 模型的评价、改进与推广

## 9.1 模型的评价

(1). 本文提出的模型在进行假设时考虑了实际情况中的各种影响因素，所做的假设较为合理，且不会对结果造成过大的的偏差或误导。这使得本文的几个模型的假设在大多数场景下表现出较高的泛用性。在确保了其在实际产业中的广泛适用性和可靠性的同时，降低了求解难度，是之可以抽象为一般化的生产决策问题的处理。

(2). 本文模型在数学表达上表现出高度的精确性和严谨性。通过简练的数学推导，模型能够准确地刻画出实际问题中的核心要素和关键关系，避免了简化或误解实际情况的风险。模型的数学表达符合理论要求，能够为实际应用提供有力的理论支持。

(3). 在现有遗传退火模型的基础上，本文进一步进行了优化和改进。具体来说，是结合遗传算法和模拟退火算法的优点，借助退火算法来改善初始解，再用遗传算法更新较快的特点逐步优化最优解，在多次对比中找到最好的解。

(4). 在问题 3 的求解过程中，我们发现求解使用的函数 fsolve 会求解出一些不合理的值，推测这是因为 fsolve 函数求解的稳态值有多个，而 fsolve 函数无法辨别哪个是合理值，一开始本文尝试通过对目标函数施加惩罚函数去限定变量范围，但是这样会导致目标函数产生的解存在较大波动，使得可行解变得稀少。于是我们采取了直接舍去参数不合理情况的决策，这可能导致一定程度上我们丢失了最优解，有待改进。

## 9.2 模型的改进

(1). 模型的可靠性和稳定性在实际应用中至关重要，特别是在复杂的生产环境中，随机性因素可能会对决策产生重大影响。因此，可以考虑在现有的模型中引入更多的鲁棒优化方法，以应对实际中的不确定性。例如，可以在模型中增加应对异常情况的子模块，或者对参数进行敏感性分析，确保模型在不同的生产条件下依然能够稳定运行。

(2). 不同的生产过程可能具有不同的特点，现有的模型可能不能完全适应各种复杂的生产场景。可以对模型进行改进，使其具备更强的通用性。例如，通过参数化的方法，使模型能够根据生产环境的不同需求进行灵活调整，甚至在面对新的生产情境时能够自适应地调整优化策略。

## 9.3 模型的推广

（1）. 该模型目前主要应用于制造业的生产优化，但它的核心思想和方法具有普适性，可以推广到其他行业和领域。例如，该模型可以用于优化医院资源的调度和分配，减少资源浪费并提升患者服务效率。

（2）. 当前模型通过不同层次的递归优化来实现决策的动态调整，具有很强的适应性，可以进一步拓展多层次递归模型的应用，使其适用于更加复杂的系统。例如，通过

将模型应用于层次化的决策结构，比如公司管理中的多个决策层级，能够实现从公司战略层到操作层的全方位优化决策。

# 十、 参考文献与引用

## 参考文献


[1] Meng ni ZHANG, Can WANG, Jia jun BU, Zhi YU, Yu ZHOU, and Chun CHEN. 基于 url 聚类的快速无障碍检测抽样方法（英文）. Frontiers of Information Technology Electronic Engineering, 16(06):449–457, 2015.

[2] 李丽. 原材料检验抽样方法探讨. 科技创新与应用, (23):165, 2015.

[3] 毛瑞石. 贴片元器件 (smd) 的抽样检测方法研究. 科技广场, (02):80–82, 2012.

[4] 王茹. 产品质量检验抽样方法探究. 福建质量管理, (01):64, 2016.

[5] 董书琴 and 张斌. 一种面向流量异常检测的概率流抽样方法. 电子与信息学报, 41(06):1450–1457, 2019.

# 附录 A　所有附件名称罗列

表 **6**　支撑材料所有附件

| 附件名 |
| --- |
| 优化结果 2.xlsx |
| 最佳决策结果 2.xlsx |
| 最佳决策结果 2_ 情况 _1.xlsx |
| 最佳决策结果 2_ 情况 _2.xlsx |
| 最佳决策结果 2_ 情况 _3.xlsx |
| 最佳决策结果 2_ 情况 _4.xlsx |
| 最佳决策结果 2_ 情况 _5.xlsx |
| 最佳决策结果 2_ 情况 _6.xlsx |
| results4-2.txt |
| output_results.txt |
| B1.1.ipynb |
| B 问题四（1）-final.ipynb |
| B 问题四（2）.ipynb |
| B 问题二.ipynb |
| 题目数据.xlsx |
| 问题三模拟退火.ipynb |
| 问题三先退火再遗传.ipynb |
| 问题三先遗传再退火 + 退火.ipynb |
| 问题三穷举.ipynb |
| 问题三遗传算法.ipynb |

# 附录 B　B1.1

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_errors(n, p0, p_true, alpha):
# 计算第一类错误概率 ( )
z_alpha = stats.norm.ppf(1 - alpha) # 临界值
# 计算在零假设下的标准误差
se = np.sqrt((p0 * (1 - p0)) / n)

# 计算第一类错误概率
# P(Z > z_alpha | H0)
alpha_error = 1 - stats.norm.cdf(z_alpha)

# 计算第二类错误概率 ( )
# 计算在真实次品率下的z值
z_beta = (p_true - p0) / se
# P(Z < z_alpha | H1)
beta_error = stats.norm.cdf(z_alpha - z_beta)

return alpha_error, beta_error

# 设置参数
p0 = 0.10 # 标称次品率
p_true = 0.15 # 假设真实次品率
alpha = 0.05 # 显著性水平
sample_sizes = range(10, 1000, 10) # 样本量从10到200

# 存储错误概率
alpha_errors = []
beta_errors = []

# 计算每个样本量下的错误概率
for n in sample_sizes:
alpha_error, beta_error = calculate_errors(n, p0, p_true, alpha)
alpha_errors.append(alpha_error)
beta_errors.append(beta_error)

# 绘制错误概率与样本量的关系图
plt.figure(figsize=(10, 6))
```

```
plt.plot(sample_sizes, alpha_errors, label='第一类错误概率 ( )', marker='o')
plt.plot(sample_sizes, beta_errors, label='第二类错误概率 ( )', marker='o')
plt.title('第一类错误与第二类错误概率与样本量的关系')
plt.xlabel('样本量 (n)')
plt.ylabel('错误概率')
plt.axhline(y=alpha, color='r', linestyle='--', label='显著性水平 (  = 0.05)')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_sample_size_for_beta(p0, p_true, beta):
# 计算所需样本量，使得第二类错误概率为
z_beta = stats.norm.ppf(1 - beta) #  对应的z值
z_alpha = stats.norm.ppf(1 - 0.05) #   = 0.05对应的z值

# 计算样本量
n = ((z_alpha + z_beta) ** 2 * p0 * (1 - p0)) / (p_true - p0) ** 2
return int(np.ceil(n))

# 设置参数
p0 = 0.10 # 标称次品率
beta = 0.05 # 目标第二类错误概率
p_true_values = np.concatenate((np.arange(0.01, 0.095, 0.002), np.arange(0.105, 0.21, 0.002)))
    # 真实次品率

# 存储样本量
sample_sizes = []

# 计算每个真实次品率下的样本量
for p_true in p_true_values:
n = calculate_sample_size_for_beta(p0, p_true, beta)
sample_sizes.append(n)

# 绘制真实次品率与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(p_true_values, sample_sizes, label='样本量 (n)', marker='o')
plt.title('0.05显著性水平下达到置信度所需抽测样本量与样本次品率的关系')
plt.xlabel('样本次品率')
plt.ylabel('所需抽测样本量')
```

```
plt.ylim(0, 9000)
plt.axhline(y=5000, color='r', linestyle='--', label='可承受样本量上限')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_sample_size_for_beta(p0, p_true, beta):
# 计算所需样本量，使得第二类错误概率为
z_beta = stats.norm.ppf(1 - beta) #  对应的z值
z_alpha = stats.norm.ppf(1 - 0.05) #   = 0.05对应的z值

# 计算样本量
n = ((z_alpha + z_beta) ** 2 * p0 * (1 - p0)) / (p_true - p0) ** 2
return int(np.ceil(n))

# 设置参数
p0 = 0.10 # 标称次品率
beta = 0.05 # 目标第二类错误概率
p_true_values = np.arange(0.20, 0.50, 0.01) # 真实次品率

# 存储样本量
sample_sizes = []

# 计算每个真实次品率下的样本量
for p_true in p_true_values:
n = calculate_sample_size_for_beta(p0, p_true, beta)
sample_sizes.append(n)

# 绘制真实次品率与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(p_true_values, sample_sizes, label='样本量 (n)', marker='o')
plt.title('0.05显著性水平下达到置信度所需抽测样本量与样本次品率的关系')
plt.xlabel('样本次品率')
plt.ylabel('所需抽测样本量')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
```

```
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_errors(n, p0, p_true, alpha):
# 计算第一类错误概率 ( )
z_alpha = stats.norm.ppf(1 - alpha) # 临界值
# 计算在零假设下的标准误差
se = np.sqrt((p0 * (1 - p0)) / n)

# 计算第一类错误概率
# P(Z > z_alpha | H0)
alpha_error = 1 - stats.norm.cdf(z_alpha)

# 计算第二类错误概率 ( )
# 计算在真实次品率下的z值
z_beta = (p_true - p0) / se
# P(Z < z_alpha | H1)
beta_error = stats.norm.cdf(z_alpha - z_beta)

return alpha_error, beta_error

# 设置参数
p0 = 0.10 # 标称次品率
p_true = 0.15 # 假设真实次品率
alpha = 0.10 # 显著性水平
sample_sizes = range(10, 500, 10) # 样本量从10到200

# 存储错误概率
alpha_errors = []
beta_errors = []

# 计算每个样本量下的错误概率
for n in sample_sizes:
alpha_error, beta_error = calculate_errors(n, p0, p_true, alpha)
alpha_errors.append(alpha_error)
beta_errors.append(beta_error)

# 绘制错误概率与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(sample_sizes, alpha_errors, label='第一类错误概率 ( )', marker='o')
plt.plot(sample_sizes, beta_errors, label='第二类错误概率 ( )', marker='o')
plt.title('第一类错误与第二类错误概率与样本量的关系')
plt.xlabel('样本量 (n)')
```

```
plt.ylabel('错误概率')
plt.axhline(y=alpha, color='r', linestyle='--', label='显著性水平 ( = 0.10)')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_sample_size_for_beta(p0, p_true, beta):
# 计算所需样本量，使得第二类错误概率为
z_beta = stats.norm.ppf(1 - beta) #  对应的z值
z_alpha = stats.norm.ppf(1 - 0.10) #   = 0.10对应的z值

# 计算样本量
n = ((z_alpha + z_beta) ** 2 * p0 * (1 - p0)) / (p_true - p0) ** 2
return int(np.ceil(n))

# 设置参数
p0 = 0.10 # 标称次品率
beta = 0.10 # 目标第二类错误概率
p_true_values = np.concatenate((np.arange(0.01, 0.095, 0.002), np.arange(0.105, 0.21, 0.002)))
    # 真实次品率

# 存储样本量
sample_sizes = []

# 计算每个真实次品率下的样本量
for p_true in p_true_values:
n = calculate_sample_size_for_beta(p0, p_true, beta)
sample_sizes.append(n)

# 绘制真实次品率与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(p_true_values, sample_sizes, label='样本量 (n)', marker='o')
plt.title('0.1显著性水平下达到置信度所需抽测样本量与样本次品率的关系')
plt.xlabel('样本次品率')
plt.ylabel('所需抽测样本量')
plt.ylim(0, 9000)
plt.axhline(y=6000, color='r', linestyle='--', label='可承受样本量上限')
plt.legend()
plt.grid()
```

```
plt.show()
#%%
import numpy as np
import matplotlib.pyplot as plt
from scipy import stats

# 设置中文字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # 使用黑体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题

def calculate_sample_size_for_beta(p0, p_true, beta):
# 计算所需样本量，使得第二类错误概率为
z_beta = stats.norm.ppf(1 - beta) #  对应的z值
z_alpha = stats.norm.ppf(1 - 0.10) #   = 0.10对应的z值

# 计算样本量
n = ((z_alpha + z_beta) ** 2 * p0 * (1 - p0)) / (p_true - p0) ** 2
return int(np.ceil(n))

# 设置参数
p0 = 0.10 # 标称次品率
beta = 0.10 # 目标第二类错误概率
p_true_values = np.arange(0.20, 0.50, 0.01) # 真实次品率从0.11到0.30

# 存储样本量
sample_sizes = []

# 计算每个真实次品率下的样本量
for p_true in p_true_values:
n = calculate_sample_size_for_beta(p0, p_true, beta)
sample_sizes.append(n)

# 绘制真实次品率与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(p_true_values, sample_sizes, label='样本量 (n)', marker='o')
plt.title('0.1显著性水平下达到置信度所需抽测样本量与样本次品率的关系')
plt.xlabel('样本次品率')
plt.ylabel('所需抽测样本量')
plt.axhline(y=30, color='r', linestyle='--', label='初始抽样数')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
import scipy.stats as stats

def binomial_test_custom(sample_size, defective_items, defective_rate, confidence_level):
```

```
alpha = 1 - confidence_level
result = stats.binomtest(defective_items, sample_size, defective_rate, alternative='greater')

if result.pvalue < alpha:
return f"拒收零配件 (p-value: {result.pvalue:.5f})"
else:
return f"接收零配件 (p-value: {result.pvalue:.5f})"

def calculate_min_sample_size_custom(defective_rate, confidence_level, error_margin):
z_score = stats.norm.ppf(1 - (1 - confidence_level))
sample_size = (z_score ** 2 * defective_rate * (1 - defective_rate)) / (error_margin ** 2)
return int(np.ceil(sample_size))

def calculate_power_custom(sample_size, p0, p_true, alpha):
# 计算临界值
z_alpha = stats.norm.ppf(1 - alpha)

# 计算在真实次品率下的z值
z_power = (p_true - p0) / np.sqrt((p0 * (1 - p0)) / sample_size)

# 计算功效
power = 1 - stats.norm.cdf(z_alpha - z_power)
return power

defective_rate = 0.10
confidence_level_95 = 0.95
confidence_level_90 = 0.90
error_margin = 0.02

sample_size_95 = calculate_min_sample_size_custom(defective_rate, confidence_level_95,
    error_margin)
sample_size_90 = calculate_min_sample_size_custom(defective_rate, confidence_level_90,
    error_margin)

print(f"95% 置信水平下的最小样本量: {sample_size_95}")
print(f"90% 置信水平下的最小样本量: {sample_size_90}")

detected_defective_items = 12

result_95 = binomial_test_custom(sample_size_95, detected_defective_items, defective_rate,
    confidence_level_95)
result_90 = binomial_test_custom(sample_size_90, detected_defective_items, defective_rate,
    confidence_level_90)

print(f"95% 置信水平结果: {result_95}")
print(f"90% 置信水平结果: {result_90}")
```

```
# 设定真实次品率
p_true = 0.12 # 假设真实次品率为11%
alpha = 1 - confidence_level_90 # 计算90%信度下的显著性水平

# 计算检测效力
power_95 = calculate_power_custom(sample_size_95, defective_rate, p_true, alpha)
power_90 = calculate_power_custom(sample_size_90, defective_rate, p_true, alpha)

print(f"95% 置信水平下的检测效力: {power_95:.4f}")
print(f"90% 置信水平下的检测效力: {power_90:.4f}")
#%%
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

def binomial_test(sample_size, defective_items, defective_rate, confidence_level):
alpha = 1 - confidence_level
result = stats.binomtest(defective_items, sample_size, defective_rate, alternative='greater')

if result.pvalue < alpha:
return f"拒收零配件 (p-value: {result.pvalue:.5f})"
else:
return f"接收零配件 (p-value: {result.pvalue:.5f})"

def calculate_min_sample_size(defective_rate, confidence_level, error_margin):
z_score = stats.norm.ppf(1 - (1 - confidence_level) / 2)
sample_size = (z_score ** 2 * defective_rate * (1 - defective_rate)) / (error_margin ** 2)
return int(np.ceil(sample_size))

def calculate_power(sample_size, p0, p_true, alpha):
# 计算临界值
z_alpha = stats.norm.ppf(1 - alpha)

# 计算在真实次品率下的z值
z_power = (p_true - p0) / np.sqrt((p0 * (1 - p0)) / sample_size)

# 计算功效
power = 1 - stats.norm.cdf(z_alpha - z_power)
return power

# 设置参数
defective_rate = 0.10
confidence_level_95 = 0.95
confidence_level_90 = 0.90
error_margin = 0.01

# 计算最小样本量
```

```
sample_size_95 = calculate_min_sample_size(defective_rate, confidence_level_95, error_margin)
sample_size_90 = calculate_min_sample_size(defective_rate, confidence_level_90, error_margin)

# 设定真实次品率范围
p_true_values = np.linspace(0.05, 0.20, 100) # 从0.05到0.20的真实次品率

# 存储功效
power_95_values = []
power_90_values = []

# 计算不同真实次品率下的功效
for p_true in p_true_values:
power_95 = calculate_power(sample_size_95, defective_rate, p_true, 1 - confidence_level_95)
power_90 = calculate_power(sample_size_90, defective_rate, p_true, 1 - confidence_level_90)
power_95_values.append(power_95)
power_90_values.append(power_90)

# 绘制真实次品率与检测效力的关系图
plt.figure(figsize=(10, 6))
plt.plot(p_true_values, power_95_values, label='95% 置信水平的检测效力', color='blue')
plt.plot(p_true_values, power_90_values, label='90% 置信水平的检测效力', color='orange')
plt.title('真实次品率与检测效力的关系')
plt.xlabel('真实次品率 (p_true)')
plt.ylabel('检测效力 (Power)')
plt.axhline(y=0.8, color='red', linestyle='--', label='80% 检测效力阈值')
plt.axhline(y=0.9, color='green', linestyle='--', label='90% 检测效力阈值')
plt.legend()
plt.grid()
plt.show()
#%%
import numpy as np
import scipy.stats as stats
import matplotlib.pyplot as plt

def calculate_min_sample_size(defective_rate, confidence_level, error_margin):
z_score = stats.norm.ppf(1 - (1 - confidence_level) / 2)
sample_size = (z_score ** 2 * defective_rate * (1 - defective_rate)) / (error_margin ** 2)
return int(np.ceil(sample_size))

def calculate_power(sample_size, p0, p_true, alpha):
z_alpha = stats.norm.ppf(1 - alpha)
z_power = (p_true - p0) / np.sqrt((p0 * (1 - p0)) / sample_size)
power = 1 - stats.norm.cdf(z_alpha - z_power)
return power

# 设置参数
defective_rate = 0.10 # 标称次品率
```

```
confidence_level = 0.95 # 置信水平
target_power = 0.95 # 目标功效
p_true = 0.15 # 假设真实次品率
alpha = 1 - confidence_level # 显著性水平

# 允许误差范围
error_margins = np.linspace(0.001, 0.05, 100) # 从0.001到0.05的允许误差
sample_sizes = []

# 计算每个允许误差下的样本量
for error_margin in error_margins:
sample_size = calculate_min_sample_size(defective_rate, confidence_level, error_margin)
# 计算功效
power = calculate_power(sample_size, defective_rate, p_true, alpha)
# 确保功效达到目标
if power >= target_power:
sample_sizes.append(sample_size)
else:
sample_sizes.append(np.nan) # 如果功效不满足，记录为NaN

# 绘制最大允许误差与样本量的关系图
plt.figure(figsize=(10, 6))
plt.plot(error_margins, sample_sizes, label='样本量', marker='o')
plt.title('最大允许误差与样本量的关系')
plt.xlabel('最大允许误差 (Error Margin)')
plt.ylabel('样本量 (Sample Size)')
plt.legend()
plt.grid()
plt.ylim(0, max(sample_sizes) * 1.1) # 设置y轴上限
plt.show()
#%%
import scipy.stats as stats
import math

def sample_size(acceptance_rate, confidence_level):
# 计算样本大小
z = abs(stats.norm.ppf((1 - confidence_level))) # z值
p = acceptance_rate # 次品率
n = (z**2 * p * (1 - p)) / (0.02**2) # 抽样公式
return math.ceil(n) # 向上取整

def decision_scheme():
tolerance_rate = 0.1 # 次品率10%

# 问题1 (1)
n1 = sample_size(tolerance_rate, 0.95) # 95%信度
print(f'在95%信度下的抽样次数: {n1}')
```

```
# 问题1 (2)
n2 = sample_size(tolerance_rate, 0.90) # 90%信度
print(f'在90%信度下的抽样次数: {n2}')

decision_scheme()

#%%
import numpy as np
import matplotlib.pyplot as plt
# 设置 matplotlib 字体
plt.rcParams['font.sans-serif'] = ['SimHei'] # SimHei 是常见的中文字体
plt.rcParams['axes.unicode_minus'] = False # 用来正常显示负号

# 定义常数
z = 1.645 # 95% 单侧置信水平下的 z 值
epsilon = 0.02 # 允许误差

# 生成次品率 p 的范围
p_values = np.linspace(0.01, 0.30, 100) # 从 0.01 到 0.30 的次品率
# 计算对应的样本量 n
n_values = (z**2 * p_values * (1 - p_values)) / (epsilon**2)

# 计算 p = 0.1 时的样本量
p_target = 0.1
n_target = (z**2 * p_target * (1 - p_target)) / (epsilon**2)

# 绘图
plt.figure(figsize=(10, 6))
plt.plot(p_values, n_values, label='样本量 n 与次品率 p 的关系', color='blue')
plt.axhline(y=n_target, color='red', linestyle='--', label=f'p = {p_target} 时的样本量: n =
    {int(n_target)}')

# 标记 p = 0.1 时的样本量
plt.scatter(p_target, n_target, color='red')
plt.text(p_target, n_target, f'n = {int(n_target)}', verticalalignment='bottom',
    horizontalalignment='right')

# 设置图表标题和标签
plt.title('最小抽样样本量 n 与真实次品率 p 的关系')
plt.xlabel('真实次品率 p')
plt.ylabel('最小抽样样本量 n')
plt.legend()
plt.grid(True)

# 显示图表
plt.show()
```

```python
#%%
import numpy as np
import scipy.stats as stats

def sprt_sample_size(p0, p1, alpha, beta, initial_samples, max_samples, batch_size=10):
"""
顺序概率比检验 (SPRT) 用于质量控制。

p0: 原假设下的次品率（例如 1%）。
p1: 备择假设下的次品率（例如 10%）。
alpha: I 类错误率（例如 0.10 对应 90% 的置信水平）。
beta: II 类错误率（例如 0.20 对应 80% 的功效）。
initial_samples: 初始抽样数量。
max_samples: 最大样本量。
batch_size: 每次迭代增加的样本数。
"""
# 计算临界值：用于拒绝和接受原假设的阈值
A = np.log((1 - beta) / alpha) # 拒绝 H0 的临界值
B = np.log(beta / (1 - alpha)) # 接受 H0 的临界值

num_defects = 0 # 次品的样本数
num_total = initial_samples # 初始总样本数

# 从原假设下抽取初始样本
successes = np.random.binomial(1, p0, num_total) # 使用 p0 抽样，模拟标称次品率
num_defects += np.sum(successes) # 统计次品的样本数

print(f"初始样本量：{num_total}，次品数：{num_defects}")

# 顺序检验循环
while num_total <= max_samples:
# 计算当前样本的次品率
p_hat = num_defects / num_total
# 防止 p_hat 为 0 或 1 导致计算错误
if p_hat == 0:
p_hat = 1e-6 # 防止除以0
elif p_hat == 1:
p_hat = 1 - 1e-6

# 计算对数似然比
likelihood_ratio = (p_hat / p0) ** num_defects * ((1 - p_hat) / (1 - p0)) ** (num_total -
    num_defects)
log_likelihood_ratio = np.log(likelihood_ratio)

print(f"总样本数：{num_total}，次品数：{num_defects}，对数似然比：{log_likelihood_ratio}")
```

```
# 判断是否接受或拒绝原假设
if log_likelihood_ratio >= A:
return num_total, "拒绝 H0: 拒收该批次产品"
elif log_likelihood_ratio <= B:
return num_total, "接受 H0: 接收该批次产品"

# 增加样本量
if num_total < max_samples:
new_samples = np.random.binomial(1, p1, batch_size) # 每次增加一批次样本
num_total += batch_size # 更新总样本量
num_defects += np.sum(new_samples) # 更新次品数

# 防止死循环：如果达到最大样本量仍未得出结论，则返回结果
if num_total >= max_samples:
return num_total, "达到最大样本量，未得出结论"

# 设置参数
p0 = 0.01 # 原假设下的次品率
p1 = 0.10 # 备择假设下的次品率
alpha = 0.10 # 90% 置信区间
beta = 0.2 # 80% 功效
initial_samples = 300 # 初始样本量
max_samples = 100000 # 最大样本量
batch_size = 1 # 每次增加的样本量

# 运行 SPRT
final_sample_size, result = sprt_sample_size(p0, p1, alpha, beta, initial_samples,
    max_samples, batch_size)
print(f"最终样本量: {final_sample_size}")
print(f"检测结果: {result}")
```

# 附录 C　B 问题四（1）- final

```
import pandas as pd
import numpy as np
#%%
def profit(decision, product_data):
P_1, C_1, Ch_1, P_2, C_2, Ch_2, P_3, C_com, Ch_3, Price, C_change, C_dis = product_data
P_1_0, P_2_0, P_com = P_1, P_2, P_3
#首先计算第一次生产的成本
profit = 0
while True:
#首先计算该回合格品率
P_success = (1 - P_com) * (1 - (1-decision[0]) * P_1_0) * (1 - (1-decision[1]) * P_2_0)
```

```
    #不变情况下合格品比例
P_success_1 = (1 - P_com) * (1 - (1-decision[0]) * P_1) * (1 - (1-decision[1]) * P_2)
    #变化情况下合格品概率
P_1 = decision[3] * (P_1_0 * P_success_1 + (1 - decision[0]) * P_1) + (1 - decision[3]) *
    P_1_0 #零件1开始时次品率，为买来零件次品与返回零件次品,进行检测就不需要加上返回的次品
P_2 = decision[3] * (P_2_0 * P_success_1 + (1 - decision[1]) * P_2) + (1 - decision[3]) *
    P_2_0 #零件2开始时次品率，为买来零件次品与返回零件次品,进行检测就不需要加上返回的次品
# P_success_1 = (1 - P_com) * (1 - (1-decision[0]) * P_1) * (1 - (1-decision[1]) * P_2)
    #变化情况下合格品概率
P_3 = decision[3] * (1 - P_success_1) + (1 - decision[3]) * (1 - P_success) #组装的次品数
#分别计算各个选项花费
#计算开始零件购买的花费，分为拆解与不拆解
C_buy = decision[3] * (C_1 * P_success + C_2 * P_success) + (1 - decision[3]) * (C_1 + C_2)
#计算检测1，2费用，包含检测费用与检测后购买新零件费用，对输入的残次品进行检测买检测买直至没有次品，为等比数列，首项为
C_detection = Ch_1 * decision[0] + Ch_2 * decision[1] + P_1 / (1 - P_1_0) * (C_1 + Ch_1) *
    decision[0] + P_2 / (1 - P_2_0) * (C_2 + Ch_2) * decision[1]
#零件装配费用与检测和调回费用，
C_det_re = Ch_3 * decision[2] + (1 - decision[2]) * C_change * P_3
C_com_det = C_com + C_det_re
#计算拆解费用
C_total_dis = decision[3] * P_3 * C_dis
#计算总成本
C_total = C_buy + C_detection + C_com_det + C_total_dis
#计算总利润
profit_1 = Price * P_success_1 - C_total
if np.abs(profit_1 - profit) < 0.000000001 :
break
#存储利润以判别是否停止
profit = profit_1
return profit

#%%
import itertools

# Generate all possible combinations of 0s and 1s for a list of length 4
combinations = list(itertools.product([0, 1], repeat=4))

# Convert each combination to a list
decision_lists = [list(combination) for combination in combinations]
product_data_total = pd.read_excel("表1(1).xlsx")
product_data_total= np.array(product_data_total.iloc[1:7,:]).tolist()
best_profit = 0

for product_data in product_data_total:
best_profit = 0
for decision in decision_lists:
tem_profit = profit(decision, product_data)
```

```
if tem_profit > best_profit :
best_decision = decision
best_profit = tem_profit

print(f"该组合最好的决策为{best_decision}")
print(f"该组合最好收益为{best_profit:.2f}元")
#%%
import numpy as np

import matplotlib.pyplot as plt
from scipy.stats import beta

# 抽样数据
n = 100 # 总样本数

# 定义不同的次品率
defect_rates = [0.2, 0.1,0.05]

# 选择先验分布参数
alpha_prior = 1 # 贝塔分布的先验参数 alpha
beta_prior = 1 # 贝塔分布的先验参数 beta

# 生成 p 的值
p_values = np.linspace(0, 1, 100)


plt.figure(figsize=(10, 6))


for rate in defect_rates:

k = int(rate * n) # 次品数

# 更新后验分布参数
alpha_post = alpha_prior + k # 后验 alpha
beta_post = beta_prior + (n - k) # 后验 beta

# 计算后验分布
posterior_distribution = beta.pdf(p_values, alpha_post, beta_post)



# 绘制图像
plt.plot(p_values, posterior_distribution, label=f'Posterior Distribution (p={rate})')


plt.title('Posterior Distribution of Defect Rate')
```

```
plt.xlabel('Defect Rate (p)')

plt.ylabel('Density')

plt.axvline(x=0.1, color='red', linestyle='--', label='Sample Estimate (p=0.1)')
plt.legend()
plt.grid()
plt.show()

#%%
# 抽样数据
def calculate_confidence_intervals(defect_rate):
n = 100 # 总样本数
k = int(defect_rate * n) # 次品数

# 选择先验分布参数
alpha_prior = 1 # 贝塔分布的先验参数 alpha
beta_prior = 1 # 贝塔分布的先验参数 beta

# 更新后验分布参数
alpha_post = alpha_prior + k # 后验 alpha
beta_post = beta_prior + (n - k) # 后验 beta

# 计算置信区间
for confidence_level in [0.95, 0.90]:
lower_bound = beta.ppf((1 - confidence_level) / 2, alpha_post, beta_post)
upper_bound = beta.ppf(1 - (1 - confidence_level) / 2, alpha_post, beta_post)
print(f"次品率为 {defect_rate} 的 {confidence_level*100}% 置信区间: [{lower_bound:.4f},
    {upper_bound:.4f}]")

# 计算不同次品率的置信区间
for rate in [0.05, 0.1, 0.2]:
calculate_confidence_intervals(rate)
#%%
# 分析结果
# 依次从第一行到第六行取初始值
initial_values = [
(0.1, 4, 2, 0.1, 18, 3, 0.1, 6, 3, 56, 6, 5),
(0.2, 4, 2, 0.2, 18, 3, 0.2, 6, 3, 56, 6, 5),
(0.1, 4, 2, 0.1, 18, 3, 0.1, 6, 3, 56, 30, 5),
(0.2, 4, 1, 0.2, 18, 1, 0.2, 6, 2, 56, 30, 5),
(0.1, 4, 8, 0.2, 18, 1, 0.1, 6, 2, 56, 10, 5),
(0.05, 4, 2, 0.05, 18, 3, 0.05, 6, 3, 56, 10, 40)
]

# # 更新次品率波动范围
```

```
defect_rate_ranges = {
    0.2: np.linspace(0.1336, 0.2891, 5),
    0.1: np.linspace(0.0556, 0.1746, 5),
    0.05: np.linspace(0.0221, 0.1118, 5)
}

# 创建一个空的DataFrame来存储结果
results = pd.DataFrame(columns=["P_1_0", "P_2_0", "P_com", "最佳决策", "利润"])

for values in initial_values:
P_1, C_1, Ch_1, P_2, C_2, Ch_2, P_3, C_com, Ch_3, Price, C_change, C_dis = values
# 获取当前次品率对应的范围
P_1_0_range = defect_rate_ranges[P_1]
P_2_0_range = defect_rate_ranges[P_2]
P_com_range = defect_rate_ranges[P_3]

for P_1_0 in P_1_0_range:
for P_2_0 in P_2_0_range:
for P_com in P_com_range:
# 更新product_data中的次品率
product_data = [P_1_0, C_1, Ch_1, P_2_0, C_2, Ch_2, P_com, C_com, Ch_3, Price, C_change, C_dis]
best_profit = 0
best_decision = None

for decision in decision_lists:
tem_profit = profit(decision, product_data)
if tem_profit > best_profit:
best_decision = decision
best_profit = tem_profit

# 将结果添加到DataFrame中
results = pd.concat([results, pd.DataFrame([{"P_1_0": P_1_0, "P_2_0": P_2_0, "P_com": P_com,
    "最佳决策": best_decision, "利润": best_profit}])], ignore_index=True)

# 输出结果到控制台
print(results)

# 将结果导出到Excel文件
results.to_excel("最佳决策结果1.xlsx", index=False) # 保存为Excel文件
#%%
import pandas as pd
# 此处是将总体的结果二划分至六个情况中
# 读取Excel文件
df = pd.read_excel("最佳决策结果2.xlsx")

# 每8000行分为一种情况
chunk_size = 8000
```

```
num_chunks = len(df) // chunk_size + (1 if len(df) % chunk_size != 0 else 0)

# 将每个分块保存到不同的Excel文件
for i in range(num_chunks):
chunk = df.iloc[i * chunk_size:(i + 1) * chunk_size]
chunk.to_excel(f"最佳决策结果2_情况_{i + 1}.xlsx", index=False)
#%%
# 导入必要的库
import numpy as np
import pandas as pd # 添加导入pandas库
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression # 添加导入LogisticRegression
from sklearn.metrics import accuracy_score # 添加导入accuracy_score
from sklearn.inspection import PartialDependenceDisplay # 更新导入方式
# 读取用户上传的新的Excel文件
file_path_new = '最佳决策结果2.xlsx'
data_new = pd.read_excel(file_path_new)

# 展示前几行数据以查看内容结构
data_new.head()

# 将 '最佳决策' 列拆分为四个独立的决策位
data_new[['决策位1', '决策位2', '决策位3', '决策位4']] =
    pd.DataFrame(data_new['最佳决策'].apply(eval).tolist(), index=data_new.index)

# 删除原 '最佳决策' 列，保留拆分后的四个位
data_new_cleaned = data_new.drop(columns=['最佳决策'])

# 展示处理后的数据
data_new_cleaned.head()

# 检查每个决策位的值分布，看看是否存在只包含一个类别的情况
distribution_new = data_new_cleaned[['决策位1', '决策位2', '决策位3',
    '决策位4']].apply(pd.Series.value_counts)

# 展示每个决策位的类别分布情况
distribution_new

# 准备自变量和因变量（分别针对每个决策位）
X_new = data_new_cleaned[['P_1_0', 'P_2_0', 'P_com']]

# 创建一个字典，用于存储每个决策位的模型和结果
new_results = {}

for i in range(1, 5):
```

```
# 提取决策位
y = data_new_cleaned[f'决策位{i}']

# 拆分训练集和测试集
X_train_new, X_test_new, y_train_new, y_test_new = train_test_split(X_new, y, test_size=0.2,
    random_state=42)

# 使用逻辑回归进行训练
model_new = LogisticRegression()
model_new.fit(X_train_new, y_train_new)

# 预测并计算准确率
y_pred_new = model_new.predict(X_test_new)
accuracy_new = accuracy_score(y_test_new, y_pred_new)

# 存储结果
new_results[f'决策位{i}'] = {
   'model': model_new,
   'accuracy': accuracy_new,
   'coefficients': model_new.coef_
}

# 展示每个决策位的回归系数和模型准确率
new_results


#%%
import matplotlib.pyplot as plt
from matplotlib import rcParams

# 设置中文字体和解决负号显示问题
rcParams['font.sans-serif'] = ['SimHei'] # 使用SimHei字体支持中文
rcParams['axes.unicode_minus'] = False # 解决负号显示问题

# 1. 可视化特征重要性（通过回归系数展示）
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
decision_vars = ['决策位1', '决策位2', '决策位3', '决策位4']
features = ['P_1_0', 'P_2_0', 'P_com']

# 绘制每个决策位的回归系数条形图
for i, ax in enumerate(axes.flat):
decision_var = decision_vars[i]
coefficients = new_results[decision_var]['coefficients'][0]

# 绘制条形图，显示每个自变量的系数
ax.bar(features, coefficients, color='skyblue')
ax.set_title(f'{decision_var} - 回归系数')
```

```
ax.set_ylabel('系数值')
ax.set_ylim([-50, 50]) # 设置y轴范围以便比较
ax.grid(True)

plt.tight_layout()
plt.show()

#%%
# 导入随机森林分类器
from sklearn.ensemble import RandomForestClassifier

# 创建一个字典，用于存储每个决策位的随机森林模型和结果
rf_results = {}
file_path_new = '最佳决策结果2.xlsx'
data_new = pd.read_excel(file_path_new)

# 展示前几行数据以查看内容结构
data_new.head()

# 将 '最佳决策' 列拆分为四个独立的决策位
data_new[['决策位1', '决策位2', '决策位3', '决策位4']] =
    pd.DataFrame(data_new['最佳决策'].apply(eval).tolist(), index=data_new.index)

# 删除原 '最佳决策' 列，保留拆分后的四个位
data_new_cleaned = data_new.drop(columns=['最佳决策'])

# 展示处理后的数据
data_new_cleaned.head()

# 准备自变量和因变量（分别针对每个决策位）
X_new = data_new_cleaned[['P_1_0', 'P_2_0', 'P_com']]
# 使用随机森林进行分析
for i in range(1, 5):
# 提取决策位
y_rf = data_new_cleaned[f'决策位{i}']

# 拆分训练集和测试集
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_new, y_rf, test_size=0.2,
    random_state=42)

# 使用随机森林进行训练
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_rf, y_train_rf)

# 预测并计算准确率
y_pred_rf = rf_model.predict(X_test_rf)
accuracy_rf = accuracy_score(y_test_rf, y_pred_rf)
```

```
# 存储结果
rf_results[f'决策位{i}'] = {
    'model': rf_model,
    'accuracy': accuracy_rf,
    'feature_importances': rf_model.feature_importances_
}

# 展示随机森林的模型准确率和特征重要性
rf_results


#%%
# 可视化每个决策位的特征重要性
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
decision_vars_rf = ['决策位1', '决策位2', '决策位3', '决策位4']
features_rf = ['P_1_0', 'P_2_0', 'P_com']

for i, ax in enumerate(axes.flat):
decision_var_rf = decision_vars_rf[i]
importances = rf_results[decision_var_rf]['feature_importances']

# 绘制条形图，显示每个自变量的特征重要性
ax.bar(features_rf, importances, color='lightgreen')
ax.set_title(f'{decision_var_rf} - 特征重要性')
ax.set_ylabel('重要性')
ax.set_ylim([0, 1]) # 设置y轴范围以便比较
ax.grid(True)

plt.tight_layout()
plt.show()
#%%
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# 创建一个字典，用于存储每个决策位的随机森林模型和结果
rf_results = {}
file_path_new = '最佳决策结果2.xlsx'
data_new = pd.read_excel(file_path_new)

# 展示前几行数据以查看内容结构
print(data_new.head())
```

```
# 将 '最佳决策' 列拆分为四个独立的决策位
data_new[['决策位1', '决策位2', '决策位3', '决策位4']] =
    pd.DataFrame(data_new['最佳决策'].apply(eval).tolist(), index=data_new.index)

# 删除原 '最佳决策' 列，保留拆分后的四个位
data_new_cleaned = data_new.drop(columns=['最佳决策'])

# 展示处理后的数据
print(data_new_cleaned.head())

# 准备自变量和因变量（分别针对每个决策位）
X_new = data_new_cleaned[['P_1_0', 'P_2_0', 'P_com']]

# 使用随机森林进行分析
for i in range(1, 5):
# 提取决策位
y_rf = data_new_cleaned[f'决策位{i}']

# 拆分训练集和测试集
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_new, y_rf, test_size=0.2,
    random_state=42)

# 使用随机森林进行训练
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_rf, y_train_rf)

# 预测并计算准确率
y_pred_rf = rf_model.predict(X_test_rf)
accuracy_rf = accuracy_score(y_test_rf, y_pred_rf)

# 存储结果
rf_results[f'决策位{i}'] = {
   'model': rf_model,
   'accuracy': accuracy_rf,
   'feature_importances': rf_model.feature_importances_
}

# 计算等权重加权特征重要性
features_rf = ['P_1_0', 'P_2_0', 'P_com'] # 特征名称
num_decisions = len(rf_results)
average_importances = np.zeros(len(features_rf))

for decision_var, data in rf_results.items():
importances = np.array(data['feature_importances'])
average_importances += importances

# 计算平均值
```

```
average_importances /= num_decisions

# 打印随机森林的模型准确率和特征重要性
print(rf_results)

# 绘制等权重加权特征重要性为横向条形图
plt.figure(figsize=(10, 6))
plt.barh(features_rf, average_importances, color='lightblue') # 改为barh，生成横向条形图
plt.title('综合特征重要性（等权重）')
plt.xlabel('等权重重要性')
plt.ylabel('特征')
plt.xlim([0, 1]) # 设置x轴范围以便比较
plt.grid(True)
plt.show()

#%%
import ast
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
n = 100
#假设已经获得了一个best_decision，想要检验他的鲁棒性
#该情形下这种决策应该有最大的收益期望：对所有情况使用该决策来进行对不同次品的计算
#这里是随机化后面的各项次品率
initial_values = [
(0.1, 4, 2, 0.1, 18, 3, 0.1, 6, 3, 56, 6, 5),
(0.2, 4, 2, 0.2, 18, 3, 0.2, 6, 3, 56, 6, 5),
(0.1, 4, 2, 0.1, 18, 3, 0.1, 6, 3, 56, 30, 5),
(0.2, 4, 1, 0.2, 18, 1, 0.2, 6, 2, 56, 30, 5),
(0.1, 4, 8, 0.2, 18, 1, 0.1, 6, 2, 56, 10, 5),
(0.05, 4, 2, 0.05, 18, 3, 0.05, 6, 3, 56, 10, 40)
]

with open('output_results.txt', 'w') as file:
re_profit1 = []
for i in range(1,7) :
re_profit2 = []
file_path = f'最佳决策结果2_情况_{i}.xlsx'
data = pd.read_excel(file_path)
# 选择要处理的列
column_name = '最佳决策' # 替换为你实际的列名
file.write(f'情况{i}:\n')
print(f'情况{i}:')
unique_values = data[column_name].dropna().unique() #取出列中独一无二的决策
for Input_decision in unique_values:
re_profit3 = []
```

```
Input_decision = ast.literal_eval(Input_decision) #转换一下Input_decision的形式
total_profit = 0
P_1, C_1, Ch_1, P_2, C_2, Ch_2, P_3, C_com, Ch_3, Price, C_change, C_dis = initial_values[i-1]
# 获取当前次品率对应的范围
for _ in range(1000):
# 从后验分布中随机生成次品率
defect_rate = {
   0.2: beta.rvs(alpha_prior + int(0.2 * n), beta_prior + (n - int(0.2 * n)), size=1)[0],
   0.1: beta.rvs(alpha_prior + int(0.1 * n), beta_prior + (n - int(0.1 * n)), size=1)[0],
   0.05: beta.rvs(alpha_prior + int(0.05 * n), beta_prior + (n - int(0.05 * n)), size=1)[0]
}

P_1_0 = defect_rate[P_1]
P_2_0 = defect_rate[P_2]
P_com = defect_rate[P_3]
# 生成 product_data 并添加到列表中
product_data = [P_1_0, C_1, Ch_1, P_2_0, C_2, Ch_2, P_com, C_com, Ch_3, Price, C_change, C_dis]
tem_profit = profit(Input_decision, product_data)
# print(tem_profit)
total_profit += tem_profit
re_profit3.append(tem_profit)
# 输出结果到控制台
re_profit2.append(re_profit3)
print(f'决策 {Input_decision} 的收益期望为: {total_profit/1000}')
variance = np.var(re_profit3)
print(f'决策 {Input_decision} 的收益期望方差为: {variance}')
file.write(f'决策 {Input_decision} 的收益期望方差为: {variance}\n')
file.write(f'决策 {Input_decision} 的收益期望为: {total_profit / 1000}\n')
re_profit1.append(re_profit2)

j = 0
column_name = '最佳决策' # 替换为你实际的列名
for i in re_profit1:
j = j + 1
file_path = f'最佳决策结果2_情况_{j}.xlsx'
data = pd.read_excel(file_path)
unique_values = data[column_name].dropna().unique()
# 创建箱型图
plt.figure(figsize=(8, 6))
sns.boxplot(data=i)
# 添加标题和标签
plt.xticks(range(len(unique_values)), unique_values)
plt.title(f'情况{j}箱型图')
plt.xlabel('决策')
```

# 附录 D　B 问题四（2）

```
import numpy as np
from scipy.optimize import fsolve
import itertools

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) * (1
    - (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars

eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) * (1
    - (1 - x_2) * P_x2_chu)))
```

```
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu', 'P_x2_ru',
    'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_jia
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
```

```
    P_x3_bu, P_zhuang_y1)
solution_1_1 = fsolve(three_equations, initial_guess_3, args=(decision[2][0], decision[0][0],
    decision[0][1], decision[0][2], decision[1][0], def_rate[0][0], def_rate[0][1],
    def_rate[0][2], def_rate[1][0]))
#构建第二个组合（1，2）,其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_1_2 = fsolve(three_equations, initial_guess_3, args=(decision[2][1], decision[0][3],
    decision[0][4], decision[0][5], decision[1][1], def_rate[0][3], def_rate[0][4],
    def_rate[0][5], def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu,
    P_zhuang_y1)
solution_1_3= fsolve(two_equations, initial_guess_2, args=(decision[2][2], decision[0][6],
    decision[0][7], decision[1][2], def_rate[0][6], def_rate[0][7], def_rate[1][2]))
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) * (1
    - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) * (1
    - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) *
    (1 - decision[1][2])
#构建第4个组合（2，1）,其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_2_1 = fsolve(three_equations, initial_guess_3, args=(decision[4][0], decision[1][0],
    decision[1][1], decision[1][2], decision[3][0], def_rate_1_1, def_rate_1_2, def_rate_1_3,
    def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solutio
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solutio
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][6]
#计算组合（2，1）单价
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solutio
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) * (1
    - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = solution_2_1
```

```
if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0
return profit

#%%
# 首先设置最优值的初值，与最优解
max_profit = 0
best_decision = None

# 生成每个子列表的 0 和 1 的组合
decision_combinations = [list(itertools.product([0, 1], repeat=len(sub_decision))) for
    sub_decision in decision]

# 生成所有可能的组合并转换为列表
all_combinations = list(itertools.product(*decision_combinations))

# 将所有组合转换为列表
all_combinations_list = [list(combination) for combination in all_combinations]

# 打印所有组合
for combination in all_combinations_list:
if combination[1][0] - combination[2][0] < 0:
continue
if combination[1][1] - combination[2][1] < 0:
continue
if combination[1][2] - combination[2][2] < 0:
continue
if combination[3][0] - combination[4][0] < 0 :
continue
# Calculate profit for the current decision combination
profit_value = profit(combination, def_rate)
# Update maximum profit and best decision if the current profit is higher
if profit_value > max_profit:
max_profit = profit_value
best_decision = combination

print(f"Maximum profit: {max_profit}")
print(f"Best decision: {best_decision}")
#%%
import numpy as np
import pandas as pd
from scipy.optimize import fsolve
import itertools
```

```
# 假设其他必要的导入和数据准备在这里

# 定义变量
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
# 定义新的def_rate范围
def_rate_values = np.linspace(0.0556, 0.1746, 5)

# 存储结果的列表
results = []
# 遍历随机生成的def_rate值
for _ in range(100): # 假设进行10次随机取样
# 随机生成def_rate
def_rate = [
np.random.choice(def_rate_values, size=8).tolist(), # 8个次品率
np.random.choice(def_rate_values, size=3).tolist(), # 3个次品率
[np.random.choice(def_rate_values)] # 1个次品率
]

# 进行优化计算（假设profit函数已经定义）
max_profit = 0
best_decision = None

# 生成每个子列表的 0 和 1 的组合
decision_combinations = [list(itertools.product([0, 1], repeat=len(sub_decision))) for
    sub_decision in decision]

# 生成所有可能的组合并转换为列表
all_combinations = list(itertools.product(*decision_combinations))

# 将所有组合转换为列表
all_combinations_list = [list(combination) for combination in all_combinations]

# 打印所有组合
for combination in all_combinations_list:
if combination[1][0] - combination[2][0] < 0:
continue
if combination[1][1] - combination[2][1] < 0:
continue
if combination[1][2] - combination[2][2] < 0:
continue
```

```
if combination[3][0] - combination[4][0] < 0:
continue

# 计算利润
profit_value = profit(combination, def_rate) # 假设profit函数已经定义

# 更新最大利润和最佳决策
if profit_value > max_profit:
max_profit = profit_value
best_decision = combination

# 保存结果
results.append({
  'def_rate': def_rate, # 输出当前def_rate
  'best_decision': best_decision,
  'max_profit': max_profit
})

# 将结果保存到Excel
results_df = pd.DataFrame(results)
results_df.to_excel("优化结果2.xlsx", index=False)
#%%
# 导入随机森林分类器
from sklearn.ensemble import RandomForestClassifier
import numpy as np
import pandas as pd # 添加导入pandas库
import matplotlib.pyplot as plt
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression # 添加导入LogisticRegression
from sklearn.metrics import accuracy_score # 添加导入accuracy_score
from sklearn.inspection import PartialDependenceDisplay # 更新导入方式
file_path_new = '优化结果2.xlsx'
data_new = pd.read_excel(file_path_new)
# 将 'best_decision' 列中的字符串转换为列表，并展平每行的所有决策，再填充到 16 位
data_new[['决策位1', '决策位2', '决策位3', '决策位4', '决策位5', '决策位6', '决策位7',
    '决策位8',
'决策位9', '决策位10', '决策位11', '决策位12', '决策位13', '决策位14', '决策位15', '决策位16']]
    = \
pd.DataFrame(
data_new['best_decision'].apply(
lambda x: eval(x) # 将字符串形式转换为列表
if isinstance(x, str) else x
).apply(
lambda x: [item for sublist in x for item in sublist] + [np.nan] * (16 - sum(len(sublist) for
    sublist in x)) # 展平并填充至16个元素
).tolist(),
```

```
index=data_new.index
)
data_new[['p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8','p9', 'p10', 'p11','p12']] = \
pd.DataFrame(
data_new['def_rate'].apply(
lambda x: eval(x) # 将字符串形式转换为列表
if isinstance(x, str) else x
).apply(
lambda x: [item for sublist in x for item in sublist] + [np.nan] * (12 - sum(len(sublist) for
    sublist in x)) # 展平并填充至12个元素
).tolist(),
index=data_new.index
)
# 删除原 'best_decision' 列，保留拆分后的十六个位
data_new_cleaned = data_new.drop(columns=['best_decision'])
# 删除原 'def_rate' 列，保留拆分后的十二个位
data_new_cleaned = data_new_cleaned.drop(columns=['def_rate'])
data_new_cleaned.head()
# 准备自变量和因变量（分别针对每个决策位）
X_new = data_new_cleaned[['p1', 'p2', 'p3', 'p4', 'p5', 'p6', 'p7', 'p8','p9', 'p10',
    'p11','p12']]
# 初始化存储随机森林结果的字典
rf_results = {}
for i in range(1, 17):
# 提取决策位
y_rf = data_new_cleaned[f'决策位{i}']

# 拆分训练集和测试集
X_train_rf, X_test_rf, y_train_rf, y_test_rf = train_test_split(X_new, y_rf, test_size=0.2,
    random_state=34)

# 使用随机森林进行训练
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train_rf, y_train_rf)

# 预测并计算准确率
y_pred_rf = rf_model.predict(X_test_rf)
accuracy_rf = accuracy_score(y_test_rf, y_pred_rf)

# 存储结果
rf_results[f'决策位{i}'] = {
  'model': rf_model,
  'accuracy': accuracy_rf,
  'feature_importances': rf_model.feature_importances_
}
# 展示随机森林的模型准确率和特征重要性
rf_results
```

```
#%%
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import rcParams
# 设置中文字体和解决负号显示问题
rcParams['font.sans-serif'] = ['SimHei'] # 使用SimHei字体支持中文
rcParams['axes.unicode_minus'] = False # 解决负号显示问题

# 计算等权重加权特征重要性
num_decision_vars = 16 # 假设有16个决策位
features = X_new.columns

# 初始化特征重要性数组
average_importances = np.zeros(len(features))

# 计算每个特征的重要性平均值
for i in range(1, num_decision_vars + 1):
if f'决策位{i}' in rf_results:
importances = np.array(rf_results[f'决策位{i}']['feature_importances'])
average_importances += importances

# 计算平均值
average_importances /= num_decision_vars

# 绘制等权重加权特征重要性
plt.figure(figsize=(10, 6))
plt.barh(features, average_importances, color='lightblue')
plt.xlabel("等权重重要性")
plt.ylabel("相应特征")
plt.title("综合特征重要性（等权重）")
plt.show()

#%%
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np

# 假设 rf_results 是包含特征重要性的字典，其中每个键为 '决策位X'，值为特征重要性列表
# 示例数据
# rf_results = {
  #     f'决策位{i}': {'feature_importances': np.random.rand(len(X_new.columns))}
  #     for i in range(1, 17)
  # }

# 特征名称（根据实际数据修改）
```

```
feature_names = X_new.columns

# 创建一个大的二维数组来存储所有的热力图数据
num_decision_positions = 16
num_features = len(feature_names)
heatmap_matrix = np.zeros((num_decision_positions, num_features))

# 填充数据
for i in range(1, num_decision_positions + 1):
feature_importances = rf_results[f'决策位{i}']['feature_importances']
heatmap_matrix[i-1, :] = feature_importances

# 创建热力图
plt.figure(figsize=(16, 12))
sns.heatmap(heatmap_matrix, cmap="YlGnBu", cbar=True, xticklabels=feature_names,
    yticklabels=[f'决策位 {i}' for i in range(1, 17)])

# 设置标题和标签
plt.title("所有决策位的特征重要性热力图")
plt.xlabel("特征")
plt.ylabel("决策位")

# 显示图表
plt.show()

#%%
import numpy as np

import matplotlib.pyplot as plt
from scipy.stats import beta

# 抽样数据
n = 100 # 总样本数

# 定义不同的次品率
defect_rates = [0.2, 0.1,0.05]

# 选择先验分布参数
alpha_prior = 1 # 贝塔分布的先验参数 alpha
beta_prior = 1 # 贝塔分布的先验参数 beta

# 生成 p 的值
p_values = np.linspace(0, 1, 100)


plt.figure(figsize=(10, 6))
```

```
for rate in defect_rates:

k = int(rate * n) # 次品数

# 更新后验分布参数
alpha_post = alpha_prior + k # 后验 alpha
beta_post = beta_prior + (n - k) # 后验 beta

# 计算后验分布
posterior_distribution = beta.pdf(p_values, alpha_post, beta_post)



# 绘制图像
plt.plot(p_values, posterior_distribution, label=f'Posterior Distribution (p={rate})')


plt.title('Posterior Distribution of Defect Rate')

plt.xlabel('Defect Rate (p)')

plt.ylabel('Density')

plt.axvline(x=0.1, color='red', linestyle='--', label='Sample Estimate (p=0.1)')
plt.legend()
plt.grid()
plt.show()

#%%
import numpy as np
import pandas as pd
from scipy.stats import beta
import ast
import matplotlib.pyplot as plt
from matplotlib import rcParams
import seaborn as sns

# 配置字体以支持中文显示（使用系统默认字体）
rcParams['font.sans-serif'] = ['SimHei'] # 中文字体
rcParams['axes.unicode_minus'] = False # 正常显示负号

# 假设的先验参数
alpha_prior = 1
beta_prior = 1
n = 100 # 总样本数
```

```
# 定义变量
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]

file_path = '优化结果2.xlsx'
data = pd.read_excel(file_path)
column_name = 'best_decision'
unique_values = data[column_name].dropna().unique()

positive_profits = []

# 计算每个决策的收益期望，并筛选出正值
with open('results4-2.txt', 'w') as file:
for Input_decision in unique_values:
re_profit2 = []
Input_decision = ast.literal_eval(Input_decision)
total_profit = 0
for _ in range(1000):
# 从后验分布中随机生成次品率
defect_rate = {
  0.2: beta.rvs(alpha_prior + int(0.2 * n), beta_prior + (n - int(0.2 * n)), size=1)[0],
  0.1: beta.rvs(alpha_prior + int(0.1 * n), beta_prior + (n - int(0.1 * n)), size=1)[0],
  0.05: beta.rvs(alpha_prior + int(0.05 * n), beta_prior + (n - int(0.05 * n)), size=1)[0]
}

def_rate = [
[beta.rvs(alpha_prior + int(0.1 * n), beta_prior + (n - int(0.1 * n)), size=1)[0] for _ in
    range(8)],
[beta.rvs(alpha_prior + int(0.1 * n), beta_prior + (n - int(0.1 * n)), size=1)[0] for _ in
    range(3)],
[beta.rvs(alpha_prior + int(0.1 * n), beta_prior + (n - int(0.1 * n)), size=1)[0]]
]
tem_profit = profit(Input_decision, def_rate)
total_profit += tem_profit
re_profit2.append(tem_profit)

avg_profit = total_profit / 1000
variance = np.var(re_profit2)

if avg_profit > 0:
positive_profits.append(re_profit2)

file.write(f'决策 {Input_decision} 的收益期望为: {avg_profit}\n')
file.write(f'决策 {Input_decision} 的收益期望方差为: {variance}\n')
```

```
print(f'决策 {Input_decision} 的收益期望为: {avg_profit}')
print(f'决策 {Input_decision} 的收益期望方差为: {variance}')

# 转换为 DataFrame
if positive_profits:
df = pd.DataFrame(positive_profits).T # 转置，使得每列为一个决策的收益

# 绘制箱型图
plt.figure(figsize=(12, 8))
sns.boxplot(data=df)

# 标注特定决策的详细信息
# for i in range(len(df.columns)):
#     # 计算中位数位置
#     median = np.median(df.iloc[:, i])
#     # 在图上添加注释
#     plt.text(i, median, f'中位数: {median:.2f}', horizontalalignment='center',
    verticalalignment='bottom', color='red', fontsize=10)

plt.title('收益期望为正值的决策方案箱型图')
plt.xlabel('决策方案')
plt.ylabel('收益期望')
plt.grid(True)
plt.show()
else:
print("没有收益期望为正值的决策方案。")
```

# 附录 E　B 问题二

```
import pandas as pd
import numpy as np
#%%
def profit(decision, product_data):
P_1, C_1, Ch_1, P_2, C_2, Ch_2, P_3, C_com, Ch_3, Price, C_change, C_dis = product_data
P_1_0, P_2_0, P_com = P_1, P_2, P_3
#首先计算第一次生产的成本
profit = 0
while True:
#首先计算该回合格品率
P_success = (1 - P_com) * (1 - (1-decision[0]) * P_1_0) * (1 - (1-decision[1]) * P_2_0)
    #不变情况下合格品比例
P_success_1 = (1 - P_com) * (1 - (1-decision[0]) * P_1) * (1 - (1-decision[1]) * P_2)
    #变化情况下合格品概率
P_1 = decision[3] * (P_1_0 * P_success_1 + (1 - decision[0]) * P_1) + (1 - decision[3]) *
```

```
    P_1_0 #零件1开始时次品率，为买来零件次品与返回零件次品,进行检测就不需要加上返回的次品
P_2 = decision[3] * (P_2_0 * P_success_1 + (1 - decision[1]) * P_2) + (1 - decision[3]) *
    P_2_0 #零件1开始时次品率，为买来零件次品与返回零件次品,进行检测就不需要加上返回的次品
P_3 = decision[3] * (1 - P_success_1) + (1 - decision[3]) * (1 - P_success) #组装的次品数
#分别计算各个选项花费
#计算开始零件购买的花费，分为拆解与不拆解
C_buy = decision[3] * (C_1 * P_success + C_2 * P_success) + (1 - decision[3]) * (C_1 + C_2)
#计算检测1，2费用，包含检测费用与检测后购买新零件费用，对输入的残次品进行检测买检测买直至没有次品，为等比数列，首项为
C_detection = Ch_1 * decision[0] + Ch_2 * decision[1] + P_1 / (1 - P_1_0) * (C_1 + Ch_1) *
    decision[0] + P_2 / (1 - P_2_0) * (C_2 + Ch_2) * decision[1]
#零件装配费用与检测和调回费用，
C_det_re = Ch_3 * decision[2] + (1 - decision[2]) * C_change * P_3
C_com_det = C_com + C_det_re
#计算拆解费用
C_total_dis = decision[3] * P_3 * C_dis
#计算总成本
C_total = C_buy + C_detection + C_com_det + C_total_dis
#计算总利润
profit_1 = Price * P_success_1 - C_total
if np.abs(profit_1 - profit) < 0.000000001 :
break
#存储利润以判别是否停止
profit = profit_1
profit = profit * (1 - decision[2] + decision[2] / (1 - P_3))
return profit

#%%
import itertools

# Generate all possible combinations of 0s and 1s for a list of length 4
combinations = list(itertools.product([0, 1], repeat=4))

# Convert each combination to a list
decision_lists = [list(combination) for combination in combinations]

product_data_total = pd.read_excel("题目数据.xlsx")
product_data_total= np.array(product_data_total.iloc[1:7,:]).tolist()
best_profit = 0

for product_data in product_data_total:
best_profit = 0
for decision in decision_lists:
tem_profit = profit(decision, product_data)
if tem_profit > best_profit :
best_decision = decision
best_profit = tem_profit
```

```
print(f"该组合最好的决策为{best_decision}")
print(f"该组合最好收益为{best_profit:.2f}元")
```

# 附录 F　问题三模拟退火

```
import pandas as pd
import numpy as np
from scipy.optimize import fsolve
import itertools
import matplotlib.pyplot as plt

import warnings

# 关闭所有的 RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# 你的代码

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu, P_x3_ru,
    P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) * (1 -
    (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))
```

```
return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars

eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) * (1 -
    (1 - x_2) * P_x2_chu)))
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu', 'P_x2_ru',
    'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_jian
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
```

```
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_1 = fsolve(three_equations，initial_guess_3，args=(decision[2][0]，decision[0][0]，
    decision[0][1]，decision[0][2]，decision[1][0]，def_rate[0][0]，def_rate[0][1]，
    def_rate[0][2]，def_rate[1][0]))
#构建第二个组合（1，2），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_2 = fsolve(three_equations，initial_guess_3，args=(decision[2][1]，decision[0][3]，
    decision[0][4]，decision[0][5]，decision[1][1]，def_rate[0][3]，def_rate[0][4]，
    def_rate[0][5]，def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_，x_1，x_2，y_1，P_x1_bu，P_x2_bu，P_zhuang_y1)
solution_1_3= fsolve(two_equations，initial_guess_2，args=(decision[2][2]，decision[0][6]，
    decision[0][7]，decision[1][2]，def_rate[0][6]，def_rate[0][7]，def_rate[1][2]))
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) * (1
    - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) * (1
    - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) * (1
    - decision[1][2])
#构建第4个组合（2，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，P_x3_bu，
    P_zhuang_y1)
solution_2_1 = fsolve(three_equations，initial_guess_3，args=(decision[4][0]，decision[1][0]，
    decision[1][1]，decision[1][2]，decision[3][0]，def_rate_1_1，def_rate_1_2，def_rate_1_3，
    def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solution
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solution
```

```
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][6],
#计算组合（2，1）单价
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solution
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) * (1
    - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
# 为每个变量赋值,对每个结果进行筛选，检查其利用fslove是否解出了不合理值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu, P_x3_ru,
    P_x3_chu = solution_2_1

if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0

return profit


#%%
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]

#定义转换函数
def decision_to_list(decision):
flat_list = []
for sublist in decision:
for item in sublist:
flat_list.append(item)
return flat_list
def list_to_decision(flat_list):
decision = []
decision.append(flat_list[:8]) # 8 elements for the first sublist
decision.append(flat_list[8:11]) # 3 elements for the second sublist
decision.append(flat_list[11:14]) # 3 elements for the third sublist
decision.append(flat_list[14:15]) # 1 element for the fourth sublist
decision.append(flat_list[15:16]) # 1 element for the fifth sublist
return decision
```

```
#定义目标函数即适应度函数
def Objective_function(gene):
gene = np.array(gene).tolist() # 确保调用 tolist() 方法
combination = list_to_decision(gene)
if combination[1][0] - combination[2][0] < 0:
return 0.001
if combination[1][1] - combination[2][1] < 0:
return 0.001
if combination[1][2] - combination[2][2] < 0:
return 0.001
if combination[3][0] - combination[4][0] < 0:
return 0.001
return profit(combination, def_rate) + 1000


#%%
import numpy as np
import random


def simulated_annealing(initial_solution, initial_temp, final_temp, cooling_rate):
current_solution = initial_solution[:]
current_fitness = Objective_function(current_solution)
temp = initial_temp

while temp > final_temp:
# 生成邻域解
neighbor_solution = current_solution[:]
mutate_point = random.randint(0, 15) # 随机选择一个位进行变动
neighbor_solution[mutate_point] = 1 - neighbor_solution[mutate_point] # 反转该位

neighbor_fitness = Objective_function(neighbor_solution)

# 判断是否接受邻域解
if neighbor_fitness > current_fitness or random.random() < np.exp((neighbor_fitness -
    current_fitness) / temp):
current_solution = neighbor_solution
current_fitness = neighbor_fitness

# 降温
temp *= cooling_rate

return current_solution, current_fitness

# 示例使用
if __name__ == "__main__":
initial_solution = [random.randint(0, 1) for _ in range(16)] # 随机生成初始解
```

```
initial_temp = 1000
final_temp = 1
cooling_rate = 0.95
iter_num = 20
best_solution, best_fitness = [] , 0
for i in range(iter_num):
tem_solution, tem_fitness = simulated_annealing(initial_solution, initial_temp, final_temp,
    cooling_rate)
if tem_fitness > best_fitness:
best_fitness = tem_fitness
best_solution = tem_solution
print("Best solution:", best_solution)
print("Best fitness:", best_fitness-1000)
```

# 附录 G　问题三先退火再遗传

```
import pandas as pd
import numpy as np
from scipy.optimize import fsolve
import itertools
import matplotlib.pyplot as plt

import warnings

# 关闭所有的 RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# 你的代码

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
```

```
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars

eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu)))
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu',
    'P_x2_ru', 'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_j
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
```

```
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_1 = fsolve(three_equations，initial_guess_3，args=(decision[2][0]，
    decision[0][0]，decision[0][1]，decision[0][2]，decision[1][0]，def_rate[0][0]，
    def_rate[0][1]，def_rate[0][2]，def_rate[1][0]))
#构建第二个组合（1，2），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_2 = fsolve(three_equations，initial_guess_3，args=(decision[2][1]，
    decision[0][3]，decision[0][4]，decision[0][5]，decision[1][1]，def_rate[0][3]，
    def_rate[0][4]，def_rate[0][5]，def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_，x_1，x_2，y_1，P_x1_bu，P_x2_bu，
    P_zhuang_y1)
solution_1_3= fsolve(two_equations，initial_guess_2，args=(decision[2][2]，decision[0][6]，
    decision[0][7]，decision[1][2]，def_rate[0][6]，def_rate[0][7]，def_rate[1][2]))
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) *
    (1 - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) *
    (1 - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) *
    (1 - decision[1][2])
```

```
#构建第4个组合（2，1）,其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_2_1 = fsolve(three_equations, initial_guess_3, args=(decision[4][0],
    decision[1][0], decision[1][1], decision[1][2], decision[3][0], def_rate_1_1,
    def_rate_1_2, def_rate_1_3, def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solut
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solut
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][
#计算组合（2，1）单价
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solut
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) *
    (1 - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
# 为每个变量赋值,对每个结果进行筛选，检查其利用fslove是否解出了不合理值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = solution_2_1

if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0

return profit


#%%
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]

#定义转换函数
def decision_to_list(decision):
flat_list = []
for sublist in decision:
```

```
for item in sublist:
flat_list.append(item)
return flat_list
def list_to_decision(flat_list):
decision = []
decision.append(flat_list[:8]) # 8 elements for the first sublist
decision.append(flat_list[8:11]) # 3 elements for the second sublist
decision.append(flat_list[11:14]) # 3 elements for the third sublist
decision.append(flat_list[14:15]) # 1 element for the fourth sublist
decision.append(flat_list[15:16]) # 1 element for the fifth sublist
return decision

#定义目标函数即适应度函数
def Objective_function(gene):
gene = np.array(gene).tolist() # 确保调用 tolist() 方法
combination = list_to_decision(gene)
if combination[1][0] - combination[2][0] < 0:
return 0.001
if combination[1][1] - combination[2][1] < 0:
return 0.001
if combination[1][2] - combination[2][2] < 0:
return 0.001
if combination[3][0] - combination[4][0] < 0:
return 0.001
return profit(combination, def_rate) + 1000


#%%
def simulated_annealing(initial_solution, initial_temp, final_temp, cooling_rate):
current_solution = initial_solution[:]
current_fitness = Objective_function(current_solution)
temp = initial_temp

while temp > final_temp:
# 生成邻域解
neighbor_solution = current_solution[:]
mutate_point = random.randint(0, 15) # 随机选择一个位进行变动
neighbor_solution[mutate_point] = 1 - neighbor_solution[mutate_point] # 反转该位

neighbor_fitness = Objective_function(neighbor_solution)

# 判断是否接受邻域解
if neighbor_fitness > current_fitness or random.random() < np.exp((neighbor_fitness -
    current_fitness) / temp):
current_solution = neighbor_solution
current_fitness = neighbor_fitness
```

```
# 降温
temp *= cooling_rate

return current_solution, current_fitness
#%%
#定义适应度函数,gene形状为（16, ）np数组
def get_fitness(population):
fitness = pd.DataFrame(population).apply(lambda x: Objective_function(x), axis=1)
return fitness

import random

# 变异
def mutation(child, mutation_rate):
# print("目前正在执行变异")
if np.random.rand() < mutation_rate:
mutate_point = random.sample(range(0, 16), 1)[0]
child[mutate_point] = 1 - child[mutate_point]

# 交叉
def crossover_and_mutation(population,population_size,crossver_rate=0.8,mutation_rate =
    0.1):
# print("目前正在执行cross_and_mutation")
new_population=[]
for father in population: # 遍历种群中的每一个个体，将该个体作为父亲
child=np.array(father.copy()) # 孩子先得到父亲的全部基因（代表一个个体的一个二进制0, 1串）
if np.random.rand() < crossver_rate: #产生子代时不是必然发生交叉，而是以一定的概率发生交叉
mother=population[np.random.randint(population_size)]# 在种群中选择另一个个体作为母亲
cross_points= np.random.randint(low=0,high=16)
child[cross_points:] = mother[cross_points:].copy()
child = child.tolist()
mutation(child,mutation_rate)
new_population.append(child)
# print("交叉互换执行完成")
return new_population

# 自然选择，优胜劣汰
def select(population,population_size,fitness,elite_num,random_size):
# print("目前正在执行select函数")
sorted_idx = np.argsort(fitness)
if elite_num > 0:
top_n_idx = sorted_idx[-elite_num:].copy()
idx =
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
idx[:elite_num] = top_n_idx.copy()
else:
idx =
```

```
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
population = pd.DataFrame(population)
population = population.iloc[idx.tolist()]
population = np.array(population)
if random_size > 0 :
population[-random_size:] = np.array(generator_random(random_size)).copy()
return population

#随机化初始种群
def generator_random(population_size):
population=[]
#二维列表，包含染色体和基因
j = 0
for j in range(population_size):
initial_solution = [random.randint(0, 1) for _ in range(16)] # 随机生成初始解
initial_temp = 1000 #模拟退火参数设置
final_temp = 1
cooling_rate = 0.95
iter_num = 1
best_solution, best_fitness = [] , 0
for i in range(iter_num):
tem_solution, tem_fitness = simulated_annealing(initial_solution, initial_temp, final_temp,
    cooling_rate)
if tem_fitness > best_fitness:
best_fitness = tem_fitness
best_solution = tem_solution
temporary = best_solution
population.append(temporary)
# print(f"主人，您好！目前帅黄仪为您生成了{population_size}个数据。")
return population

#生成报告
def print_info(population,fitness):
max_fitness_index = np.argmax(fitness)
return fitness[max_fitness_index] - 1000 , population[max_fitness_index]
#%%
population_size = 100
population = generator_random(population_size)
chromosome_length = 16
mutation_rate=0.2#变异概率
crossver_rate=0.8 #交叉互换概率
num_iter = 100
random_size = 0 #随机插入
elite_num = 10 #精英保留数


#主函数
```

```
best_cost = []
tem_result = 0
fitness = get_fitness(population) # 得到适应度
for i in range(num_iter):
fitness = get_fitness(population) # 得到适应度
population =
    np.array(crossover_and_mutation(population,population_size,crossver_rate,mutation_rate))
    # 交叉变异
fitness = get_fitness(population) # 得到适应度
print(f'第{i+1}次迭代:')
iter_best_cost,iter_best_decision = print_info(population,fitness)
iter_best_decision = list_to_decision(iter_best_decision)
print(f"本次最大收益{iter_best_cost}")
best_cost.append(iter_best_cost)
population = select(population,population_size,fitness,elite_num,random_size)

print(f"本次最好决策{iter_best_decision}")
fig = plt.figure()
ax1 = fig.add_subplot(1,1,1)
ax1.plot(best_cost,color ="black" , linestyle = "dashed")
```

# 附录 H　问题三先遗传再退火 + 退火

```
import pandas as pd
import numpy as np
from scipy.optimize import fsolve
import itertools
import matplotlib.pyplot as plt

import warnings

# 关闭所有的 RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# 你的代码

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
```

```
    P_x3_ru, P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars

eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu)))
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu',
    'P_x2_ru', 'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_j
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
```

```
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_1 = fsolve(three_equations，initial_guess_3，args=(decision[2][0]，
    decision[0][0]，decision[0][1]，decision[0][2]，decision[1][0]，def_rate[0][0]，
    def_rate[0][1]，def_rate[0][2]，def_rate[1][0]))
#构建第二个组合（1，2），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_2 = fsolve(three_equations，initial_guess_3，args=(decision[2][1]，
    decision[0][3]，decision[0][4]，decision[0][5]，decision[1][1]，def_rate[0][3]，
    def_rate[0][4]，def_rate[0][5]，def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_，x_1，x_2，y_1，P_x1_bu，P_x2_bu，
    P_zhuang_y1)
solution_1_3= fsolve(two_equations，initial_guess_2，args=(decision[2][2]，decision[0][6]，
    decision[0][7]，decision[1][2]，def_rate[0][6]，def_rate[0][7]，def_rate[1][2]))
```

```
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) *
    (1 - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) *
    (1 - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) *
    (1 - decision[1][2])
#构建第4个组合（2，1）,其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_2_1 = fsolve(three_equations, initial_guess_3, args=(decision[4][0],
    decision[1][0], decision[1][1], decision[1][2], decision[3][0], def_rate_1_1,
    def_rate_1_2, def_rate_1_3, def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solut
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solut
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][
#计算组合（2，1）单价
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solut
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) *
    (1 - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
# 为每个变量赋值,对每个结果进行筛选，检查其利用fslove是否解出了不合理值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = solution_2_1

if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0

return profit


#%%
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
```

```
[0.1,0.1,0.1],
[0.1]]

#定义转换函数
def decision_to_list(decision):
flat_list = []
for sublist in decision:
for item in sublist:
flat_list.append(item)
return flat_list
def list_to_decision(flat_list):
decision = []
decision.append(flat_list[:8]) # 8 elements for the first sublist
decision.append(flat_list[8:11]) # 3 elements for the second sublist
decision.append(flat_list[11:14]) # 3 elements for the third sublist
decision.append(flat_list[14:15]) # 1 element for the fourth sublist
decision.append(flat_list[15:16]) # 1 element for the fifth sublist
return decision

#定义目标函数即适应度函数
def Objective_function(gene):
gene = np.array(gene).tolist() # 确保调用 tolist() 方法
combination = list_to_decision(gene)
if combination[1][0] - combination[2][0] < 0:
return 0.001
if combination[1][1] - combination[2][1] < 0:
return 0.001
if combination[1][2] - combination[2][2] < 0:
return 0.001
if combination[3][0] - combination[4][0] < 0:
return 0.001
return profit(combination, def_rate) + 1000


#%%
#定义适应度函数,gene形状为（16, ）np数组
def get_fitness(population):
fitness = pd.DataFrame(population).apply(lambda x: Objective_function(x), axis=1)
return fitness

import random

# 变异
def mutation(child, mutation_rate):
# print("目前正在执行变异")
if np.random.rand() < mutation_rate:
mutate_point = random.sample(range(0, 16), 1)[0]
```

```
child[mutate_point] = 1 - child[mutate_point]

# 交叉
def crossover_and_mutation(population,population_size,crossver_rate=0.8,mutation_rate =
    0.1):
# print("目前正在执行cross_and_mutation")
new_population=[]
for father in population: # 遍历种群中的每一个个体，将该个体作为父亲
child=np.array(father.copy()) # 孩子先得到父亲的全部基因（代表一个个体的一个二进制0，1串）
if np.random.rand() < crossver_rate: #产生子代时不是必然发生交叉，而是以一定的概率发生交叉
mother=population[np.random.randint(population_size)]# 在种群中选择另一个个体作为母亲
cross_points= np.random.randint(low=0,high=16)
child[cross_points:] = mother[cross_points:].copy()
child = child.tolist()
mutation(child,mutation_rate)
new_population.append(child)
# print("交叉互换执行完成")
return new_population

# 自然选择，优胜劣汰
def select(population,population_size,fitness,elite_num,random_size):
# print("目前正在执行select函数")
sorted_idx = np.argsort(fitness)
if elite_num > 0:
top_n_idx = sorted_idx[-elite_num:].copy()
idx =
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
idx[:elite_num] = top_n_idx.copy()
else:
idx =
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
population = pd.DataFrame(population)
population = population.iloc[idx.tolist()]
population = np.array(population)
if random_size > 0 :
population[-random_size:] = np.array(generator_random(random_size)).copy()
return population

#随机化初始种群
def generator_random(population_size):
population=[]
#二维列表，包含染色体和基因
j = 0
for j in range(population_size):
temporary = np.random.randint(2, size=(16,)).tolist()
population.append(temporary)
# print(f"主人，您好！目前帅黄仪为您生成了{population_size}个数据。")
```

```
return population

#生成报告
def print_info(population,fitness):
max_fitness_index = np.argmax(fitness)
return fitness[max_fitness_index] - 1000 , population[max_fitness_index]
#%%
population_size = 100
population = generator_random(population_size)
chromosome_length = 16
mutation_rate=0.1#变异概率
crossver_rate=0.8 #交叉互换概率
num_iter = 100
random_size = 0 #随机插入
elite_num = 10 #精英保留数

#主函数
best_cost = []
tem_result = 0
fitness = get_fitness(population) # 得到适应度
for i in range(num_iter):
fitness = get_fitness(population) # 得到适应度
population =
    np.array(crossover_and_mutation(population,population_size,crossver_rate,mutation_rate))
    # 交叉变异
fitness = get_fitness(population) # 得到适应度
print(f'第{i+1}次迭代:')
iter_best_cost,iter_best_decision = print_info(population,fitness)
iter_best_decision = list_to_decision(iter_best_decision)
print(f"本次最大收益{iter_best_cost}")
best_cost.append(iter_best_cost)
population = select(population,population_size,fitness,elite_num,random_size)

print(f"本次最好决策{iter_best_decision}")
fig = plt.figure()
ax1 = fig.add_subplot(1,1,1)
ax1.plot(best_cost,color ="black" , linestyle = "dashed")
#%%
import numpy as np
import random


def simulated_annealing(initial_solution, initial_temp, final_temp, cooling_rate):
current_solution = initial_solution[:]
current_fitness = Objective_function(current_solution)
temp = initial_temp
```

```
while temp > final_temp:
# 生成邻域解
neighbor_solution = current_solution[:]
mutate_point = random.randint(0, 15) # 随机选择一个位进行变动
neighbor_solution[mutate_point] = 1 - neighbor_solution[mutate_point] # 反转该位

neighbor_fitness = Objective_function(neighbor_solution)

# 判断是否接受邻域解
if neighbor_fitness > current_fitness or random.random() < np.exp((neighbor_fitness -
    current_fitness) / temp):
current_solution = neighbor_solution
current_fitness = neighbor_fitness

# 降温
temp *= cooling_rate

return current_solution, current_fitness

# 示例使用
if __name__ == "__main__":
initial_solution = decision_to_list(iter_best_decision) # 随机生成初始解
initial_temp = 1000
final_temp = 1
cooling_rate = 0.95
iter_num = 20
best_solution, best_fitness = [] , 0
for i in range(iter_num):
tem_solution, tem_fitness = simulated_annealing(initial_solution, initial_temp, final_temp,
    cooling_rate)
if tem_fitness > best_fitness:
best_fitness = tem_fitness
best_solution = tem_solution
print("Best solution:", best_solution)
print("Best fitness:", best_fitness-1000)
```

# 附录 I　问题三穷举

```
import pandas as pd
import numpy as np
from scipy.optimize import fsolve
import itertools
import matplotlib.pyplot as plt
```

```
import warnings

# 关闭所有的 RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# 你的代码

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars

eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu)))
```

```
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu',
    'P_x2_ru', 'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_j
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
```

```
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_1_1 = fsolve(three_equations, initial_guess_3, args=(decision[2][0],
    decision[0][0], decision[0][1], decision[0][2], decision[1][0], def_rate[0][0],
    def_rate[0][1], def_rate[0][2], def_rate[1][0]))
#构建第二个组合（1，2），其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_1_2 = fsolve(three_equations, initial_guess_3, args=(decision[2][1],
    decision[0][3], decision[0][4], decision[0][5], decision[1][1], def_rate[0][3],
    def_rate[0][4], def_rate[0][5], def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu,
    P_zhuang_y1)
solution_1_3= fsolve(two_equations, initial_guess_2, args=(decision[2][2], decision[0][6],
    decision[0][7], decision[1][2], def_rate[0][6], def_rate[0][7], def_rate[1][2]))
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) *
    (1 - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) *
    (1 - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) *
    (1 - decision[1][2])
#构建第4个组合（2，1），其中的args依次分别为(y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu,
    P_x3_bu, P_zhuang_y1)
solution_2_1 = fsolve(three_equations, initial_guess_3, args=(decision[4][0],
    decision[1][0], decision[1][1], decision[1][2], decision[3][0], def_rate_1_1,
    def_rate_1_2, def_rate_1_3, def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solut
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solut
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][
#计算组合（2，1）单价
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solut
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) *
    (1 - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
# 为每个变量赋值,对每个结果进行筛选，检查其利用fslove是否解出了不合理值
```

```
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = solution_2_1

if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0

return profit

# 首先设置最优值的初值，与最优解
max_profit = 0
best_decision = None

# 生成每个子列表的 0 和 1 的组合
decision_combinations = [list(itertools.product([0, 1], repeat=len(sub_decision))) for
    sub_decision in decision]

# 生成所有可能的组合并转换为列表
all_combinations = list(itertools.product(*decision_combinations))

# 将所有组合转换为列表
all_combinations_list = [list(combination) for combination in all_combinations]

# 打印所有组合
for combination in all_combinations_list:
if combination[1][0] - combination[2][0] < 0:
continue
if combination[1][1] - combination[2][1] < 0:
continue
if combination[1][2] - combination[2][2] < 0:
continue
if combination[3][0] - combination[4][0] < 0 :
continue
# Calculate profit for the current decision combination
profit_value = profit(combination, def_rate)
# Update maximum profit and best decision if the current profit is higher
if profit_value > max_profit:
max_profit = profit_value
best_decision = combination

print(f"Maximum profit: {max_profit}")
print(f"Best decision: {best_decision}")
```

# 附录 J　问题三遗传算法

```
import pandas as pd
import numpy as np
from scipy.optimize import fsolve
import itertools
import matplotlib.pyplot as plt

import warnings

# 关闭所有的 RuntimeWarning
warnings.filterwarnings("ignore", category=RuntimeWarning)

# 你的代码

# 定义变量名称
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x3_zong_ru',
'x_zong_chu', 'P_x2_ru', 'P_x2_chu', 'P_x3_ru', 'P_x3_chu']

# 定义三个方程的方程组函数
def three_equations(vars, y_1_, x_1, x_2, x_3, y_1, P_x1_bu, P_x2_bu, P_x3_bu, P_zhuang_y1):
# 为每个变量赋值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = vars

# 定义方程
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x3_ru - ((1 - y_1_) * P_x3_bu + y_1_ / x3_zong_ru * (P_x3_bu * (x3_zong_ru -
    (x_zong_chu - 1)) + (1 - x_3) * P_x3_chu * x_zong_chu))
eq4 = P_x1_chu - (1 - x_1) * P_x1_ru
eq5 = P_x2_chu - (1 - x_2) * P_x2_ru
eq6 = P_x3_chu - (1 - x_3) * P_x3_ru
eq7 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu) * (1 - (1 - x_3) * P_x3_chu)))
eq8 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq9 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))
eq10 = x3_zong_ru - x_zong_chu / (x_3 * (1 - P_x3_bu) + (1 - x_3))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7, eq8, eq9, eq10]

# 定义两个方程的方程组的函数
def two_equations(vars, y_1_, x_1, x_2, y_1, P_x1_bu, P_x2_bu, P_zhuang_y1):
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu = vars
```

```
eq1 = P_x1_ru - ((1 - y_1_) * P_x1_bu + y_1_ / x1_zong_ru * (P_x1_bu * (x1_zong_ru -
    (x_zong_chu - 1)) + (1 - x_1) * P_x1_chu * x_zong_chu))
eq2 = P_x2_ru - ((1 - y_1_) * P_x2_bu + y_1_ / x2_zong_ru * (P_x2_bu * (x2_zong_ru -
    (x_zong_chu - 1)) + (1 - x_2) * P_x2_chu * x_zong_chu))
eq3 = P_x1_chu - (1 - x_1) * P_x1_ru
eq4 = P_x2_chu - (1 - x_2) * P_x2_ru
eq5 = x_zong_chu - (1 - y_1 + y_1 * 1 / ((1 - P_zhuang_y1) * (1 - (1 - x_1) * P_x1_chu) *
    (1 - (1 - x_2) * P_x2_chu)))
eq6 = x1_zong_ru - x_zong_chu / (x_1 * (1 - P_x1_bu) + (1 - x_1))
eq7 = x2_zong_ru - x_zong_chu / (x_2 * (1 - P_x2_bu) + (1 - x_2))

return [eq1, eq2, eq3, eq4, eq5, eq6, eq7]
variable_names = ['P_x1_ru', 'P_x1_chu', 'x1_zong_ru', 'x2_zong_ru', 'x_zong_chu',
    'P_x2_ru', 'P_x2_chu']

#定义3组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_3(x1_zong_ru,x2_zong_ru,x3_zong_ru,x_zong_chu,x_1,x_2,x_3,x1_gou,x2_gou,x3_gou,x1_jian,x2_jian,x3_j
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
x3_cost = x3_gou * (x3_zong_ru - x_chai) + x_3 * x3_zong_ru * x3_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + x3_cost + y1_cost

#定义2组单元的费用计算函数，计算出函数是该单元输出一个单位的费用，该费用也会被用来作为下一个单元的购买费用
def
    cost_2(x1_zong_ru,x2_zong_ru,x_zong_chu,x_1,x_2,x1_gou,x2_gou,x1_jian,x2_jian,y1_zhaung,y1_chai,y_1_):
x_chai = y_1_ * (x_zong_chu - 1) #计算拆解数量
x1_cost = x1_gou * (x1_zong_ru - x_chai) + x_1 * x1_zong_ru * x1_jian
    #计算单个零件的费用，分为购买费用和检测费用
x2_cost = x2_gou * (x2_zong_ru - x_chai) + x_2 * x2_zong_ru * x2_jian
    #计算单个零件的费用，分为购买费用和检测费用
y1_cost = y1_zhaung * x_zong_chu + y_1_ * y1_chai * x_chai
    #计算总装零件的费用，包括组装费用和拆解费用，检测费用在下一个组合中，最终成品的检测费用最后计算
return x1_cost + x2_cost + y1_cost
#%%
#设置变量，变量中的第一行的8个元素分别表示零配件1-8是否进行检测，
# 变量中第二行表示半成品1-3是否进行检测，变量中第三行表示半成品1-3是否进行拆解，
# 变量中第四，五行分别表示成品是否进行检测和拆解。
decision = [[0,0,0,0,0,0,0,0],
[0,0,0],
```

```
[0,0,0],
[0],
[0]]
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]
def profit(decision,def_rate):
initial_guess_3 = [0.5] * 10
initial_guess_2 = [0.5] * 7
#构建第一个组合（1，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_1 = fsolve(three_equations，initial_guess_3，args=(decision[2][0]，
    decision[0][0]，decision[0][1]，decision[0][2]，decision[1][0]，def_rate[0][0]，
    def_rate[0][1]，def_rate[0][2]，def_rate[1][0]))
#构建第二个组合（1，2），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_1_2 = fsolve(three_equations，initial_guess_3，args=(decision[2][1]，
    decision[0][3]，decision[0][4]，decision[0][5]，decision[1][1]，def_rate[0][3]，
    def_rate[0][4]，def_rate[0][5]，def_rate[1][1]))
#构建第三个组合（1，3），其中的args依次分别为(y_1_，x_1，x_2，y_1，P_x1_bu，P_x2_bu，
    P_zhuang_y1)
solution_1_3= fsolve(two_equations，initial_guess_2，args=(decision[2][2]，decision[0][6]，
    decision[0][7]，decision[1][2]，def_rate[0][6]，def_rate[0][7]，def_rate[1][2]))
#计算三个组合所提供的次品率
def_rate_1_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_1_1[1]) * (1 - solution_1_1[7]) *
    (1 - solution_1_1[9])) * (1 - decision[1][0])
def_rate_1_2 = (1 - (1 - def_rate[1][1]) * (1 - solution_1_2[1]) * (1 - solution_1_2[7]) *
    (1 - solution_1_2[9])) * (1 - decision[1][1])
def_rate_1_3 = (1 - (1 - def_rate[1][2]) * (1 - solution_1_3[1]) * (1 - solution_1_3[6])) *
    (1 - decision[1][2])
#构建第4个组合（2，1），其中的args依次分别为(y_1_，x_1，x_2，x_3，y_1，P_x1_bu，P_x2_bu，
    P_x3_bu，P_zhuang_y1)
solution_2_1 = fsolve(three_equations，initial_guess_3，args=(decision[4][0]，
    decision[1][0]，decision[1][1]，decision[1][2]，decision[3][0]，def_rate_1_1，
    def_rate_1_2，def_rate_1_3，def_rate[2][0]))
#计算每个组合的单位成本并进行组合求解最终成本
#计算组合（1，1）单价
cost_1_1 =
    cost_3(x1_zong_ru=solution_1_1[2],x2_zong_ru=solution_1_1[3],x3_zong_ru=solution_1_1[4],x_zong_chu=solut
#计算组合（1，2）单价
cost_1_2 =
    cost_3(x1_zong_ru=solution_1_2[2],x2_zong_ru=solution_1_2[3],x3_zong_ru=solution_1_2[4],x_zong_chu=solut
#计算组合（1，3）单价
cost_1_3 =
    cost_2(x1_zong_ru=solution_1_3[2],x2_zong_ru=solution_1_3[3],x_zong_chu=solution_1_3[4],x_1=decision[0][
#计算组合（2，1）单价
```

```
cost_2_1 =
    cost_3(x1_zong_ru=solution_2_1[2],x2_zong_ru=solution_2_1[3],x3_zong_ru=solution_2_1[4],x_zong_chu=solut
#计算产出成品的次品率
def_rate_2_1 = (1 - (1 - def_rate[1][0]) * (1 - solution_2_1[1]) * (1 - solution_2_1[7]) *
    (1 - solution_2_1[9])) * (1 - decision[3][0])
#计算生产出一个产品的利润
z_1 = decision[3][0] #决策是否进行检测
profit = 200 * (1 - def_rate_2_1) - z_1 * 4 - (1 - z_1) * (40 * def_rate_2_1) - cost_2_1
# 为每个变量赋值,对每个结果进行筛选，检查其利用fslove是否解出了不合理值
P_x1_ru, P_x1_chu, x1_zong_ru, x2_zong_ru, x3_zong_ru, x_zong_chu, P_x2_ru, P_x2_chu,
    P_x3_ru, P_x3_chu = solution_2_1

if (P_x1_ru < 0) or (P_x1_ru > 1) or (P_x1_chu < 0) or (P_x1_chu > 1) or (P_x2_ru < 0) or
    (P_x2_ru > 1) or (P_x2_chu < 0) or (P_x2_chu > 1) or (P_x3_ru < 0) or (P_x3_ru > 1) or
    (P_x3_chu < 0) or (P_x3_chu > 1) \
or (x1_zong_ru < 1) or (x2_zong_ru < 1) or (x3_zong_ru < 1) or (x_zong_chu < 1):
return 0

return profit


#%%
#定义次品率矩阵，方便之后进行更改，添加误差
def_rate =[[0.1,0.1,0.1,0.1,0.1,0.1,0.1,0.1],
[0.1,0.1,0.1],
[0.1]]

#定义转换函数
def decision_to_list(decision):
flat_list = []
for sublist in decision:
for item in sublist:
flat_list.append(item)
return flat_list
def list_to_decision(flat_list):
decision = []
decision.append(flat_list[:8]) # 8 elements for the first sublist
decision.append(flat_list[8:11]) # 3 elements for the second sublist
decision.append(flat_list[11:14]) # 3 elements for the third sublist
decision.append(flat_list[14:15]) # 1 element for the fourth sublist
decision.append(flat_list[15:16]) # 1 element for the fifth sublist
return decision

#定义目标函数即适应度函数
def Objective_function(gene):
gene = np.array(gene).tolist() # 确保调用 tolist() 方法
combination = list_to_decision(gene)
```

```
if combination[1][0] - combination[2][0] < 0:
return 0.001
if combination[1][1] - combination[2][1] < 0:
return 0.001
if combination[1][2] - combination[2][2] < 0:
return 0.001
if combination[3][0] - combination[4][0] < 0:
return 0.001
return profit(combination, def_rate) + 1000


#%%
#定义适应度函数,gene形状为（16，）np数组
def get_fitness(population):
fitness = pd.DataFrame(population).apply(lambda x: Objective_function(x), axis=1)
return fitness

import random

# 变异
def mutation(child, mutation_rate):
# print("目前正在执行变异")
if np.random.rand() < mutation_rate:
mutate_point = random.sample(range(0, 16), 1)[0]
child[mutate_point] = 1 - child[mutate_point]

# 交叉
def crossover_and_mutation(population,population_size,crossver_rate=0.8,mutation_rate =
    0.1):
# print("目前正在执行cross_and_mutation")
new_population=[]
for father in population: # 遍历种群中的每一个个体，将该个体作为父亲
child=np.array(father.copy()) # 孩子先得到父亲的全部基因（代表一个个体的一个二进制0，1串）
if np.random.rand() < crossver_rate: #产生子代时不是必然发生交叉，而是以一定的概率发生交叉
mother=population[np.random.randint(population_size)]# 在种群中选择另一个个体作为母亲
cross_points= np.random.randint(low=0,high=16)
child[cross_points:] = mother[cross_points:].copy()
child = child.tolist()
mutation(child,mutation_rate)
new_population.append(child)
# print("交叉互换执行完成")
return new_population

# 自然选择，优胜劣汰
def select(population,population_size,fitness,elite_num,random_size):
# print("目前正在执行select函数")
sorted_idx = np.argsort(fitness)
```

```
if elite_num > 0:
top_n_idx = sorted_idx[-elite_num:].copy()
idx =
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
idx[:elite_num] = top_n_idx.copy()
else:
idx =
    np.random.choice(np.arange(population_size),size=population_size,replace=True,p=(fitness)/fitness.sum())
population = pd.DataFrame(population)
population = population.iloc[idx.tolist()]
population = np.array(population)
if random_size > 0 :
population[-random_size:] = np.array(generator_random(random_size)).copy()
return population

#随机化初始种群
def generator_random(population_size):
population=[]
#二维列表，包含染色体和基因
j = 0
for j in range(population_size):
temporary = np.random.randint(2, size=(16,)).tolist()
population.append(temporary)
# print(f"主人，您好！目前帅黄仪为您生成了{population_size}个数据。")
return population

#生成报告
def print_info(population,fitness):
max_fitness_index = np.argmax(fitness)
return fitness[max_fitness_index] - 1000 , population[max_fitness_index]
#%%
population_size = 100
population = generator_random(population_size)
chromosome_length = 16
mutation_rate=0.2#变异概率
crossver_rate=0.8 #交叉互换概率
num_iter = 100
random_size = 10 #随机插入
elite_num = 10 #精英保留数

#主函数
best_cost = []
tem_result = 0
fitness = get_fitness(population) # 得到适应度
for i in range(num_iter):
fitness = get_fitness(population) # 得到适应度
population =
```

```
        np.array(crossover_and_mutation(population,population_size,crossver_rate,mutation_rate))
        # 交叉变异
    fitness = get_fitness(population) # 得到适应度
    print(f'第{i+1}次迭代:')
    iter_best_cost,iter_best_decision = print_info(population,fitness)
    iter_best_decision = list_to_decision(iter_best_decision)
    print(f"本次最大收益{iter_best_cost}")
    best_cost.append(iter_best_cost)
    population = select(population,population_size,fitness,elite_num,random_size)

    print(f"本次最好决策{iter_best_decision}")
    fig = plt.figure()
    ax1 = fig.add_subplot(1,1,1)
    ax1.plot(best_cost,color ="black" , linestyle = "dashed")
```

# 附录 K　output_results.txt

```
情况1:

决策 [1, 1, 0, 1] 的收益期望方差为: 4.704223390151723

决策 [1, 1, 0, 1] 的收益期望为: 15.163477083313142

情况2:

决策 [1, 1, 0, 1] 的收益期望方差为: 7.767697969848782

决策 [1, 1, 0, 1] 的收益期望为: 8.076019306462088

情况3:

决策 [1, 1, 0, 1] 的收益期望方差为: 10.01534235364243

决策 [1, 1, 0, 1] 的收益期望为: 12.66596935791401

决策 [1, 1, 1, 1] 的收益期望方差为: 4.210993266544923

决策 [1, 1, 1, 1] 的收益期望为: 12.786374436531235

情况4:

决策 [1, 1, 1, 1] 的收益期望方差为: 6.134122154562768

决策 [1, 1, 1, 1] 的收益期望为: 10.979096564467751
```

```
情况5:

决策 [0, 1, 0, 0] 的收益期望方差为：15.093227093769173

决策 [0, 1, 0, 0] 的收益期望为：8.468347672902595

决策 [0, 1, 1, 0] 的收益期望方差为：10.690873711283503

决策 [0, 1, 1, 0] 的收益期望为：8.68388367770782

决策 [1, 1, 0, 1] 的收益期望方差为：4.123904583394822

决策 [1, 1, 0, 1] 的收益期望为：8.056341257824442

情况6:

决策 [0, 0, 0, 0] 的收益期望方差为：15.145062748780655

决策 [0, 0, 0, 0] 的收益期望为：17.34613757457286

决策 [0, 1, 0, 0] 的收益期望方差为：11.166283345339325

决策 [0, 1, 0, 0] 的收益期望为：16.063340638109768

决策 [1, 0, 0, 0] 的收益期望方差为：9.354710951919547

决策 [1, 0, 0, 0] 的收益期望为：18.114585544963877

决策 [1, 1, 0, 0] 的收益期望方差为：4.978574398675048

决策 [1, 1, 0, 0] 的收益期望为：17.40488754346956
```

# 附录 L　results4-2.txt

```
决策 [(1, 1, 1, 1, 0, 0, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-11.276695687006521
决策 [(1, 1, 1, 1, 0, 0, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：1135.0596552353006
决策 [(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：56.620108127402034
决策 [(1, 1, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：4.793570232419243
决策 [(1, 1, 1, 1, 1, 0, 0, 1), (1, 0, 0), (1, 0, 0), (1,), (1,)]
```

```
    的收益期望为：-92.80737033255015
决策 [(1, 1, 1, 1, 1, 0, 0, 1), (1, 0, 0), (1, 0, 0), (1,), (1,)]
    的收益期望方差为：10882.848557173062
决策 [(0, 0, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-27.640735403887255
决策 [(0, 0, 1, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：1840.6702782262823
决策 [(1, 1, 1, 1, 1, 1, 1, 0), (1, 1, 1), (1, 1, 0), (1,), (1,)]
    的收益期望为：54.779196196355
决策 [(1, 1, 1, 1, 1, 1, 1, 0), (1, 1, 1), (1, 1, 0), (1,), (1,)]
    的收益期望方差为：7.661064091191558
决策 [(0, 0, 0, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-24.326220163677533
决策 [(0, 0, 0, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：914.6971312720182
决策 [(0, 0, 1, 0, 1, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望为：-117.34400891751795
决策 [(0, 0, 1, 0, 1, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望方差为：13416.493366884206
决策 [(1, 1, 1, 1, 1, 0, 1, 1), (1, 1, 1), (1, 0, 1), (1,), (1,)]
    的收益期望为：54.194605050647
决策 [(1, 1, 1, 1, 1, 0, 1, 1), (1, 1, 1), (1, 0, 1), (1,), (1,)]
    的收益期望方差为：8.28065132509531
决策 [(1, 1, 0, 1, 0, 1, 0, 1), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望为：-52.763626235368015
决策 [(1, 1, 0, 1, 0, 1, 0, 1), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望方差为：2339.4263592961624
决策 [(1, 1, 1, 0, 0, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-25.023023533580442
决策 [(1, 1, 1, 0, 0, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：1629.0671591331022
决策 [(1, 1, 1, 1, 0, 0, 0, 0), (1, 0, 0), (1, 0, 0), (1,), (1,)]
    的收益期望为：-115.61308480728273
决策 [(1, 1, 1, 1, 0, 0, 0, 0), (1, 0, 0), (1, 0, 0), (1,), (1,)]
    的收益期望方差为：13453.292901538052
决策 [(1, 1, 0, 0, 1, 1, 1, 0), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望为：-52.85522124808054
决策 [(1, 1, 0, 0, 1, 1, 1, 0), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望方差为：2425.6384941525635
决策 [(1, 0, 0, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-12.036429984212331
决策 [(1, 0, 0, 1, 1, 1, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：1343.881189497103
决策 [(1, 1, 1, 0, 0, 0, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望为：-23.387697321115688
决策 [(1, 1, 1, 0, 0, 0, 1, 1), (1, 1, 1), (1, 1, 1), (1,), (1,)]
    的收益期望方差为：851.058608589738
```

```
决策 [(0, 1, 1, 1, 1, 0, 0, 1), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望为：-52.46148513074923
决策 [(0, 1, 1, 1, 1, 0, 0, 1), (0, 0, 0), (0, 0, 0), (1,), (1,)]
    的收益期望方差为：2036.7713894961219
决策 [(0, 1, 0, 1, 0, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望为：-114.58481550464856
决策 [(0, 1, 0, 1, 0, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望方差为：12263.018323256307
决策 [(0, 1, 0, 0, 1, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望为：-118.14269824292856
决策 [(0, 1, 0, 0, 1, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望方差为：13983.089875011527
决策 [(1, 0, 0, 1, 0, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望为：-114.28208295296443
决策 [(1, 0, 0, 1, 0, 0, 1, 1), (0, 0, 1), (0, 0, 1), (1,), (1,)]
    的收益期望方差为：14171.03136980133
决策 [(1, 1, 1, 1, 0, 1, 1, 0), (1, 0, 0), (1, 0, 0), (1,), (1,)]
    的收益期望为：-86.12936174831407
决策 [(1, 1, 1, 1, 0, 1, 1, 0), (1, 0, 0), (1, 0, 0), (1,), (1,)]
    的收益期望方差为：9093.251668321347
```