基于马尔可夫决策与抽样检测概率分布的企业最优生产决策

摘要

生产过程中涉及许多决策问题。企业需要综合考虑行为成本，选择最优决策，从而提高预期收益或减少成本投入。本文利用概率统计相关知识，分析**抽样检测**时最小抽样检测次数，以及样本实际次品率基于抽样结果的**概率分布**；并基于**马尔可夫决策模型**，帮助企业对生产过程中的决策节点制定合理策略。该问题的研究可以帮助企业提升企业经济效益，对实际生产生活提供指导建议。

针对问题一，需要根据给定置信区间与零配件次品率标称值，求解判断零配件次品率是否超过标称值所需的最少检测次数。本文采用假设检验方法，根据问题中“接受”与“不接受”确定使用**左侧检验**或**右侧检验**，并设定允许抽样检测误差为 0.01，带入抽样统计公式计算得到，在 95% 的信度拒收条件下最少检测次数为 **2436 次**；在 90% 的信度接受条件下最少检测次数为 **1475 次**。随后本文对允许抽样检测误差进行**灵敏度分析**，发现允许抽样检测误差对于最终检测次数影响较大，允许误差越大，所需最少检测次数越少。

针对问题二，需要根据企业在生产中遇到的不同情况，制定企业生产过程各个阶段的决策策略，并给出相应的指标结果。本文提出探索式方案一与方案二，采用枚举的方法，列举在一定约束条件下的所有可能决策方案，选择预期收益最大的决策方案；最终方案三采用**马尔可夫决策过程**，结合**贝尔曼公式**递归求解成功制造并售出一个正品的期望收益，根据**状态转移过程**确定最优决策方案。由于方案一与方案二为探索式方案，故并未求解方案指标，所求最优方案请见表 3，表 4；最终方案三以成功制造并售出一个正品的期望收益为指标，具体决策方案请见表 5。

针对问题三，在问题二的基础上，进行了生产工序与零配件个数的扩展，制定在更复杂生产情境下的生产各阶段决策策略。本文将 m 道工序，n 个零件拆成与问题二相似的**若干子问题**，利用**贝尔曼公式**求解预期收益为 0 的点，设定中间半成品的价值，从而**递归求解**每个子问题中的最优决策方案，最终合并得到两道工序、8 个零配件的生产过程决策方案。本文以**成功制造并售出一个正品的期望收益**为指标，最终求解结果请见表 6。

针对问题四，题目中给出**次品率为抽样检测得到**的假设，相当于将确定的次品率修改为概率分布次品率，要求重新求解问题二与问题三。本文选择使用 **Beta 分布**表示概率的**先验分布**，结合**贝叶斯公式**，得到实际次品率的**后验概率分布**；决策方案的优劣判断指标即为抽样检测条件下的预期收益，据此求解问题二、问题三。最终求解结果请见表 7, 表 8。

关键词：马尔可夫决策过程　贝尔曼方程　抽样检测　贝叶斯公式

一、 问题重述

1.1 问题背景

某企业加工某电子产品的过程中，需要用到两种零配件（零配件 1 和零配件 2），这两种零配件需要从外部购买，再由该企业装配成成品。零配件的质量可以影响成品质量：如果零配件不合格，无论是零配件 1 还是零配件 2，都会导致成品不合格。然而，即使两个零配件都合格，由它们装配而成的成品也不一定合格。企业可以将这些不合格的成品进行报废，也可以将这些不合格成品拆解，拆解不会损害所使用的零配件，但是拆解需要付出一定的费用。

1.2 问题重述

该企业需要建立数学模型，解决以下生产过程中可能出现的问题：

问题一：该企业所需零配件（零配件 1 和零配件 2）的供应商声称，该批零配件的次品率不会超过 10%。企业准备自费检测，采取抽样检测 [1] 的方法判断该批零配件的质量，并由抽样结果决定是否从供应商手中购买这批零配件。本文需要给出检测次数尽可能少的检测方案，并判断在以下两种情形的情况下企业会如何做决定：

（1）若在 95% 的信度下认为该零配件的次品率超过了 10%，企业则不会购买。

（2）若在 90% 的信度下认为该零配件的次品率没有超过 10%，企业则购买。

问题二：本文需要根据表中所给出的不同情形中零配件、成品的不合格率以及相应成本，为企业生产过程中的各个阶段作出决策，并需要给出依据及相应指标结果。企业生产过程的四个阶段如下：

（1）决定是否对零配件进行检测。若不检测则该零配件直接进入装配环节；若检测则丢弃检测不合格的零配件。

（2）决定是否对装配好的成品进行检测。若不检测则该成品直接进入市场；若检测则只有检测合格的成品进入市场。

（3）决定是否拆解检测不合格的成品。若不拆解则直接丢弃不合格成品；若拆解则对拆解后的零配件重复步骤（1）（2）。

（4）若有用户购买到不合格产品，企业将无条件调换，并对退回的不合格品重复步骤（3）。调换过程将造成一定损失。

企业生产流程图如下：

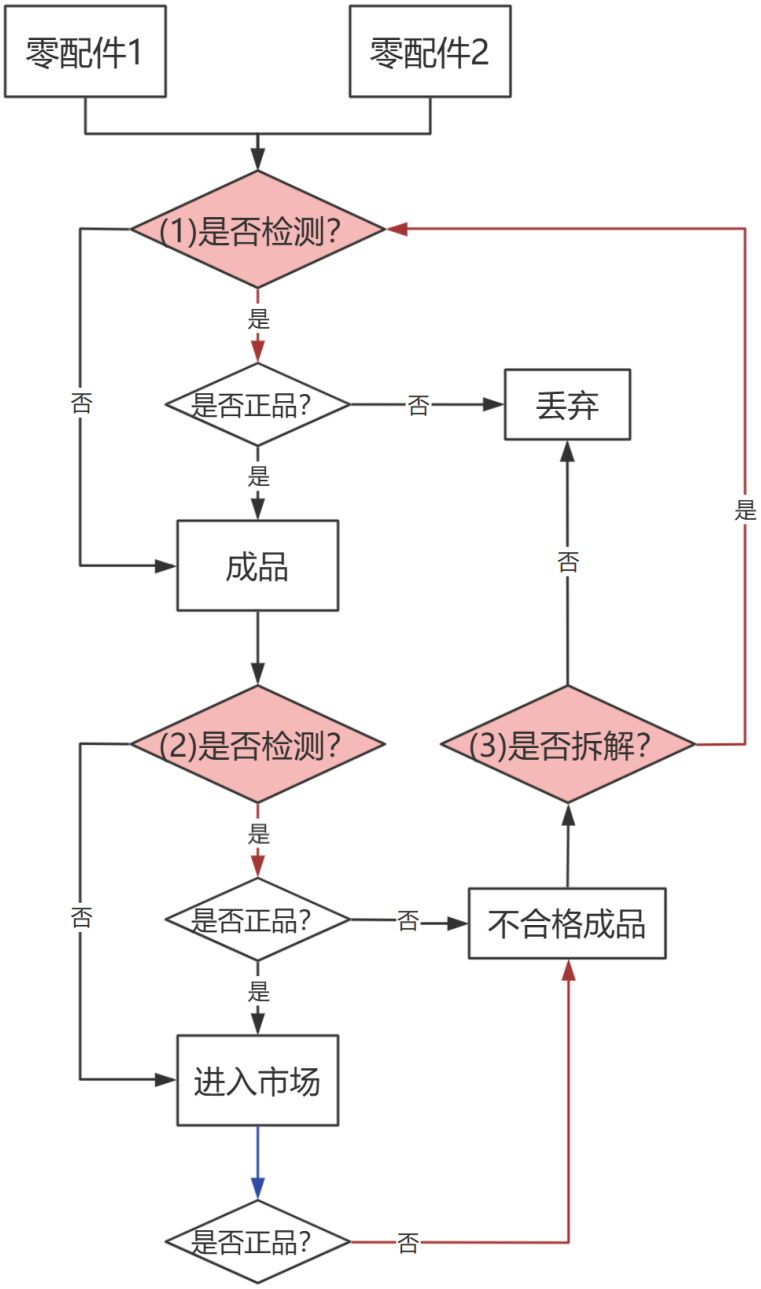


图 1　企业生产过程流程图

问题三：针对成品生产需要 2 道工序、8 个零配件的情形下，已知零配件、半成品和成品的次品率，以及相应生产步骤中的成本，再次求解问题 2，给出在该情景下生产过程的决策方案，并给出决策依据及相应指标。

问题四：若问题 2 和问题 3 中本文所已知的零配件、半成品和成品的次品率都是通过抽样检测的方法所的得到的，在这种情况下问题 2 和问题 3 所得到的决策方案应该如何更改？需要思考重新完成决策。

二、 问题分析

2.1 问题一的分析

问题一需要在给定信度和标称值的前提下，求解企业应当如何制定抽样检测方案，用尽可能少的检测次数判断次品率是否超过标称值。本文利用置信区间宽度、次品率与样本量之间的关系式，根据题目要求选择相应的单侧置信区间，带入公式求得最少样本量，即最少检测次数。

2.2 问题二的分析

问题二需要根据给定的次品率及生产各个阶段的采购、检测、装配、拆解等成本，给出各个生产阶段的最优决策。通过分析生产各个阶段，本文选择将零配件是否检测，成品是否检测，成品是否拆解当作决策节点，制定不同的决策方案。由于考虑成品拆解时会涉及到多轮决策问题，因此本文由简到繁，先探究不考虑拆解问题的决策方式，再考虑拆解不合格产品，最后考虑多轮（大于等于两轮）的多轮决策方式，制定不同方案。具体探究过程将在问题二模型的建立与求解中详述。本文探索发现，下一步的决策，只需考虑当前零件状态（例如已检验或未检验）；对于多轮决策的情况下，可以构建多个状态解决后验概率的问题，因此本文考虑使用**马尔可夫决策 [2] 过程（MDP）**处理，构建不同决策动作的状态转移矩阵，通过**贝尔曼方程**递归求解最优策略。

2.3 问题三的分析

在问题二的基础上，问题三对生产决策过程情形进行进一步讨论，将问题二中一道工序，2 个零配件的情况转变成 m 道工序，n 个零配件的拓展问题。由于增加工序和零配件数并不增加马尔可夫决策过程中状态转移动作种类，且状态设定类似，因此本文首先考虑直接沿用问题二中的方案三方法，拓展方案三中状态数量及对应迁移概率矩阵。但对一个零配件而言，至少有未拥有，未检测，先验，后验四个状态，因此总状态数量随着零配件数量变化呈指数增长，显然无法直接扩展原模型进行求解。而问题三中的每道工序都可拆解为与问题二类似的零配件组装过程，此本文考虑将每道工序隔离求解。为了保持前后工序求解过程的一致性，本文需要求解拼接得到的半成品的次品率和其预期单价，使其符合零配件具有次品率和购买成本的特征，从而能够作为零配件纳入下一道生产工序迭代。由此，本文可以通过一系列迭代过程，利用马尔可夫决策过程，在状态节点数较少的情况下求解出整体 m 道工序的最佳决策。

2.4 问题四的分析

在问题二、三的基础上，问题四将确定的次品率改成为抽样检测得到，此时无法得到具体次品率，只知道分布。考虑问题一对于抽样检测样本量的考虑，不妨认为问题四中总样本量即为问题一中所求最少检测样本量，从而利用 **Beta 分布**和**贝叶斯公式**，复现得到得到信度为 95% 和 90%，置信区间为 [9%,11%] 的**真实概率分布**。再基于**概率更新贝尔曼公式**，对真实分布模型进行大量采样，使用样本点的平均预期收益，完成问题二、问题三的重新求解。

三、 模型假设

1. 假设每批零配件样本足够大，抽样检测结果近似于正态分布。
2. 假设装配、拆解、运输过程不会影响产品零配件的质量。
3. 假设确定为正品的零配件和产品无需再次检验，且明确其质量合格。
4. 假设所有流入市场的产品都会成功出售。

四、 符号说明

符号	意义	单位
E	估计误差	–
Z_α	数据点在标准正态分布函数中, 距离均值的标准差	–
n	样本量	个
p	次品率	–
α	显著性水平	–
β	信度	–
H_0	零假设	–
H_1	备择假设	–
x_i	问题 2 的第 i 个决策变量	–
p_i	零配件 i 的次品率	–
w_i	零配件 i 的购买单价	元/件
d_i	零配件 i 的检测成本	元/件
p_c	成品的次品率	–
d_c	成品的检测成本	元/件
q_s	市场上售卖成品的合格率	–

s	成品的市场售价	元/件
r	不合格成品的调换损失	元/件
t	不合格成品的拆解费用	元/件
P_i	第 i 道工序	–
S	马尔可夫模型中生产过程状态	–
A	马尔可夫模型中生产决策动作	–
P	马尔可夫模型中迁移概率	–
R	马尔可夫模型中决策成本	元
γ	马尔可夫模型中的折扣因子	–
ω	Beta 分布中支持“成功”事件权重的形状参数	–
ϵ	Beta 分布中支持“失败”事件权重的形状参数	–

五、 模型的建立与求解

5.1 问题一模型的建立与求解

求解问题一需要根据题意，确定置信区间、置信率及次品率，利用样本量计算公式求得最少检测次数。

5.1.1 模型的建立

根据题目中所给予的已知条件，考虑重复抽样条件下，估计总体比例置信区间的估计误差 E 由 Z_α 的值、次品率 p 和样本量 n 共同决定。它们之间的数量关系如下：

$$n = \frac{(Z_\alpha)^2 \cdot p(1-p)}{E^2} \tag{1}$$

在该题中，理论次品率 p 与供应商标称值相等，估计总体比例置信区间的误差由本文预先敲定，公式中 Z_α 表示的是数据点在标准正态分布函数中，距离均值的标准差的倍数。它用于衡量数据点相对于总体均值的偏离程度。Z_α 值通常与采用的统计检验方法与显著性水平有关。

显著性水平是假设检验中的一个参数。假设检验是指从对参数做出假设开始，收集样本数据，计算出样本统计量。假设检验还可分为单侧检验（单尾检验）和双侧检验（双尾检验），而单侧检验又可以分为左侧检验和右侧检验，其中左侧检验关心的是数据是否比某个阈值小，即关注的是正态分布的左尾；而右侧检验关心的是数据是否比某个阈值大，即关注的是正态分布的右尾。本文需要根据标称值，给出实际次品率的参数假设，由此计算出最少抽样检测次数。而假设检验的基本思想是小概率事件原理，它指小概率

事件在一次随机试验中几乎不可能发生，而这种小概率事件发生的概率被称之为“显著性水平”，在本题中由题目所需信度 β 决定：

$$\alpha = 1 - \beta \tag{2}$$

本文采用假设检验中单侧检验的方法，结合题目中所给信度求得显著性水平 α 的值，从而确定 Z_α 值。

样本量为本文求解的内容。考虑到本题求解内容为最少抽样检测次数，且题目中仅给出企业决定是否接收该批零配件的标称值界限。因此，为了方便计算，本文对上述公式进行数学推导，得到如下不等式：

$$n \geq \left(\frac{Z_\alpha \times \sqrt{p(1-p)}}{E} \right)^2 \tag{3}$$

本文用该不等式计算最少抽样检测次数。

由于标称值确定，因此样本量主要由估计误差来确定。估计误差越小，所需检测样本量越大。查询国家统计局《估计总体比例时样本量的确定》一文，本文了解到大多数情况下，E 的取值一般应小于 0.1, 考虑到本文给出的标称值较小，故依照标称值的 0.1 给出估计值误差，并由此计算一定允许估计误差条件下所需的样本量。

5.1.2 模型的求解

估计误差标定：本文在估计误差 E 为 0.01 的情形下，对最少抽样检测次数进行求解。

总体比例求解：在本题中，总体比例在数值上与零配件次品率相等，但本文并不知道实际次品率，只知道供应商的标称值。因此不妨将标称值 10% 带入总体比例，利用不等式决定用于判断实际次品率是否超过标称值阈值的最少抽样检测次数。

Z_α **值求解**：本文采用假设检验的方法，将问题看作是一个基于比例的假设检验问题，做出假设：

- 零假设 H_0：零配件的次品率 $p \leq 10\%$，即次品率不超过标称值 10%
- 备择假设 H_1：零配件的次品率 $p > 10\%$，即次品率超过了标称值 10%

由于本文关注的是次品率是否超出标称值，而非最靠近标称值两侧的概率，因此本文选择单侧检验类型。

对于第一种情形，本文关注次品率是否超出标称值 10%，因此本文采用右尾检验。由题目给定的信度，根据显著性水平计算公式 $\alpha = 1 - \beta$(信度)，可以求得第一题的显著性水平 α_1 为 0.05。再根据正态近似模型，利用标准正态分布查表确定临界值 $Z_{\alpha 1}$ 为 1.645。

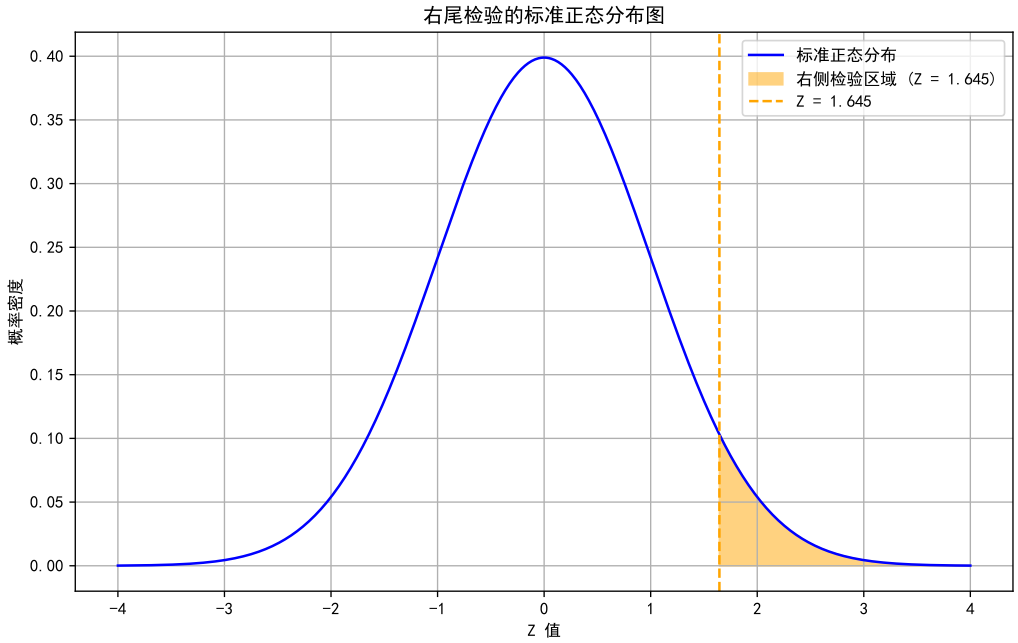


图 2　右尾检验

对于第二问，本文关注次品率是否小于等于标称值 10%，因此采用左尾检验。由题目给定的信度，根据显著性水平计算公式 $\alpha = 1 - \text{信度}$，可以求得第二题的显著性水平 α_2 为 0.10。再根据正态近似模型，利用标准正态分布查表确定临界值 $Z_{\alpha 2}$ 为 1.28。

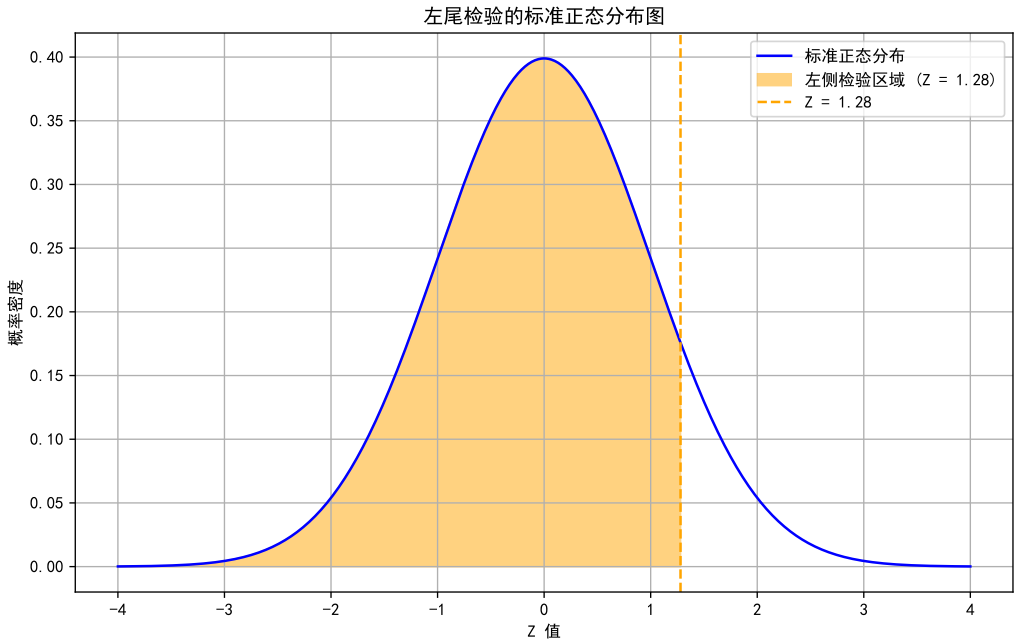


图 3　左尾检验

5.1.3 模型求解结果

根据本文确定的不同估计误差 E 值，最终求得在设定估计误差容忍条件下，最少抽样次数结果如表 2：

表 2　确定估计误差 E 下两种情形抽样次数和置信区间

E	95% 信度抽样次数	95% 信度置信区间	90% 信度抽样次数	90% 信度置信区间
0.01	**2436**	$[10\%, 11\%]$	**1475**	$[9\%, 10\%]$

即在情形一 **95%** 的信度下认定零配件次品率超过标称值，则拒收次品零件抽样检测 **2436** 个样品，在情形二 **90%** 的信度下认定零配件次品率不超过标称值，则接受这批零件抽样检测 **1475** 个样品。

5.2 问题二模型的建立与求解

关于生产过程各阶段决策节点的确定，以及基本方案设计出发点已在问题分析中给出，故此处不再赘述。不同决策节点间会相互影响：当设定成品一定不拆解时，无需考虑零配件多轮复用的问题，可以简化情况；而当考虑成品拆解问题时，某些生产阶段会进行多轮。在后续轮次考虑配件复用时，作为不合格产品的零配件，其更有可能是次品，即零配件次品率的后验概率更大；或者由于先前轮次已检验零配件 1 或 2 的质量，其次品率为 0。本文采用探究式解题方案，对于本题中较为复杂的**成品拆解**与**多轮决策**问题，由简入繁，先设计不考虑成品拆解与多轮复用的方案一作为简单方案；再在方案一的基础上开始考虑成品拆解和多轮决策的问题，但限制在前两轮中将零配件 1 与零配件 2 均检测完毕，保证在第二轮后所用的配件均为合格品；最后结合方案一与方案二的探究结果，制定考虑情形较为全面的方案三。接下来介绍本文在探究过程中收获的创新灵感：

企业对于不合格产品的退换问题：本文注意到步骤（4）说明企业会无条件退换用户得到的不合格品，并支付一定的调换损失，同时重复步骤（3）。若按常规思路分析，当用户买到不合格产品并进行调换时，需要考虑调换后的产品是否依旧为不合格产品。因此，如果一直聚焦于退换货行为，需要考虑多轮退换问题，这显然不利于讨论。本文发现可以将退换过程转化为退货与新交易的过程：第一步，企业承担不合格产品所造成的损失，予以退货。相比于卖出一个合格品，此步的损失为：

$$\text{调换第一步损失成本} = \text{调换损失} + \text{市场售价} \tag{4}$$

此时企业既损失了出售正品得到的市场售价，也承担了调换损失。第二步，让顾客购买一个新的产品。此时因为不确定是否再次取到的是不合格品，又可以看成一个全新的交

易，这样避免了从多轮的角度考虑调换的行为，在考虑产品的预期收益时便只需要考虑合格卖出和不合格退货的情况，而不需要考虑多轮调换后的总预期收益。

不合格产品检验问题：若不对成品进行检验，则不合格产品有可能流入市场，此时顾客会对不合格产品进行退换。因此对于所有组装而成的不合格产品，需要讨论由生产决策检验而找出的不合格产品，以及由顾客退回的不合格产品。前者需要支付成品检测费用 d_c，后者需要支付调换损失 r。考虑到本文假设 “所有为合格品的成品都可以出售，即销售环节获利只需要考虑成品的比例或数量”，两种决策就收益而言都可以获得全部合格品的销售费用，区别仅在于执行两个操作上的成本。因此，可以考虑将顾客退回理解为顾客检测：即对于不合格产品的检测分为由企业检测与由顾客检测，且它们的费用不同。基于预期期望的求解考虑，可以认为顾客检测费用为

$$\text{顾客检测费用} = \text{调换损失} \times \text{成品次品率} \tag{5}$$

企业检测费用由题目给定。对于每一种策略，根据零件 1、零件 2 与装配时的次品率，可以直接求得成品次品率，进而计算出顾客检测费用。比较顾客检测与企业检测费用，即可确定生产过程决策中关于成品检测节点的决策，可以减少需要考虑的节点，简化模型。

不合格产品拆解问题：对于不合格产品，决定是否对其进行拆解是很重要的问题。从产品生产的角度，无论对不合格产品是直接丢弃还是进行拆解，都是为了得到新的零配件，比较新零配件的价值即可判断是否进行拆解。而零配件的价值可以通过次品率与购买价格判断。对于给定的决策方案，若成品为不合格品，其拆解所得零件的后验次品率为定值。因此本文在方案三中并不考虑不合格产品拆解概率的计算，而是针对每种情况，预先计算对于该种决策是否应该拆解不合格产品，给出固定状态转换，简化模型。

5.2.1 方案一：不考虑拆解成品（探究性方案）

当选择不拆解成品时，无需考虑配件多轮复用的问题，可以简化情况。因此本文制定方案一决策，将不合格成品全部丢弃，不考虑成品是否拆解，即步骤（3）中本文默认丢弃。此时本文只需要考虑零配件 1、零配件 2 是否检测，以及成品是否检测的决策节点。本文对不同决策分别求解期望收益，对比得出最佳策略。方案一生产决策流程如下：

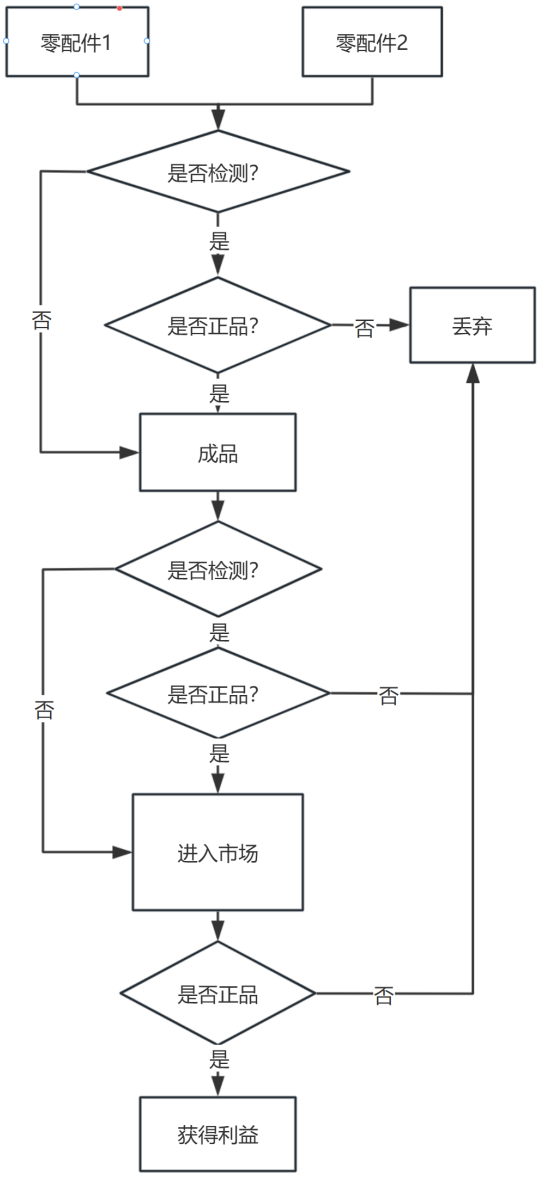


图 4　方案一生产决策流程

首先本文需要计算零配件 1 在检测前后次品率发生的变化。若不进行检测，则零配件 1 的次品率为该情况下给定的次品率 p_1; 若进行检测，则零配件 1 的次品率为 1 或 0。因此，本文定义决策变量 x_1，表示在第一步决策中的决定（即零配件 1 是否进行检测）。若 $x_1 = 1$ 则表示对零配件 1 进行检测；若 $x_1 = 0$ 则表示不对零配件 1 进行检测。因此本文利用下述公式计算在装配过程中零配件 1 的合格率 q_1:

$$q_1 = (1 - p_1)^{1-x_1} \tag{6}$$

关于零配件 2 在检测前后次品率的变化，分析过程与零配件 1 基本相同。题目已给定零配件 2 的次品率 p_2，本文定义决策变量 x_2，表示在第二步决策中的决定（即零配

件 2 是否进行检测)。若 $x_2 = 1$ 则表示对零配件 2 进行检测；若 $x_2 = 0$ 则表示不对零配件 2 进行检测，利用下述公式计算在装配过程中零配件 2 的合格率 q_2:

$$q_2 = (1 - p_2)^{1-x_2} \tag{7}$$

根据装配时零配件 1 与零配件 2 的合格率，本文可以计算出成品的次品率。成品不合格的情境可分为两种情况：一种是零配件 1 或零配件 2 不合格，则导致成品一定不合格；一种是零配件 1 和零配件 2 均合格，但成品依旧不合格，这种情况的概率为题目给定值 p_c。基于上述两种情况，本文可以给出成品实际次品率 p_f 的计算公式:

$$p_f = (1 - q_1 \cdot q_2) + q_1 \cdot q_2 \cdot p_c \tag{8}$$

由于生产决策节点中有对成品质量的检验，因此在市场中实际售卖产品的合格率 q_s 与成品实际次品率 p_f 并不为简单的 $q_s = 1 - p_f$ 的关系。参考零配件 1 与零配件 2 在装配阶段合格率的计算方法，本文定义决策变量 x_3，表示在第三步决策中的决定（即对成品是否进行检测），市场中实际售卖产品的合格率可以由下述公式求解:

$$q_s = (1 - p_f)^{1-x_3} \tag{9}$$

基于上述生产决策分析，本文已知每步生产决策所需的检测成本，可以计算得到一轮生产决策后的总检测成本 $cost$。由于该方案不需要对不合格成品进行拆解，因而不存在多轮生产决策的情况。根据第一步生产决策（即是否对零配件 1 进行检测）的检测成本 d_1，第二步生产决策（即是否对零配件 2 进行检测）的检测成本 d_2，第三步生产决策（即是否对成品进行检测）的检测成本 d_3，可使用公式

$$cost = x_1 \cdot d_1 + x_2 \cdot d_2 + x_3 \cdot d_3 \tag{10}$$

来计算总检测成本，而预期收益的计算方法应该为

$$profit = q_s \cdot s - p_f \cdot r - cost \tag{11}$$

本文根据对该方法进行分析后，以决策组合：对零配件 1、零配件 2 及成品均不进行检测的预期收益为基准，计算其它决策与基准决策的预期收益之差 $\Delta profit$。本文设定 $p_{s.j}$ 为第 j 种决策组合（即关于零配件 1、零配件 2、成品检验与否的决策）的市场实际售卖产品合格率，$p_{s.0}$ 为基准决策的市场实际售卖产品合格率，通过产品合格与否所造成的收益差 $s + t$，来计算由于市场实际售卖产品合格率的差值造成的预期收益的差值。本文所使用的预期收益差值计算公式为

$$\Delta profit = (p_{s.j} - p_{s.0}) \cdot (s + r) - \Delta cost \tag{12}$$

由于本方案为探究性方案，并非最终求解方式，因此仅根据与基准决策的预期收益只差决定最优决策，并未计算绝对预期收益。求得方案一在各个情况下的最优生产决策如 表 3:

表 3　生产方案一最优决策

情况	零配件 1 是否检测	零配件 2 是否检测	成品是否检测	不合格品是否拆解
1	否	否	是	否
2	否	否	是	否
3	否	否	是	否
4	否	否	是	否
5	否	否	是	否
6	否	否	是	否

5.2.2 方案二：考虑拆解不合格成品（探究性方案）

当考虑拆解不合格成品时，本文需要考虑配件多轮复用问题。因此本文制定方案二，在收到不合格产品时考虑是否进行拆解。此时本文需要考虑问题分析中所提及的所有生产决策节点，根据预期收益决定是否检测或拆解。方案二生产决策流程如下：

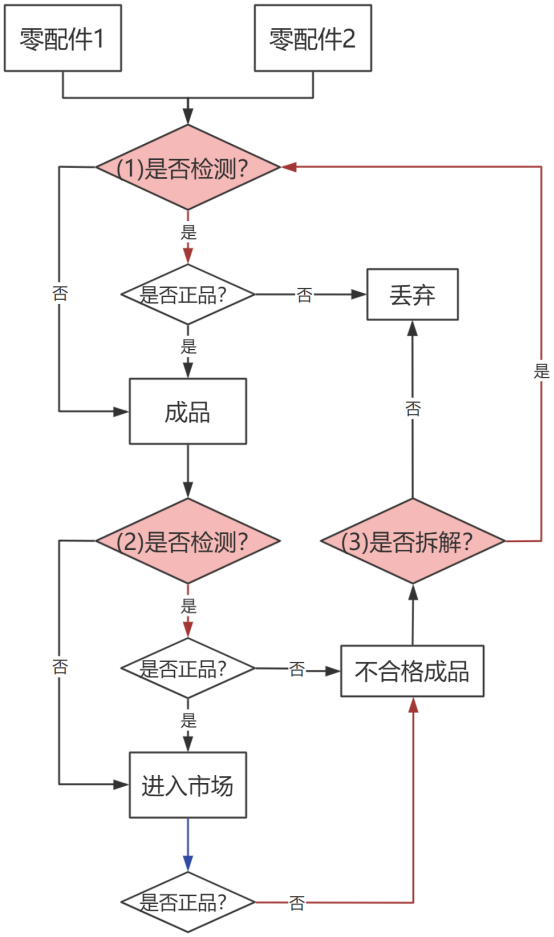


图 5　方案二生产决策流程

方案二中存在部分与方案一求解步骤相似的部分，因此不再赘述，仅在此处简单提及。方案二中也需要决策是否对零配件 1、零配件 2 进行检测，得出在装配过程中零配件 1 的合格率 q_1 与零配件 2 的合格率 q_2

本文可以由零配件 1 的合格率 q_1 与零配件 2 的合格率 q_2 得出成品实际次品率 p_f

$$p_f = (1 - q_1 \cdot q_2) + q_1 \cdot q_2 \cdot p_c \tag{13}$$

再由题目中所给定的成品合格率计算方法，求得市场中实际售卖产品的合格率

$$q_s = (1 - p_f)^{1-x_3} \tag{14}$$

然而在方案二中，对于不合格产品，需要决策是否对其进行拆解。本文认为拆解行为与非拆解行为可以理解为：当进行拆解时，本文可以得到经过或未经过检验的零配件 1 和零配件 2 （是否经过检验由先前决策决定），但是需要支付额外的拆解费用；当不进行拆解时，本文把原成品直接丢弃，与拆解时相比，相当于本文需要重新购买未经检验的零配件 1 和零配件 2。

因此，是否拆解不合格产品的判断条件应为比较拆解与直接购买所得到的零配件 1 与零配件 2 的成本差异。本文将零配件的成本定义为 “获得一个明确合格零配件的价格”，即：

$$\text{零配件成本} = \frac{\text{零配件售价}}{\text{零配件合格率}} \tag{15}$$

需要注意的是，对于拆解后得到的零配件，由于本文已知由零配件 1 与零配件 2 组装而成的成品为不合格品，则若该零配件在先前决策中未经过检验，其后验概率会大于先验概率，即此时零配件的次品率会大于题中所给定的原次品率，在决策是否进行拆解时需要考虑。结合本文中拆解决策的判定方式，本文认为，当下述不等式成立时，对不合格成品进行拆解：

$$\sum_{i=1}^{2} \frac{\text{零配件 i 售价}}{1 - \text{零配件 i 次品率}} > \sum_{i=1}^{2} \frac{\text{零配件 i 售价}}{1 - \text{零配件 i 后验次品率}} + \text{零配件拆解成本} \tag{16}$$

对上述公式进行一系列数学推导后，可以得到成品拆解决策的判断不等式：

$$0 < \sum_{i=1}^{2} (1 - \frac{p_i}{1 - p_f} \cdot (1 - x_i)) \cdot \frac{w_i}{1 - p_i} - t \tag{17}$$

其中 w_i 为零配件 i 的购买成本（i 可以取值 1 或 2）。当该不等式成立时，对不合格产品进行拆解时，期望收益更大。

而当对不合格产品进行拆解后，拆解后得到的零配件 1 和零配件 2 需要重新投入新一轮生产。本文考虑在第一轮决策的基础上，在第二轮将所有未经检测的零配件全部进行检验，确保在第二轮装配时所有零件均检验合格。此时成品的合格与否直接为给定成品次品率 p_c。因此，综合考虑两轮情况，方案二的在市场中售卖产品的成品合格率 p_{st}

应为第一轮直接合格的情况与第一轮不合格，但第二轮合格的情况概率之和。第一轮市场产品合格概率为 p_{s1}，第二轮市场产品合格概率为 p_{s2}。合格率的计算公式如 (14)。则可以得到两轮综合市场中成品合格率

$$p_{st} = p_{s1} + (1 - p_{s1}) \cdot p_{s2} \tag{18}$$

由于方案二中保证当第二次装配时，零配件 1 与零配件 2 都是合格元件，因此在后续 n 轮中，成品合格率都为固定值 p_c，一定可以找到某个极限装配次数 n，在该点后成品合格率无线趋近于 1，因此对于后续轮次无需再做过多分析.

零配件 1 与零配件 2 的检测费用分别为 d_1 与 d_2，且最多被检测一次；成品的检测成本为 d_c，在多轮中可能被多次检验，因此用 n_1 记录成品总检测轮数；每进入新轮次都需要再次拆解，本文用 n_2 记录总拆解次数，其在数值上与决策轮次相等。因此，可以得到总体检测成本 $cost$ 的计算公式

$$cost = x_1 \cdot d_1 + x_2 \cdot d_2 + x_3 \cdot d_3 \cdot n_1 + x_4 \cdot t \cdot n_2 \tag{19}$$

其中 $x_1x_2x_3x_4$ 分别表示对于零配件 1、零配件 2、成品、不合格产品的检测或拆解决策，1 表示执行，0 表示不执行。

与方案一类似，根据预期收益的计算方式 (11)，以决策组合：对零配件 1、零配件 2 及成品均不进行检测，并不拆解不合格产品的预期收益为基准，计算其它决策与基准决策的预期收益之差 $\Delta profit$。本文设定 $p_{st.j}$ 为第 j 种决策组合（即关于零配件 1、零配件 2、成品检验与否、不合格产品拆解与否的决策）的市场实际售卖产品合格率，$p_{st.0}$ 为基准决策的市场实际售卖产品合格率，通过产品合格与否所造成的收益差 $s + t$，来计算由于市场实际售卖产品合格率的差值造成的预期收益的差值。与方案一不同，方案二中存在多轮决策的问题，因此需要处理每轮成品组装的成本 w_3。基准方案中一定只进行一轮决策，利用其它方案进行决策时所记录**总拆解次数** n_2，给出相对预期收益计算公式

$$\Delta profit = (p_{st.j} - p_{st.0}) \cdot (s + r) - cost - w_3 \cdot n_2 \tag{20}$$

由于本方案为探究性方案，并非最终求解方式，因此仅根据与基准决策的预期收益只差决定最优决策，并未计算绝对预期收益。求得方案二在各个情况下的最优生产决策如表 4：

表 4　生产方案二最优决策

情况	一轮零配件 1 是否检测	一轮零配件 2 是否检测	一轮成品是否检测	不合格品是否拆解	二轮零配件 1 是否检测	二轮零配件 2 是否检测	二轮成品是否检测
1	否	否	是	是	是	是	是
2	否	否	是	是	是	是	是
3	否	否	是	是	是	是	是
4	否	否	是	是	是	是	是
5	否	否	是	是	是	是	是
6	否	否	是	否	-	-	-

5.2.3 方案三：基于马尔可夫决策过程与贝尔曼方程的最优决策模型

基于上述两个探究方案，本文最终得出基于马尔可夫决策过程与贝尔曼方程的最优决策模型。本题涉及零配件 1 与零配件 2 经过多道工序的生产过程，其次品率会由于生产过程决策而发生变化。但整体而言，只要设计合适的状态节点，就可以表示所有可能出现的零配件状态，保证每个决策的结果只依赖于当前所在节点，类似于自动机，符合马尔可夫链的无记忆特性。因此，本文利用马尔可夫链建模该生产过程，并使用马尔可夫决策过程（MDP）解决问题。

马尔可夫决策过程是包括状态、动作、状态转移概率和成本（或奖励）的框架，在每个状态下本文根据动作选择，最小化期望成本（或最大化期望奖励）。本文通过合理的零配件与成品检测和拆解决策来最小化总成本。

根据马尔可夫决策过程基本内容，本文需要根据零配件 1、零配件 2 与成品状态定义状态节点。考虑到当配件检测不合格时，需要更换新的未检验零配件，因此零配件应存在未拥有状态，本文假定为 “状态 0”；对于新购买的未检验零配件，其次品率均为题目给定次品率 p，本文假定为 “状态 1”；检测后的零配件次品率为 0，本文假定为 “状态 2”；对于成品状态，本文在前文中已对**不合格产品检验问题**进行解释，可以认为成品必将经过检验，且检验代价为成品检测代价与顾客检测代价中的最小值，因此本文仅设置 “合格状态” 与 “不合格状态”，而不具体分析成品检测生产决策；对于决策拆解不合格产品得到的零配件 1 与零配件 2，其后验次品率与新购买的零配件次品率不同，考虑马尔可夫链 “无记忆特性”，本文仅考虑经过一次拆解后的后验次品率，设置此时零配件状态为 “状态 3”。由于本文需要定义迭代出口，因此设定状态 “售卖”。综上，本文定义如下状态 S：

- $S_{0,0}$：未拥有零配件 1，未拥有零配件 2
- $S_{0,1}$：未拥有零配件 1，未检测先验次品率零配件 2
- $S_{0,2}$：未拥有零配件 1，已检测零配件 2
- $S_{0,3}$：未拥有零配件 1，未检测后验次品率零配件 2
- $S_{1,0}$：未检测先验次品率零配件 1，未拥有零配件 2
- $S_{1,1}$：未检测先验次品率零配件 1，未检测先验次品率零配件 2
- $S_{1,2}$：未检测先验次品率零配件 1，已检测零配件 2
- $S_{1,3}$：未检测先验次品率零配件 1，未检测后验次品率零配件 2
- $S_{2,0}$：已检验零配件 1，未拥有零配件 2
- $S_{2,1}$：已检验零配件 1，未检测先验次品率零配件 2
- $S_{2,2}$：已检验零配件 1，已检测零配件 2
- $S_{2,3}$：已检验零配件 1，未检测后验次品率零配件 2
- $S_{3,0}$：未检测后验次品率零配件 1，未拥有零配件 2
- $S_{3,1}$：未检测后验次品率零配件 1，未检测先验次品率零配件 2

- $S_{3,2}$：未检测后验次品率零配件 1，已检测零配件 2
- $S_{3,3}$：未检测后验次品率零配件 1，未检测后验次品率零配件 2
- $S_{\text{合格}}$：产品检验合格
- $S_{\text{不合格}}$：产品经检验不合格
- $S_{\text{售卖}}$：产品售出

本文还需要定义各个状态之间的转移动作 A。基于马尔可夫模型中设定的状态 S，在生产决策过程中，主要涉及零配件 1 购买、零配件 2 购买、零配件 1 检验、零配件 2 检验、成品检验（此处本文将企业检验与顾客检验合并解释为成品检验）、不合格成品拆解、不合格产品不拆解、产品售卖的动作。

在定义各个状态之间转移动作 A 的基础上，还需要定义马尔可夫决策过程中的动作迁移概率矩阵 P。每个动作的动作迁移概率矩阵如下：

- 设定**零配件 1 购买**动作的迁移概率矩阵为 P_0，仅有 $S_{0,0}$，$S_{0,1}$，$S_{0,2}$，$S_{0,3}$ 状态到 $S_{1,0}$，$S_{1,1}$，$S_{1,2}$，$S_{1,3}$ 节点间有单向边，且概率均为 1。
- 设定**零配件 2 购买**动作的迁移概率矩阵为 P_1，仅有 $S_{0,0}$，$S_{1,0}$，$S_{2,0}$，$S_{3,0}$ 的节点到 $S_{0,1}$，$S_{1,1}$，$S_{2,1}$，$S_{3,1}$ 节点间有单向边，且概率均为 1。
- 设定**零配件 1 检验**动作的迁移概率矩阵为 P_2，仅有 $S_{1,0}$，$S_{1,1}$，$S_{1,2}$，$S_{1,3}$ 的节点与 $S_{3,0}$，$S_{3,1}$，$S_{3,2}$，$S_{3,3}$ 的节点到 $S_{2,0}$，$S_{2,1}$，$S_{2,2}$，$S_{2,3}$ 或 $S_{0,0}$，$S_{0,1}$，$S_{0,2}$，$S_{0,3}$ 节点间有单向边，且概率由零配件 1 先验或后验次品率决定。
- 设定**零配件 2 检验**动作的迁移概率矩阵为 P_3，仅有 $S_{0,1}$，$S_{1,1}$，$S_{2,1}$，$S_{3,1}$ 的节点与 $S_{0,3}$，$S_{1,3}$，$S_{2,3}$，$S_{3,3}$ 的节点到 $S_{0,2}$，$S_{1,2}$，$S_{2,2}$，$S_{3,2}$ 或 $S_{0,0}$，$S_{1,0}$，$S_{2,0}$，$S_{3,0}$ 节点间有单向边，且概率由零配件 2 先验或后验次品率决定。
- 设定**成品检验**动作的迁移概率矩阵为 P_4，仅当 $S_{1,0}$，$S_{1,1}$，$S_{1,2}$，$S_{1,3}$，$S_{2,0}$，$S_{2,1}$，$S_{2,2}$，$S_{2,3}$，$S_{3,0}$，$S_{3,1}$，$S_{3,2}$，$S_{3,3}$ 节点到 $S_{\text{合格}}$ 或 $S_{\text{不合格}}$ 状态之间有单向边，迁移概率由上一个状态的成品次品率 p_f 决定。
- 设定**不合格产品拆解**动作的迁移概率矩阵为 P_5，仅当由 $S_{\text{不合格}}$ 状态转移至 $S_{3,3}$ 的节点有单向边，且概率为 1。
- 设定**不合格产品不拆解**的动作迁移概率矩阵为 P_6，仅当由 $S_{\text{不合格}}$ 状态转移至 $S_{0,0}$ 节点间有单向边，且概率为 1。
- 设定**产品售卖**动作的迁移概率矩阵为 P_7，仅当由状态 $S_{\text{合格}}$ 到状态 $S_{\text{售卖}}$ 有单向边，且概率为 1。

马尔可夫决策过程还需要提供动作转移至不同状态所对应的奖励或代价矩阵。由于本文以最小化总成品的方式求解最优决策，因此选择求解代价矩阵 R。本文在前文中已对**不合格产品检验问题**进行解释，可以认为成品检验代价为成品检测代价与顾客检测代价中的最小值。仅考虑在迁移概率矩阵中概率不为零的边，每个动作决定代价矩阵中的部分取值：

- **零配件 1 购买**动作对应代价矩阵中节点代价均为零配件 1 购买成本 w_1。
- **零配件 2 购买**动作对应代价矩阵中节点代价均为零配件 2 购买成本 w_2。
- **零配件 1 检验**动作对应代价矩阵中节点代价均为零配件 1 检验成本 d_1。
- **零配件 2 检验**动作对应代价矩阵中节点代价均为零配件 2 检验成本 d_2。
- **成品检验**动作对应代价矩阵中的节点代价为基于上一个节点确定的零配件 1 与零配件 2 的不合格率，求得组装后成品的合格率，计算得到的成品检测代价与顾客检测代价中的最小值。
- **不合格产品拆解**动作对应代价矩阵中节点代价均为不合格产品拆解费用 t。
- **不合格产品不拆解**动作对应代价矩阵中节点代价均为 0。
- **产品售卖**动作对应的是“奖励”，而非“代价”，因此本文设定该动作对应代价矩阵中节点代价为 $-s$

每个动作所对应的状态节点转移已在分析动作迁移概率矩阵 P 时给出，故此处不再赘述。本题所求得的具体状态转移图、动作迁移概率矩阵与代价矩阵可在支撑材料中查询。

在准备好马尔可夫决策决策所需基本内容后，即可带入**贝尔曼方程**中求解最优决策。贝尔曼方程根据动态规划 [3] 的思想，通过迭代更新每个决策的代价函数，最终找到该条件下的最优决策。针对当前状态 S，此时的值函数 V 取决于该状态下采取的下个动作 A，求解当前状态值函数的最优值 $V_{(S)}$ 需要当前决策的成本加上未来期望成本的总和。贝尔曼方程中 $C_{(S,A)}$ 为在状态 S 时执行动作 A 的成本，$P_(S'|S,A)$ 是在状态 S 时选择执行动作 A 后，转移到状态 S' 的概率，$V_(S')$ 为未来状态 S' 的最优值函数。由于本文通过最小化决策成本的方式来制定最优策略，因此使用的贝尔曼方程为：

$$V(S) = \min_{a \in A(S)} \left[C(S,a) + \gamma \sum_{S'} P(S' \mid S,a) V(S') \right] \tag{21}$$

本文使用**价值迭代法**求解 (21)。

首先初始化所有状态的值函数 $V_{0(S)}$ 为 0，再对于每个状态，计算采取不同动作 A 后的成本 $C_{(S,A)}$，再根据状态转移概率 $P(S' \mid S,a)$ 与将要跳转到的状态对应值函数 $V_(S')$，计算跳转到未来状态 S' 的期望成本。针对状态 S 所有可能的动作，选择使得总成本最小的动作 A 进行跳转，并更新值函数。

随着迭代次数增加，马尔可夫链不断延长，由于贝尔曼方程中折扣因子 γ 的存在，迭代次数越多，未来状态的期望值函数 $V_(S')$ 对当前状态的影响越小，迭代过程将更加注重眼前代价而非未来预期代价，因此当前状态的值函数最终一定会收敛，当值函数的变化范围小于某个本文设定的阈值时，即可停止迭代过程，求得最终的最优策略。

价值迭代过程初始时刻，企业生产决策作为较长生产过程中的重要部分，值函数的求解应更加注重未来预期收益，即在未来作为成品卖出的收益，因此本文设置初始折扣

因子 0.99，表示未来收益几乎与当前收益等同；收敛域设置为 10^{-5}，当值函数变化范围小于该收敛域则迭代停止。

5.2.4 模型求解结果

方案三为本问最终决策结果。求得方案三在各个情况下的最优生产决策如表 5：

表 5　方案三的生产最优决策

轮数	第一轮				第二轮				预期收益
决策	是否进行检测			是否拆解	是否进行检测			是否拆解	
	零配件 1	零配件 2	成品		零配件 1	零配件 2	成品		
1	是	否	否	是	是	是	否	是	16.68
2	是	否	否	是	是	是	否	是	6.78
3	否	否	是	是	是	否	是	是	14.77
4	是	否	是	是	是	是	是	是	9.57
5	否	是	否	是	是	是	否	是	10.70
6	否	否	否	否	—	—	—	—	20.26

5.3 问题三模型的建立与求解

问题三模型与问题二中方案三模型基本一致，仅增加对于中间产品“价值”的求解以及多道工序的迭代过程。

5.3.1 拓展原马尔可夫模型

结合问题分析的思路，针对本题中 m 道工序、n 个零配件的情形，本文将零配件经过一系列操作组装成半成品，与半成品组装成成品的过程转换成问题二中零配件经过一系列生产决策，组装成成品并获得一定预期收益的过程。

针对上述提到的两个过程，建立模型需要完成两个主要的操作：

1. 设定半成品（中间产物）合理的价值。当该半成品作为“成品”时，其价值对应售价；当该半成品作为“零配件”时，其价值对应成本。
2. 通过决策和预期收益修正经过第一道工序得到的半成品的次品率和其预期单价，使其符合零配件具有次品率和购买成本的特征。

拓展模型需要进行如下操作：

1. **针对第一个操作的市场售价**：由于最后的结果只需要给出最优的决策方案，无需具体的预期收益（或预期成本），因此只需要设定一个足够大的市场售价，使整个马尔科夫链可以进行状态的改变，而不是由于预期收益为负数而停止。
2. **针对第一个操作的调换损失**：半成品实际上没有调换损失的操作，考虑到之前成本检验的设定，这里需要让 顾客检测费用 = 调换损失 × 成品次品率足够大，即给调换损失一个足够大的数，使其会选择半成品检验而不是调换损失。
3. **针对第二个操作的次品率**：在得到半成品的组装策略后，便可以直接算出它的次品率，符合需要。
4. **针对第二个操作的成本**：考虑到零配件的购买单价表示的是购买一个零配件所花费的金额，而不是得到一个正品零配件的花费。因此需要用以下公式：

$$\text{等价零配件购买单价} = \text{半成品预期成本} \times \text{半成品次品率} \tag{22}$$

其中等价零配件购买单价是半成品满足花费购买单价但可能得到次品的零配件性质的拟合，因为半成品的预期成本得到的实际上是得到一个合格的半成品需要的成本，因此要乘上半成品次品率。

使用上述调整后，所有的零配件拼接得到半成品或前一道工序的拼接成后一道工序的半成品的过程都可以视为问题二中零配件拼接得到成品并进行检测，售卖操作的过程。

本文结合生成过程和问题 2 的思路, 继续发现第 i 道工序 $P_i(0 \leq i<m)$ 的零配件或半成品之间的预期收益 (或成本) 互不干扰, 可仅由第 $i-1$ 道工序 $P_{i-1}(0 \leq i<m)$ 对应组装它的零配件或半成品的预期收益 (或成本) 得到, 同时作用于它合成的第 $i+1$ 道工序 $P_{i+1}(0 \leq i<m)$ 的半成品或成品, 且同一道工序内的零配件或半成品的预期收益 (或成本) 相互独立。因此，本文将第 $i-1$ 道工序 $P_{i-1}(0 \leq i \leq m)$ 内对应零件，半成品组装成第 i 道工序 $P_i(0 \leq i \leq m)$ 的对应半成品或成品的生产过程看成一个一道工序，若干零配件合成成品的独立过程。

由以上操作，本文可以将可以将工作流程以工序数量 i 和具体组装情况分解成一个个层层递进的子问题, 子问题得到的半成品的次品率和等价零配件购买单价又可以作为下一个子问题的初始条件，则此时在每个小过程中都需要得到最大的预期收益，拓展问题二中模型规模，修改参数就可以进行求解这一系列问题。

综上，**问题三的模型建立** 可以转换为以下几步：

1. 从已给出次品率和购买成本的零配件开始，将已经得到等价零配件购买单价和次品率的第 i 道工序 P_i（$0 \leq i < m$）和第 $i+1$ 道工序 P_{i+1} 中对应组装合成的一组作为子问题单独考虑。
2. 优化问题二马尔可夫决策过程与贝尔曼方程的最优决策模型对步骤 1 给出的子问题求解。

3. 得到组装出 $i+1$ 道工序 P_{i+1} 的一个半成品的决策和以及对应的次品率和等价零配件购买单价。若此时满足 $i+1=m$ 则进行步骤 4，若不满足，则跳到步骤 1。
4. 输出得到的最大的预期收益，综合得到的每一步决策，得到整个过程的决策。

子过程迭代求解轮数优化：本文考虑到在问题二使用马尔可夫决策过程与贝尔曼方程的最优决策模型由于不考虑生产中的时间成本和机器损耗成本，最后得到的是生成出一个成品的预期收益，因此问题三在使用相同模型时得到的半成品预期收益所耗费的时间成本和机器损耗成本被忽略了，生产一个半成品时会进行多轮检测，拆解，购买，结算的操作，这显然不符合提倡生产效率的实际情况。由于此题并未给出具体可计算的时间成本和机器损耗成本，因此选择修改折扣因子 γ 来减少未来奖励或成本对当前决策的影响，这样可以让每一步决策都更加考虑最近几步的影响，从而减少得到生产一个半成品的预期收益（成本）的轮数，使其更加符合实际情况。

对于具体给出的两道工序，8 个零配件的问题，直接使用上述模型。

5.3.2 模型调整的其他操作

零配件的次品率、购买成本与检测成本，和半成品的次品率和检测都已给定。为了得到半成品的购买成本，本文首先对其进行假设，用一个假设购买成本作为售价带入贝尔曼方程，通过贝尔曼方程迭代优化，最终得到状态 $S_{0,0}$ 的预期收益来判断是否到达收益的临界值。为了方便计算，本文假定购买成本均为整数。如果某一个售价使得状态 $S_{0,0}$ 的预期收益较明显，说明该售价显著大于其购买成本，产生了较大的盈利空间。而如果售价使得成功卖出后的预期收益接近于 0，则说明盈利空间几乎没有，售价刚好覆盖成本，其他因素如次品和退换货只是对预期收益的细微调整。因此，可以通过调整假设购买成本和对比预期收益得到半成品的购买成本，以及通过确定零配件到半成品生产过程中的决策计算得到半成品的次品率，用于第二道工序——半成品组装为成品的预期收益计算。

本文认为，在第一道工序中，若半成品作为成品参与过至少一轮质量检测，可以说明该半成品必为合格品，在第二道工序中半成品作为零配件则无需再次进行质量检测，付出检测成本。因此，在进行马尔可夫决策前，需要先判断该半成品的合格率。若合格率为 1，那么当该半成品的状态为 1 时，直接将该状态的最优策略确定为对该半成品进行检测，但在计算当前状态值函数时并不加上检测该半成品的成本，保留当前状态值函数然后对下一个状态进行重新判断；若合格率不为 1，则仍使用价值迭代法，将贝尔曼方程作为优化目标函数作出总成本最小的决策，实现预期收益的最大化。

5.3.3 模型求解结果

模型在该情形下的最优生产决策如下：

表 6　第一道工序的生产最优决策

轮数	第一轮					第二轮				
决策	是否进行检测				是否拆解	是否进行检测				是否拆解
	零配件 1	零配件 2	零配件 3	成品		零配件 1	零配件 2	零配件 3	成品	
半成品 1	是	是	否	是	是	是	是	是	是	是
半成品 2	是	是	否	是	是	是	是	是	是	是
半成品 3	是	否	—	是	是	是	是	—	是	是
成品[1]	是	是	是	是	是	是	是	是	是	是

[1] 第二道工序组装成成品时，“零配件”应为“半成品”

在这样的决策方案下，最大预期收益为 45.09699078。

5.4 问题四模型的建立与求解

5.4.1 真实次品率概率分布求解

由问题一可知，在已知次品率平均值而不知其具体分布的情况下，若要在某种信度下得到某个置信区间，可以选择增加检验次数来逼近；因此此时从反方向考虑，只要给定一定信度和检验次数的结果，就可以反向得到置信区间和次品率概率分布。在这里，本文选择使用 Beta 分布表示概率的先验分布，并使用贝叶斯公式得到具体的次品率概率分布和置信区间。下面给出次品率 p 的后验分布求解公式：

$$p \sim \text{Beta}(\omega + k, \epsilon + n - k) \tag{23}$$

k 是抽样检测到的次品数，n 为检测的样本数，ω 和 ϵ 是先验分布的参数，其中 ω 通常可以理解为成功的次数（或支持“成功”事件的权重），越大表示对成功”的期望越强，即 Beta 分布的密度函数在靠近 1 的地方值更大。其中 ϵ 通常可以理解为失败的次数（或支持“失败”事件的权重）。越大表示对“失败”的期望越强，即 Beta 分布的密度函数在靠近 0 的地方值更大。

通过贝叶斯定理，即可根据上述公式得到次品率概率分布。

之后可利用以下公式计算得到置信区间上下限：

$$\text{CI}_{\text{lower}} = \text{Beta}^{-1}\left(\frac{\alpha}{2}, \omega + k, \epsilon + n - k\right) \tag{24}$$

$$\text{CI}_{\text{upper}} = \text{Beta}^{-1}\left(1 - \frac{\alpha}{2}, \omega + k, \epsilon + n - k\right) \tag{25}$$

其中 α 表示显著性水平。

考虑到初始时是无信息先验，对次品率的大小没有任何先检偏好, 因此取 $\omega = \epsilon = 1$。

5.4.2 基于概率的贝尔曼方程更新

之后即可以通过改变样本数量 n 和显著性水平 α 来得到不同的次品率概率分布和置信区间，考虑到无法直接根据置信区间得到修正概率后的最大预期收益对应的策略，因此在置信区间内抽取多个概率带进去计算，每个节点都进行一次基于概率的计算，最后得到次品率为抽样检测条件下的最优方案和相应指标结果，此时的贝尔曼方程更新为:

$$V(S) = \min_{a \in A(S)} \frac{\sum_{i=0}^{n} \left[C_i(S,a) + \gamma \sum_{S'} P_i(S' \mid S,a) V(S') \right]}{n} \tag{26}$$

其中 $C_i(S,A)$ 为第 i 个抽样结果在状态 S 时执行动作 A 的成本，$P_i(S'|S,A)$ 为第 i 个抽样结果在状态 S 时选择执行动作 A 后，转移到状态 S' 的概率，$V_(S')$ 为未来状态 S' 的最优值函数。n 为总的抽样结果。

5.4.3 模型的求解

考虑联系到第一问，选择构造双边检验条件下，置信区间为 $[9\%, 11\%]$ 的信度 β 分别为 95% 和 90% 的检测方案，使用以下公式即可求解:

$$n \geq \left(\frac{Z_{\frac{\alpha}{2}} \times \sqrt{p(1-p)}}{E} \right)^2 \tag{27}$$

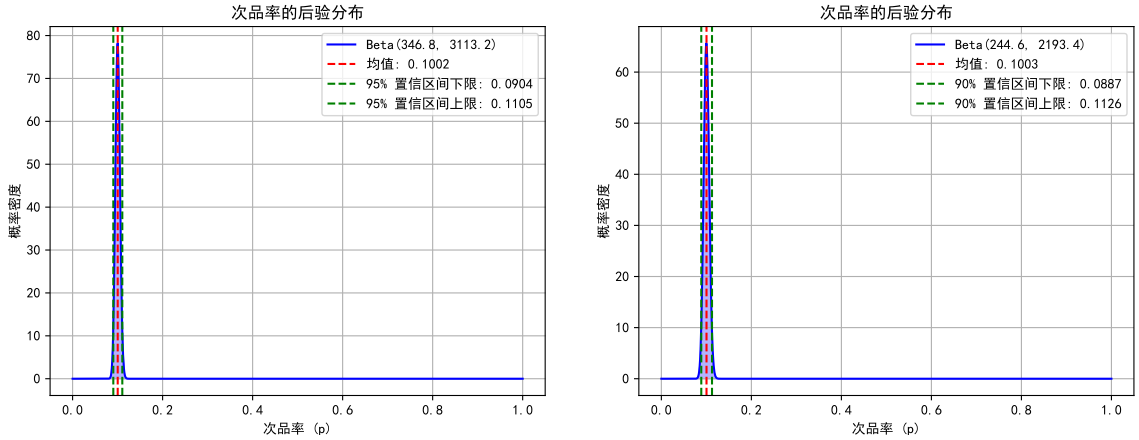
此时解得显著性水平 α 分别为 5%,10%，查找对应的 $Z_{\frac{\alpha}{2}}$ 为 1.96 和 1.645，此时的 E 仍为 0.01, 代入解得 n 为 3458 和 2436.

进一步使用 Beta 分布和贝叶斯公式得到具体的次品率概率分布图, 如图6a和图6b所示:

可以看出，此时两种信度下的置信区间都近似为 $[9\%, 11\%]$。从置信区间内采样 10000 个概率，在问题二和问题三的基础上利用公式 (26) 对答案进行求解，得到次品率通过抽样检测方法得到时，最大的预期成本的步骤。

5.4.4 模型求解结果

表 7和表 8分别为 95% 信度下和 90% 信度下问题二最优决策方案及相应指标结果，表 9和表 10分别为 95% 信度下和 90% 信度下问题三最优决策方案及相应指标结果，其中黑色加粗地方为与 100% 信度（即问题二原条件）下最优决策方案不同的地方。由此可看出，在置信区间相同的情况下，信度不同表示两种概率分布方式也不同，从而会导致尽管每一步概率期望相同，但是一些状态的最优决策不同，进而导致总的最优策略和



(a) 信度为 95% 时次品率概率分布图像　　**(b) 信度为 90% 时次品率概率分布图像**

图 6　信度为 90% 时次品率概率分布图像

指标结果不同。说明实际生产中，不仅要考虑标称值，还需要考虑如信度，置信区间等指标。

表 7　95% 信度下问题二最优决策方案

轮数	第一轮				第二轮				预期收益
决策	是否进行检测			是否拆解	是否进行检测			是否拆解	
	零配件 1	零配件 2	成品		零配件 1	零配件 2	成品		
1	是	否	否	是	是	是	否	是	16.67
2	是	否	否	是	是	是	否	是	6.79
3	**是**	**否**	是	是	是	**是**	**否**	是	14.79
4	是	否	是	是	是	是	是	是	9.61
5	否	是	否	是	是	是	否	是	10.68
6	否	否	否	否	—	—	—	—	20.28

表 8　90% 信度下问题二最优决策方案

轮数	第一轮				第二轮				预期收益
决策	是否进行检测			是否拆解	是否进行检测			是否拆解	
	零配件 1	零配件 2	成品		零配件 1	零配件 2	成品		
1	是	否	否	是	是	否	否	是	16.66
2	是	否	否	是	是	否	是	是	6.74
3	是	否	是	是	是	是	否	是	14.78
4	是	否	是	是	是	否	是	是	9.50
5	否	是	否	是	**否**	是	**是**	是	10.65
6	否	否	否	否	—	—	—	—	20.30

表 9　95% 信度下问题三最优决策方案

轮数	第一轮					第二轮				
决策	是否进行检测				是否拆解	是否进行检测				是否拆解
	零配件 1	零配件 2	零配件 3	成品		零配件 1	零配件 2	零配件 3	成品	
半成品 1	是	是	否	是	是	是	是	是	是	是
半成品 2	是	是	否	是	是	是	是	是	是	是
半成品 3	是	否	—	是	是	是	是	—	是	是
成品[1]	是	是	是	是	是	是	是	是	是	是

[1] 第二道工序组装成成品时，“零配件”应为“半成品”

表 10　90% 信度下问题三最优决策方案

轮数	第一轮					第二轮				
决策	是否进行检测				是否拆解	是否进行检测				是否拆解
	零配件 1	零配件 2	零配件 3	成品		零配件 1	零配件 2	零配件 3	成品	
半成品 1	是	是	否	是	是	是	是	是	是	是
半成品 2	是	是	否	是	是	是	是	是	是	是
半成品 3	是	否	—	是	是	是	是	—	是	是
成品[1]	是	是	是	是	是	是	是	是	是	是

[1] 第二道工序组装成成品时，“零配件”应为“半成品”

六、 模型的分析与检验

6.1 模型的灵敏度分析

在问题一计算最少抽样检测次数的求解模型中，利用公式

$$n \geq \left(\frac{Z_\alpha \times \sqrt{p(1-p)}}{E} \right)^2 \tag{28}$$

求解样本量 n 时，涉及允许估计值误差 E。在给定标称值 10% 的情况下，取估计误差为 $[0.001, 0.2]$ 间的点利用 python 绘制函数图像，得到在不同 E 值与信度的情况下，估计误差 E 与抽样样本量 n 的关系如图 7，图 8：

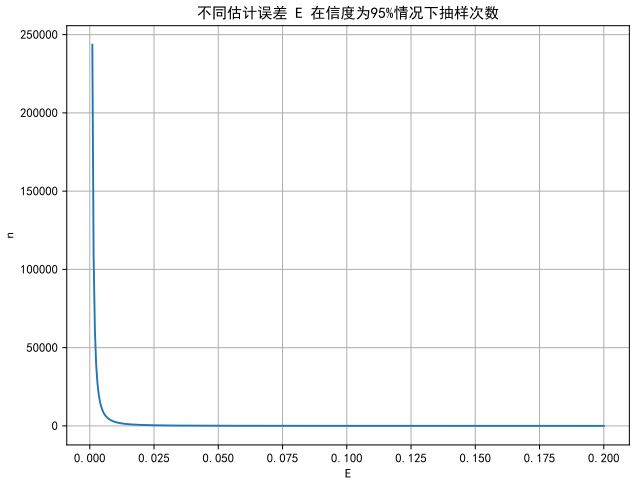


图 7　不同估计误差 E 在信度为 95% 情况下抽样次数

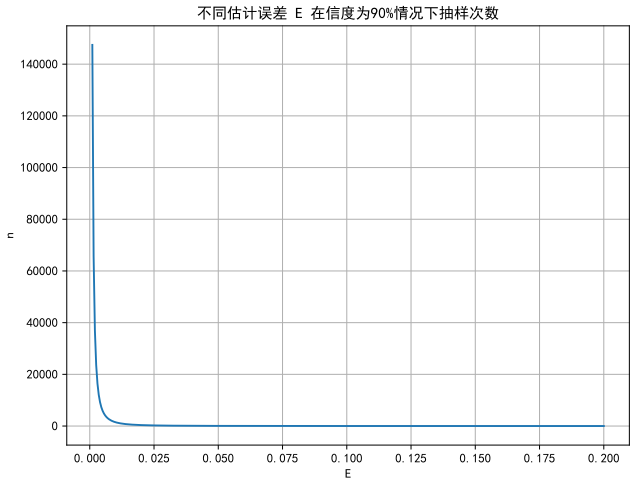


图 8　不同估计误差 E 在信度为 90% 情况下抽样次数

由图像可以看出，当允许估计误差很小时，随着允许估计误差增大，抽样次数明显减少，参数 E 的灵敏性较高；而当允许估计误差逐渐增大，抽样样本量 n 的减少速度变缓；且当允许估计误差值为 0.15 左右时，抽样次数几乎不随允许估计样本率变化而变化，且抽样次数趋近于 0 次。

因此，当参数 E 偏小时，每减少一点允许误差都需要耗费很大成本；当 E 偏大时，抽样检测结果根本无法体现零配件实际次品率。考虑到国家统计局《估计总体比例时样本量的确定》一文中的误差估计值建议，以及模型求解抽样次数结果，本文建议在设定估计误差 0.01 的情形下决策最少抽样次数。

6.2 马尔可夫模型合理性检验

在问题二模型的建立与分析中，本文已分析得出：当所有零配件的购买成本与零配件、成品的检测成本一定时，是否对不合格产品进行拆解的决策是预先确定的，并不取决于先前决策。根据公式 (17)，可以求得当前情况下是否需要拆解。

问题二中，马尔可夫模型求得最佳策略由表 5给出。仅检验第一轮决策：已知零配件 1 、零配件 2 与成品是否检验，可以直接确定当前状态是否应该拆解。本文将由马尔可夫模型求出的最佳策略与由公式 (17) 直接求得的拆解决策对比，结果如表 11：

表 11　马尔可夫求解最佳策略与直接由公式计算所得拆解决策对比

情况	零配件 1 是否检测	零配件 2 是否检测	成品是否检测	不合格品是否拆解	公式计算是否拆解
1	是	否	否	是	是
2	是	否	否	是	是
3	否	否	是	是	是
4	是	否	是	是	是
5	否	是	否	是	是
6	否	否	否	否	否

由上表可知，马尔可夫求解的最佳策略，在拆解决策中与本文探究结果完全吻合，这在一定程度上可以证明马尔可夫模型的合理性。

七、 模型的评价、改进与推广

7.1 模型的优点

1. 本文模型建立始终考虑了后验概率，符合实际工业生产中如分类放置零件等改变次品率操作的实际情况，使模型的稳定性更强和更具有现实意义。

2. 本文模型建立为一个探索性的过程，所进行的假设较少，更多基于题目给出的条件，在一步步探索中不断优化方法，条件，通过文献和结果相互检验，提高了模型的科学性、可行性。

3. 本文模型建立的过程中不断挖掘决策细节，简化计算，使得模型更加精简。

4. 本文第三问中提出的模型可以解决 m 道工序，n 个零件的普遍情况，引入其他影响参数后还可以解决更加复杂的问题，可推广性强。

7.2 模型的缺点

1. 对于第三问的模型建立中，考虑到无法直接用模型处理整个问题，选择了拆分成若干子问题，并做一些近似，可能会降低结果的准确性。

2. 模型建立中对实际生产中客观存在的时间成本，机器损耗成本考虑较少，尽管考虑通过 γ 的调整优化轮数，但是有时候仍然会出现组装轮数过多的情况，与实际生产存在矛盾。

7.3 模型的改进

1. 考虑更多影响生产过程的参数，如时间成本，提高模型的科学性、准确性。2. 可以增加马尔可夫决策中状态个数，考虑更多轮次后的零配件后验概率。

7.4 模型的推广

1. 本文的模型可以解决实际生产中涉及到装配，检测，拆解的加工问题。2. 在调整奖励或代价矩阵后，可以运用拆分，迭代的思想解决类似使用马尔可夫链模拟的问题（满足马尔可夫链无记忆性的问题）。

参考文献

[1] 李卿卿. 面板区间值数据模型的估计和假设检验[D].

[2] 薄胜, 李媛, 刘海伦. 基于马尔可夫决策的钢铁产成品订单分配模型研究[J].

[3] 姜河. 基于自适应动态规划的非线性控制理论与优化方法的研究[D].

附录 A　支撑材料

```
Problem1.py
Problem2.py
Problem2_Bellman.py
bellman.py
bellman_3.py
normal_distribution_photo1.py
normal_distribution_photo2.py
迁移概率矩阵和代价函数.xlsx
图片文件夹figure
文件文件夹file
```

附录 B　问题 1

```
import math

def calculate_size(E, Z, p):
    """
    计算单侧检验所需的样本量 n

    参数:
    E -- 允许误差
    Z -- 单侧置信度对应的 Z 值（如 1.645 对应 95% 单侧置信度）
    p -- 标称值

    返回:
    所需的样本量 n（向上取整）
    """
    n = (pow(Z,2) * p * (1 - p)) / pow(E,2)
    return math.ceil(n) # 取整


E1 = 0.005 #
    考虑到四舍五入,允许的误差为0.5%,最后的次品率设定在9.5%~10.5%（根据单侧区间进行修改）
E2 = 0.01 #最后的次品率在9%~11%（根据单侧区间修改）
E3 = 0.05  #最后的次品率在5%~15%（根据单侧区间修改）
Z1 = 1.645 # 对应95%单侧置信度的Z值(上侧单侧区间，右尾检验)
Z2 = 1.28 # 对应90%单侧置信度的Z值（下侧单侧区间，左尾检验）
p = 0.1 # 给出的标称值为10%

# 计算样本量
size1 = calculate_size(E1, Z1, p)
```

```
size2 = calculate_size(E1,Z2,p)
print(f"此时允许误差E为: {E1}")
print(f"95%信度下认定零配件次品率超过标称值则拒收，所需的样本量: {size1}")
print(f"此时样本区间为[10%,10.5%]")
print(f"90%信度下认定零配件次品率不超过标称值则接受，所需样本量:{size2}")
print(f"此时样本区间为[9.5%,10%]")
print("\n")

size1 = calculate_size(E2, Z1, p)
size2 = calculate_size(E2,Z2,p)
print(f"此时允许误差E为: {E2}")
print(f"95%信度下认定零配件次品率超过标称值则拒收，所需的样本量: {size1}")
print(f"此时样本区间为[10%,11%]")
print(f"90%信度下认定零配件次品率不超过标称值则接受，所需样本量:{size2}")
print(f"此时样本区间为[9%,10%]")
print("\n")

size1 = calculate_size(E3, Z1, p)
size2 = calculate_size(E3,Z2,p)
print(f"此时允许误差E为: {E3}")
print(f"95%信度下认定零配件次品率超过标称值则拒收，所需的样本量: {size1}")
print(f"此时样本区间为[10%,15%]")
print(f"90%信度下认定零配件次品率不超过标称值则接受，所需样本量:{size2}")
print(f"此时样本区间为[5%,10%]")
```

附录 C　问题 2 的方案 1 和方案 2

```
import itertools

def pass_rate(p, x):
    return (1-p)**(1-x)

def total_reject_rate(q1, q2, p3):
    return (1-q1*q2)+q1*q2*p3

def strategy_cost(d1, x1, d2, x2, d3, q3, r, x3, x4=0):
    if x4==0:
        return x1*d1+x2*d2
    return x1*d1+x2*d2+x3*d3+x4*r
    # dcost = min(q3*r, d3)
    # if dcost==d3:
    #     x3, x4 = 1, 0
    # elif dcost==q3*r:
    #     x3, x4 = 0, 1
    # return x3, x4, x1*d1+x2*d2+dcost
```

```python
def if_disassemble(p1, p2, pf, w1, w2, t, x1, x2):
    pp1 = p1 / pf * (1-x1) # 零配件1的后验次品率
    pp2 = p2 / pf * (1-x2)
    w1_real = w1 / (1-p1)
    w2_real = w2 / (1-p2)
    if t >= (1-pp1)*w1_real + (1-pp2)*w2_real:
        return pp1, pp2, w1_real, w2_real, 0
    else:
        return pp1, pp2, w1_real, w2_real, 1

def cal_benchmark(p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t, x1=0, x2=0, x3=0, x4=0):
    # q1 = pass_rate(p1, x1)
    # q2 = pass_rate(p2, x2)
    # pf = total_reject_rate(q1, q2, p3)
    # total_pass_rate = pass_rate(pf, x3)
    # benchmark = total_pass_rate*(s+r)
    # print(total_pass_rate)
    # print(benchmark)
    total_pass_rate = (1-p1)*(1-p2)*(1-p3)
    # print(f'total_pass_rate:{total_pass_rate} s:{s} r:{r} w1:{w1} w2:{w2} d3:{d3}')
    benchmark = total_pass_rate*s-(1-total_pass_rate)*r-w1-w2-d3 # 000的预期收益
    # print(f'total_pass_rate:{total_pass_rate} s:{s} r:{r} w1:{w1} w2:{w2} d3:{d3}
        benchmark:{benchmark}')

    return benchmark

def round2_x3(situa, strategy_2, benchmark_2, pf_1, strategies):
    pf_21, drate, dprofit_20 = func(*(situa+strategy_2+[0]),benchmark=benchmark_2, scheme=2,
        pf_1=pf_1, x3_p=strategies[2]) # 收益
    pf_22, drate, dprofit_21 = func(*(situa+strategy_2+[1]),benchmark=benchmark_2, scheme=2,
        pf_1=pf_1, x3_p=strategies[2]) # 收益
    if dprofit_20 > dprofit_21:
        dprofit_2 = dprofit_20
        pf_2 = pf_21
        strategy_2=0
    else:
        pf_2 = pf_22
        dprofit_2 = dprofit_21
        strategy_2=1
    return pf_2, dprofit_2, strategy_2

def func(p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t, x1=0, x2=0, x3=0, x4=0, benchmark=0,
    scheme=1, pf_1=0, x3_p=0):
    def all_zero(*args):
        return all(arg == 0 for arg in args)
    # print(f'决策: {x1} {x2} {x3} {x4}')
```

```
    if scheme==2:
        x1 = 1
        x2 = 1
    q1 = pass_rate(p1, x1)
    q2 = pass_rate(p2, x2)
    pf = total_reject_rate(q1, q2, p3)
    # print(f'q1:{q1} q2:{q2} pf:{1-pf}')
    total_pass_rate = pass_rate(pf, x3)
    bm_pass = (1-p1)*(1-p2)*(1-p3)
    prime_cost = strategy_cost(d1, x1, d2, x2, d3, r, total_pass_rate, x3, x4)
    if all_zero(x1,x2,x3,x4):
        benchmark = total_pass_rate
        # profit = total_pass_rate*s-(1-q)*r-w1-w2-d3 # 000的预期收益
    if scheme==1:
        profit = (1-pf)*s-x3*d3-(1-x3)*pf*r-(w1+w2+w3)-prime_cost
        # profit = (total_pass_rate-bm_pass)*(s+r)-prime_cost
        # print(f'x1 x2 x3 x4={x1} {x2} {x3} {x4}')
        # print(f'total_pass_rate:{round(total_pass_rate, 3)} bm_pass:{round(bm_pass, 3)}
            prime_cost:{prime_cost} profit:{round(profit, 3)}')
        # print(round(profit, 3))

        return pf, total_pass_rate, profit, prime_cost
    elif scheme==2:
        p = pf_1 + (1-pf_1)*total_pass_rate # 两轮内合格，pf_1是第一次的通过率
        # profit = (p-benchmark)*s-(1-p)*r-prime_cost
        # profit = (p-benchmark)*(s+r)-prime_cost-x3_p*d3
        # print(x3_p)-x3_p*d3
        profit = (pf_1-benchmark)*(s+r)+x4*(1-pf_1)*p*(s+r-6)-prime_cost-w1-w2
        return p, total_pass_rate, profit

def scheme_1(situa, strategies, benchmark):
    strategy_0 = [0,0,0,0]
    for i in range(len(situa)):
        # if i!=0:
        #     break
        comp = []
        comp_dr = []
        print(f'第{i+1}种情形：',end='')
        # print(f'第{i+1}种情形：')
        # a, bm, profit, b = func(*(situa[i]+strategy_0))
        for j in strategies:
            # pf, drate, dprofit, cost = func(*(situa[i]+j),benchmark=benchmark)
            pf, drate, dprofit, cost = func(*(situa[i]+j))
            # dprofit += benchmark[i]
            comp.append(round(dprofit, 3))
            comp_dr.append(round(drate, 3))
            ## print(f'合格率：{drate:.3f}，相对收益：{dprofit:.3f}')
```

```
        max_index = comp.index(max(comp))

        # expect = situa[i][9]-situa[i][8]-situa[i][1]-situa[i][4]
        print(strategies[max_index],end='')
        print(round(max(comp), 3))
        # print(comp_dr[max_index])
    return strategies[max_index]

def scheme_2(situa, strategies, benchmark):
    strategy_01 = [0,0,0,0]
    strategy_02 = [0,0,0,1]
    for i in range(6):
        # if i != 0:
        #    continue
        comp = []
        strategies = [list(map(int, bin(i)[2:].zfill(3))) for i in range(8)] #
            生成所有可能的四位二进制组合
        print(f'第{i+1}种情形: ',end='')
        a, benchmark_1, profit_1, c = func(*(situa[i]+strategy_01))
        # benchmark_1 = a
        # pf_benchmark, benchmark_2, profit_2, c = func(*(situa[i]+strategy_02))
        # benchmark_2, a, b = func(*(situa[i]+[0,0,0]), scheme=2, pf_1=pf_benchmark)
        for j in range(len(strategies)): # 每种决策
            strategy_2 = strategies[j].copy()
            # 得到第一轮的后验次品率drate
            pf_1, drate, dprofit_1, c =
                func(*(situa[i]+strategies[j]+[0]),benchmark=benchmark_1, scheme=1)
            # 判断是否需要进行拆解
            # p1_2, p2_2后验次品率, w1_2, w2_2真成本
            p1_2, p2_2, w1_2, w2_2, x4 = if_disassemble(situa[i][0], situa[i][3], drate,
                situa[i][1], situa[i][4], situa[i][11], strategies[j][0], strategies[j][1])
            # 将结果x4加入决策
            strategies[j].append(x4)
            if x4 == 0:
                # max_index = comp.index(max(comp))
                # print(strategies[max_index],end='')
                stra = scheme_1([situa[i]], strategies, benchmark)
                stra.append(x4)
                print(stra, end='')
                print("   不进行拆解")
                break
            strategy_2[0], strategy_2[1] = 1-strategies[j][0], 1-strategies[j][1]
            situa[i][0], situa[i][3] = p1_2, p2_2
            # situa[i][1], situa[i][4] = w1_2, w2_2

            pf_2, dprofit_2, strategy_2[2] = round2_x3(situa[i], strategy_2[:1], benchmark_1,
                drate, strategies[j])
```

```
            dprofit_2 += benchmark[i]
            strategies[j] += strategy_2
            comp.append(round(dprofit_2, 3))

            # print(strategies[j],end=' ')
            # print(round(dprofit_2, 3))

            # print(strategies[j],end=' ')
            # print(round(dprofit_1, 3),end=' ')
            # print(round(dprofit_2, 3))
            # print(round(dprofit_1 + dprofit_2, 3))

        # print(comp)
        # print(strategies)
        if len(comp) > 0:
            max_index = comp.index(max(comp))
            print(strategies[max_index],end=' ')
            print(max(comp))

        # if i==4:
        #     # 第五种情况[0, 1, 1, 1, 1, 0, 1]
        #     pf_1, benchmark_1, profit_1, c = func(*(situa[4]+strategy_01))
        #     pf_21, drate, dprofit_2 = func(*(situa[4]+[1,0]+[1]),benchmark=benchmark_2,
            scheme=2, pf_1=pf_1, x3_p=1) # 收益
        #     print(f'第五种情况: {dprofit_2}')

def scheme_3():
    pass


situa = [[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]]

strategies = [list(map(int, bin(i)[2:].zfill(3))) for i in range(8)] #
    生成所有可能的四位二进制组合


benchmark = []
for i in situa:
    bm = round(cal_benchmark(*i),3)
    benchmark.append(bm)

print(benchmark)
```

```
scheme_2(situa, strategies, benchmark)
# scheme_1(situa, strategies, benchmark)
```

附录 D　问题 2 的方案 3 贝尔曼方程（2 个零配件）

```
import numpy as np
import mdptoolbox
from Problem2_Bellman import value_iteration, policy_evaluation

def fill_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if np.all(row == 0):
                matrix[num][index][index] = 1
    return matrix

def get_P(num_states, num_actions, p1, p2, p3, p1_h, p2_h):
    P = np.zeros((num_actions, num_states, num_states))

    '''买1'''
    P[0][0][1] = P[0][2][4] = P[0][5][8] = P[0][13][15] = 1

    '''买2'''
    P[1][0][2] = P[1][1][4] = P[1][3][6] = P[1][12][14] = 1

    '''检1'''
    P[2][1][0] = P[2][4][2] = P[2][8][5] = P[2][15][13] = p1
    P[2][1][3] = P[2][4][6] = P[2][8][7] = P[2][15][9] = 1-p1
    P[2][10][13] = P[2][11][5] = P[2][12][0] = P[2][14][2] = p1_h
    P[2][10][9] = P[2][11][7] = P[2][12][3] = P[2][14][6] = 1-p1_h

    '''检2'''
    P[3][2][0] = P[3][4][1] = P[3][6][3] = P[3][14][12] = p2
    P[3][2][5] = P[3][4][8] = P[3][6][7] = P[3][14][11] = 1 - p2
    P[3][9][3] = P[3][10][12] = P[3][13][0] = P[3][15][1] = p2_h
    P[3][9][7] = P[3][10][11] = P[3][13][5] = P[3][15][8] = 1-p2_h

    q1 = 1 - p1
    q2 = 1 - p2
    q3 = 1 - p3
    q1_h = 1 - p1_h
    q2_h = 1 - p2_h
```

```
    '''成检'''
    P[4][4][16] = q1*q2*q3
    P[4][4][17] = 1 - q1*q2*q3
    P[4][6][16] = q2 * q3
    P[4][6][17] = 1 - q2*q3
    P[4][7][16] = q3
    P[4][7][17] = 1-q3
    P[4][8][16] = q1*q3
    P[4][8][17] = 1 - q1*q3
    P[4][9][16] = q2_h*q3
    P[4][9][17] = 1 - q2_h*q3
    P[4][10][16] = q1_h*q2_h*q3
    P[4][10][17] = 1 - q1_h*q2_h*q3
    P[4][11][16] = q1_h*q3
    P[4][11][17] = 1 - q1_h*q3
    P[4][14][16] = q1_h*q2*q3
    P[4][14][17] = 1 - q1_h*q2*q3
    P[4][15][16] = q1*q2_h*q3
    P[4][15][17] = 1 - q1*q2_h*q3

    '''拆'''
    P[5][17][10] = 1

    '''不拆'''
    P[6][17][0] = 1

    '''合格'''
    P[7][16][18] = 1

    P = fill_P(P)
    return P

def get_c(d3, w3, r, p):
    return min(d3, r * p) + w3

def get_R(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t, p1_h, p2_h):
    R = np.zeros((num_states, num_actions))

    p1 = 1-p1
    p2 = 1-p2
    p3 = 1-p3
    p1_h = 1-p1_h
    p2_h = 1-p2_h

    R[0][0] = w1
    R[0][1] = w2
```

```
R[1][1] = w2
R[1][2] = d1

R[2][0] = w1
R[2][3] = d2

R[3][1] = w2

R[4][2] = d1
R[4][3] = d2
R[4][4] = get_c(d3, w3, r, 1-p1*p2*p3)
R[5][0] = w1

R[6][3] = d2
R[6][4] = get_c(d3, w3, r, 1-p2*p3)

R[7][4] = get_c(d3, w3, r, 1-p3)

R[8][2] = d1
R[8][4] = get_c(d3, w3, r, 1-p1*p3)

R[9][3] = d2
R[9][4] = get_c(d3, w3, r, 1-p2_h*p3)

R[10][2] = d1
R[10][3] = d2
R[10][4] = get_c(d3, w3, r, 1-p1_h*p2_h*p3)

R[11][2] = d1
R[11][4] = get_c(d3, w3, r, 1-p1_h*p3)

R[12][1] = w2
R[12][2] = d1

R[13][0] = w1
R[13][3] = d2

R[14][2] = d1
R[14][3] = d2
R[14][4] = get_c(d3, w3, r, 1 - p1_h*p2*p3)

R[15][2] = d1
R[15][3] = d2
R[15][4] = get_c(d3, w3, r, 1 - p1*p2_h*p3)

R[16][7] = -s
```

```
    R[17][5] = t


    return R

def check_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if sum(row) != 1:
                print(num, index)

def bellman(num_states, num_actions, params:list):
    p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t = params


    p1_h = p1 / (1 - (1-p1) * (1-p2) * (1-p3))
    p2_h = p2 / (1 - (1-p1) * (1-p2) * (1-p3))


    P = get_P(num_states, num_actions, p1, p2, p3, p1_h, p2_h)
    R = get_R(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t, p1_h,
        p2_h)
    # print(P.shape, R.shape)

    # print(policy_evaluation(num_states, num_actions, P.transpose((1, 0, 2)), -R, 0.99, [0,
        1, 0, 1, 4, 0, 4, 4, 4, 4, 4, 4, 1, 0, 4, 4, 7, 5, 0]))

      # check_P(P)
    # vi = mdptoolbox.mdp.ValueIteration(P, -R, 1)
    # vi.run()

    # optimal_policy = vi.policy
    # print("最优策略:", optimal_policy)

    # # 获取每个状态的预期收益（值函数）
    # state_values = vi.V
    # print("每个状态的预期收益:", state_values)

    V, best_policy = value_iteration(num_states, num_actions, P.transpose((1, 0, 2)), -R,
        gamma=0.99)

    print("状态价值函数：", V)
    print("最优策略：", best_policy)
    return best_policy
```

```
def solve_c(param, best_policy):
    strategy = []
    p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t = param
    p1_h = p1 / (1 - (1-p1) * (1-p2) * (1-p3))
    p2_h = p2 / (1 - (1-p1) * (1-p2) * (1-p3))

    # x1 = best_policy[1] == 2
    # x2 = best_policy[2] == 3
    x1 = 2 in best_policy[[1,4,8]]
    x2 = 3 in best_policy[[2,4,6]]
    if x1 and x2:
        p = p3
    elif x1 and not x2:
        p = 1 - (1-p2)*(1-p3)
    elif x2 and not x1:
        p = 1 - (1-p1)*(1-p3)
    else:
        p = 1 - (1-p1)*(1-p2)*(1-p3)

    x3 = r * p > d3
    x4 = best_policy[-2] == 5
    # x5 = best_policy[11] == 2
    # x6 = best_policy[9] == 3
    x5 = 2 in best_policy[[10,11,12,14]]
    x6 = 3 in best_policy[[9,10,13,15]]

    if x5 and x6:
        p = p3
    elif x5 and not x6:
        p = 1 - (1-p2_h)*(1-p3)
    elif x5 and not x6:
        p = 1 - (1-p1_h)*(1-p3)
    else:
        p = 1 - (1-p1_h)*(1-p2_h)*(1-p3)

    x7 = r * p > d3
    x8 = x4

    if x1:
        # print('先验 检测1')
        strategy.append(1)
    else:
        # print('先验 不检测1')
        strategy.append(0)
    if x2:
        # print('先验 检测2')
        strategy.append(1)
```

```
        else:
            # print('先验 不检测2')
            strategy.append(0)
        if x3:
            # print('先验 成品检测')
            strategy.append(1)
        else:
            # print('先验 不成品检测')
            strategy.append(0)
        if x4:
            # print('先验 拆')
            strategy.append(1)
        else:
            # print('先验 不拆')
            strategy.append(0)
        if x5:
            # print('后验 检测1')
            strategy.append(1)
        else:
            # print('后验 不检测1')
            strategy.append(0)
        if x6:
            # print('后验 检测2')
            strategy.append(1)
        else:
            # print('后验 不检测2')
            strategy.append(0)
        if x7:
            # print('后验 成品检测')
            strategy.append(1)
        else:
            # print('后验 不成品检测')
            strategy.append(0)
        if x8:
            # print('后验 拆')
            strategy.append(1)
        else:
            # print('后验 不拆')
            strategy.append(0)
        print(strategy)

    if __name__=='__main__':
        params = [[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]]
    for param in params:
        best_policy = bellman(19, 8, param)
        solve_c(param, best_policy)

    # param_h = [0.1,8,1,0.1,12,2,0.1,8,4,42,500,6]
    # best_policy = bellman(19, 8, param_h)
    # solve_c(param_h, best_policy)
```

附录 E　问题 3 贝尔曼方程（3 个零配件）

```
import numpy as np
import mdptoolbox
from Problem2_Bellman import *

def fill_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if np.all(row == 0):
                matrix[num][index][index] = 1
    return matrix

def get_P(num_states, num_actions, p1, p2, p3, pc, p1_h, p2_h, p3_h):
    P = np.zeros((num_actions, num_states, num_states))

    '''买1'''
    for i in range(16):
        P[0][i][i+16] = 1

    '''买2'''
    for i in range(4):
        P[1][i][i+4] = 1
    for i in range(16, 20):
        P[1][i][i+4] = 1
    for i in range(32, 36):
        P[1][i][i+4] = 1
    for i in range(48, 52):
        P[1][i][i+4] = 1

    '''买3'''
    for i in range(16):
        P[2][i*4][i*4+1] = 1
```

```
    '''检1'''
    for i in range(16, 32):
        P[3][i][i-16] = p1
        P[3][i+32][i-16] = p1_h
        P[3][i][i+16] = 1-p1
        P[3][i+32][i+16] = 1-p1_h

    '''检2'''
    for i in range(4, 8):
        P[4][i][i-4] = p2
        P[4][i+8][i-4] = p2_h
        P[4][i][i+4] = 1 - p2
        P[4][i+8][i+4] = 1 - p2_h
    for i in range(20, 24):
        P[4][i][i-20] = p2
        P[4][i+8][i-20] = p2_h
        P[4][i][i+4] = 1 - p2
        P[4][i+8][i+4] = 1 - p2_h
    for i in range(36, 40):
        P[4][i][i-36] = p2
        P[4][i+8][i-36] = p2_h
        P[4][i][i+4] = 1 - p2
        P[4][i+8][i+4] = 1 - p2_h
    for i in range(52, 56):
        P[4][i][i-52] = p2
        P[4][i+8][i-52] = p2_h
        P[4][i][i+4] = 1 - p2
        P[4][i+8][i+4] = 1 - p2_h

    '''检3'''
    for i in range(16):
        P[5][i*4+1][i*4] = p3
        P[5][i*4+3][i*4] = p3_h
        P[5][i*4+1][i*4+2] = 1 - p3
        P[5][i*4+3][i*4+2] = 1 - p3_h

    q1 = 1 - p1
    q2 = 1 - p2
    q3 = 1 - p3
    qc = 1 - pc
    q1_h = 1 - p1_h
    q2_h = 1 - p2_h
    q3_h = 1 - p3_h

    '''成检'''
    P[6][21][64] = q1*q2*q3*qc
    P[6][21][65] = 1 - q1*q2*q3*qc
```

```
P[6][22][64] = q1*q2*qc
P[6][22][65] = 1 - q1*q2*qc
P[6][23][64] = q1*q2*q3_h*qc
P[6][23][65] = 1 - q1*q2*q3_h*qc

P[6][25][64] = q1*q3*qc
P[6][25][65] = 1 - q1*q3*qc
P[6][26][64] = q1*qc
P[6][26][65] = 1 - q1*qc
P[6][27][64] = q1*q3_h*qc
P[6][27][65] = 1 - q1*q3_h*qc

P[6][29][64] = q1*q2_h*q3*qc
P[6][29][65] = 1 - q1*q2_h*q3*qc
P[6][30][64] = q1*q2_h*qc
P[6][30][65] = 1 - q1*q2_h*qc
P[6][31][64] = q1*q2_h*q3_h*qc
P[6][31][65] = 1 - q1*q2_h*q3_h*qc
# ------------
P[6][37][64] = q2*q3*qc
P[6][37][65] = 1 - q2*q3*qc
P[6][38][64] = q2*qc
P[6][38][65] = 1 - q2*qc
P[6][39][64] = q2*q3_h*qc
P[6][39][65] = 1 - q2*q3_h*qc

P[6][41][64] = q3*qc
P[6][41][65] = 1 - q3*qc
P[6][42][64] = qc
P[6][42][65] = 1 - qc
P[6][43][64] = q3_h*qc
P[6][43][65] = 1 - q3_h*qc

P[6][45][64] = q2_h*q3*qc
P[6][45][65] = 1 - q2_h*q3*qc
P[6][46][64] = q2_h*qc
P[6][46][65] = 1 - q2_h*qc
P[6][47][64] = q2_h*q3_h*qc
P[6][47][65] = 1 - q2_h*q3_h*qc
# ------------
P[6][53][64] = q1_h*q2*q3*qc
P[6][53][65] = 1 - q1_h*q2*q3*qc
P[6][54][64] = q1_h*q2*qc
P[6][54][65] = 1 - q1_h*q2*qc
P[6][55][64] = q1_h*q2*q3_h*qc
P[6][55][65] = 1 - q1_h*q2*q3_h*qc
```

```
    P[6][57][64] = q1_h*q3*qc
    P[6][57][65] = 1 - q1_h*q3*qc
    P[6][58][64] = q1_h*qc
    P[6][58][65] = 1 - q1_h*qc
    P[6][59][64] = q1_h*q3_h*qc
    P[6][59][65] = 1 - q1_h*q3_h*qc

    P[6][61][64] = q1_h*q2_h*q3*qc
    P[6][61][65] = 1 - q1_h*q2_h*q3*qc
    P[6][62][64] = q1_h*q2_h*qc
    P[6][62][65] = 1 - q1_h*q2_h*qc
    P[6][63][64] = q1_h*q2_h*q3_h*qc
    P[6][63][65] = 1 - q1_h*q2_h*q3_h*qc

    '''拆'''
    P[7][65][63] = 1

    '''不拆'''
    P[8][65][0] = 1

    '''结算'''
    P[9][64][66] = 1

    P = fill_P(P)
    return P

def get_c(dc, wc, r, p):
    return min(dc, r * p) + wc

def get_R(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r, t,
     p1_h, p2_h, p3_h):
    R = np.zeros((num_states, num_actions))

    q1 = 1-p1
    q2 = 1-p2
    q3 = 1-p3
    qc = 1-pc
    q1_h = 1-p1_h
    q2_h = 1-p2_h
    q3_h = 1-p3_h

    for i in range(16):
        R[i][0] = w1

    for i in range(4):
        R[16*i][1] = w2
        R[16*i+1][1] = w2
```

```
        R[16*i+2][1] = w2
        R[16*i+3][1] = w2

    for i in range(16):
        R[4*i][2] = w3

    for i in range(16):
        R[16+i][3] = d1
        R[48+i][3] = d1

    for i in range(8):
        R[8*i+4][4] = d2
        R[8*i+5][4] = d2
        R[8*i+6][4] = d2
        R[8*i+7][4] = d2

    for i in range(32):
        R[2*i+1][5] = d3

    # for i in range(21, 64):
    #     if i % 4 == 0:
    #         continue
    #     R[i][6] = get_c(dc, wc, r, 1-p1*p2*p3*pc)
    R[21][6] = get_c(dc, wc, r, 1 - q1*q2*q3*qc)
    R[22][6] = get_c(dc, wc, r, 1 - q1*q2*qc)
    R[23][6] = get_c(dc, wc, r, 1 - q1*q2*q3_h*qc)
    R[25][6] = get_c(dc, wc, r, 1 - q1*q3*qc)
    R[26][6] = get_c(dc, wc, r, 1 - q1*qc)
    R[27][6] = get_c(dc, wc, r, 1 - q1*q3_h*qc)
    R[29][6] = get_c(dc, wc, r, 1 - q1*q2_h*q3*qc)
    R[30][6] = get_c(dc, wc, r, 1 - q1*q2_h*qc)
    R[31][6] = get_c(dc, wc, r, 1 - q1*q2_h*q3_h*qc)
    R[37][6] = get_c(dc, wc, r, 1 - q2*q3*qc)
    R[38][6] = get_c(dc, wc, r, 1 - q2*qc)
    R[39][6] = get_c(dc, wc, r, 1 - q2*q3_h*qc)
    R[41][6] = get_c(dc, wc, r, 1 - q3*qc)
    R[42][6] = get_c(dc, wc, r, 1 - qc)
    R[43][6] = get_c(dc, wc, r, 1 - q3_h*qc)
    R[45][6] = get_c(dc, wc, r, 1 - q2_h*q3*qc)
    R[46][6] = get_c(dc, wc, r, 1 - q2_h*qc)
    R[47][6] = get_c(dc, wc, r, 1 - q2_h*q3_h*qc)
    R[53][6] = get_c(dc, wc, r, 1 - q1_h*q2*q3*qc)
    R[54][6] = get_c(dc, wc, r, 1 - q1_h*q2*qc)
    R[55][6] = get_c(dc, wc, r, 1 - q1_h*q2*q3_h*qc)
    R[57][6] = get_c(dc, wc, r, 1 - q1_h*q3*qc)
    R[58][6] = get_c(dc, wc, r, 1 - q1_h*qc)
    R[59][6] = get_c(dc, wc, r, 1 - q1_h*q3_h*qc)
```

```
    R[61][6] = get_c(dc, wc, r, 1 - q1_h*q2_h*q3*qc)
    R[62][6] = get_c(dc, wc, r, 1 - q1_h*q2_h*qc)
    R[63][6] = get_c(dc, wc, r, 1 - q1_h*q2_h*q3_h*qc)
    R[64][9] = -s
    R[65][7] = t

    return R

def check_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if sum(row) != 1:
                print(num, index)

def bellman(num_states, num_actions, params:list):
    p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r, t = params

    p1_h = p1 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))
    p2_h = p2 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))
    p3_h = p3 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))

    P = get_P(num_states, num_actions, p1, p2, p3, pc, p1_h, p2_h, p3_h)
    R = get_R(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r,
        t, p1_h, p2_h, p3_h)

    x = [0] * 3
    x[0] = 1 if p1==0 else 0
    x[1] = 1 if p2==0 else 0
    x[2] = 1 if p3==0 else 0
    # print(P.shape, R.shape)

    # print(policy_evaluation(num_states, num_actions, P.transpose((1, 0, 2)), -R, 0.99, [0,
        1, 0, 1, 4, 0, 4, 4, 4, 4, 4, 4, 1, 0, 4, 4, 7, 5, 0]))

    # vi = mdptoolbox.mdp.ValueIteration(P, -R, 0.99)
    # vi.run()
    # optimal_policy = vi.policy
    # print("最优策略:", optimal_policy)
    # # 获取每个状态的预期收益（值函数）
    # state_values = vi.V
    # print("每个状态的预期收益:", state_values)
    # return optimal_policy

    # --------
    # V, best_policy = value_iteration(num_states, num_actions, P.transpose((1, 0, 2)), -R,
```

```
        gamma=0.99)
    V, best_policy = value_iteration_1(num_states, num_actions, x, P.transpose((1, 0, 2)),
        -R, gamma=0.99)
    print("状态价值函数: ", V)
    print("最优策略: ", best_policy)
    return best_policy

def solve_c(param, best_policy):
    strategy_0 = []
    p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r, t = param
    p1_h = p1 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))
    p2_h = p2 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))
    p3_h = p3 / (1 - (1-p1) * (1-p2) * (1-p3) * (1-pc))
    print(f'次品率: {1 - (1-p1) * (1-p2) * (1-p3) * (1-pc)}')
    x1 = 3 in (best_policy[i]for i in [x for x in range(16,32)])
    x2 = 4 in (best_policy[i]for i in [4, 5, 6, 7, 20, 21, 22, 23, 36, 37, 38, 39, 52, 53,
        54, 55])
    x3 = 5 in (best_policy[i]for i in [x for x in range(1, 64, 4)])
    # x4 = True # 是否检测成品

    x6 = 3 in (best_policy[i]for i in [x for x in range(48,64)])
    x7 = 4 in (best_policy[i]for i in [12, 13, 14, 15, 28, 29, 30, 31, 44, 45, 46, 47, 60,
        61, 62, 63])
    x8 = 5 in (best_policy[i]for i in [x for x in range(3, 64, 4)])

    p1_r = 1 if x1 else p1
    p2_r = 1 if x2 else p2
    p3_r = 1 if x3 else p3


    p6_r = 1 if x6 else p1_h
    p7_r = 1 if x7 else p2_h
    p8_r = 1 if x8 else p3_h


    p = 1 - (1-p1_r)*(1-p2_r)*(1-p3_r)*(1-pc)

    x4 = r * p > dc
    x5 = True # 拆
    p = 1 - (1-p6_r)*(1-p7_r)*(1-p3_r)*(1-pc)

    x9 = r * p > dc
    x10 = x5

    strategy_0.append(x1)
    strategy_0.append(x2)
    strategy_0.append(x3)
```

```
    strategy_0.append(x4)
    strategy_0.append(x5)
    strategy_0.append(x6)
    strategy_0.append(x7)
    strategy_0.append(x8)
    strategy_0.append(x9)
    strategy_0.append(x10)
    strategy = [1 if x else 0 for x in strategy_0]

    print(strategy)

if __name__=='__main__':

    # params = [[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]]
    # for param in params:
    #    best_policy = bellman(67, 10, param)
    #    solve_c(param, best_policy)

    # p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r, t = params
    param1 = [0.1, 2, 1, 0.1, 8, 1, 0.1, 12, 2, 0.1, 8, 4, 46, 500, 6]
    param2 = [0, 46, 4, 0, 46, 4, 0, 42, 4, 0.1, 8, 6, 200, 40, 10]
    best_policy = bellman(67, 10, param2)
    solve_c(param2, best_policy)
```

附录 F　问题 4 重新完成问题 2

```
import numpy as np
from scipy.stats import beta
from bellman import *


def get_ps(n, p, size):
    k = int(p * n) # 观测到的次品数
    alpha_prior = 1
    beta_prior = 1

    alpha_post = alpha_prior + k # 后验分布的alpha参数
    beta_post = beta_prior + n - k # 后验分布的beta参数

    posterior_samples = beta.rvs(alpha_post, beta_post, size=size)
```

```
    # print(posterior_samples)
    return posterior_samples

def value_iterations(num_states, num_actions, P, R, gamma, theta=1e-5):
    V = np.zeros(num_states) # 初始化价值函数 V(s)

    while True:
        delta = 0
        for s in range(num_states):
            v = V[s]
            Q_s = np.zeros(num_actions)
            for a in range(num_actions):
                for i in range(0, P.shape[0]):
                    Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] +
                        gamma * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
            V[s] = max(Q_s)
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    # 策略提取
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        Q_s = np.zeros(num_actions)
        for a in range(num_actions):
            for i in range(0, P.shape[0]):
                Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] + gamma
                    * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
        policy[s] = np.argmax(Q_s)

    return V, policy

def fill_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if np.all(row == 0):
                matrix[num][index][index] = 1
    return matrix


def get_PS(num_states, num_actions, p1, p2, p3):
    n = len(p1)
```

```
    Ps = np.zeros((n, num_actions, num_states, num_states))
    for i in range(0, n):
        p1_h = p1[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]))
        p2_h = p2[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]))
        P = get_P(num_states, num_actions, p1[i], p2[i], p3[i], p1_h, p2_h)
        Ps[i] = P
    return Ps

def get_RS(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t):
    n = len(p1)
    Rs = np.zeros((n, num_states, num_actions))
    for i in range(0, n):
        p1_h = p1[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]))
        p2_h = p2[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]))
        R = get_R(num_states, num_actions, p1[i], w1, d1, p2[i], w2, d2, p3[i], w3, d3, s,
            r, t, p1_h, p2_h)
        Rs[i] = R
    return Rs


def bellmans(num_states, num_actions, params:list, n, size):
    p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t = params
    p1 = get_ps(n, p1, size)
    p2 = get_ps(n, p2, size)
    p3 = get_ps(n, p3, size)

    P = get_PS(num_states, num_actions, p1, p2, p3)
    R = get_RS(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, s, r, t)

    V, best_policy = value_iterations(num_states, num_actions, P, -R, gamma=0.99)

    print("状态价值函数: ", V)
    print("最优策略: ", best_policy)
    return best_policy

if __name__ == '__main__':
    params = [[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]]
    # for param in params:
    #     # best_policy = bellmans(19, 8, param, 1475, 10)
    #     best_policy = bellmans(19, 8, param, 3458, 100)
    #     solve_c(param, best_policy)
    #     # bellman(19, 8, param) # 3458 2436
```

```
    param_h = [0.1,8,1,0.1,12,2,0.1,8,4,42,500,6]
    best_policy = bellmans(19, 8, param_h, 3458, 100)
    solve_c(param_h, best_policy)
```

附录 G　问题 4 重新完成问题 3

```
import numpy as np
from scipy.stats import beta
from bellman_3 import *


def get_ps(n, p, size):
    k = int(p * n) # 观测到的次品数
    alpha_prior = 1
    beta_prior = 1

    alpha_post = alpha_prior + k # 后验分布的alpha参数
    beta_post = beta_prior + n - k # 后验分布的beta参数

    posterior_samples = beta.rvs(alpha_post, beta_post, size=size)
    # print(posterior_samples)
    return posterior_samples

def value_iterations(num_states, num_actions, P, R, gamma, theta=1e-5):
    V = np.zeros(num_states) # 初始化价值函数 V(s)

    while True:
        delta = 0
        for s in range(num_states):
            v = V[s]
            Q_s = np.zeros(num_actions)
            for a in range(num_actions):
                for i in range(0, P.shape[0]):
                    Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] +
                        gamma * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
            V[s] = max(Q_s)
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    # 策略提取
    policy = np.zeros(num_states, dtype=int)
```

```
    for s in range(num_states):
        Q_s = np.zeros(num_actions)
        for a in range(num_actions):
            for i in range(0, P.shape[0]):
                Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] + gamma
                    * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
        policy[s] = np.argmax(Q_s)

    return V, policy

def value_iterations_1(num_states, num_actions, P, R, gamma, theta=1e-5):
    V = np.zeros(num_states) # 初始化价值函数 V(s)
    # 先验情况下可以进行检1的状态
    exam_before = [[x for x in range(16, 32)], [4, 5, 6, 7, 20, 21, 22, 23, 36, 37, 38, 39,
        52, 53, 54, 55], [x for x in range(1, 64, 4)]]
    exam_after = [[x for x in range(48, 64)], [12, 13, 14, 15, 28, 29, 30, 31, 44, 45, 46,
        47, 60, 61, 62, 63], [x for x in range(3, 64, 4)]]

    while True:
        delta = 0
        for s in range(num_states):
            v = V[s]
            Q_s = np.zeros(num_actions)
            for a in range(num_actions):
                if a in [0,1,2]:
                    if s in exam_before[a]:
                        continue
                for i in range(0, P.shape[0]):
                    Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] +
                        gamma * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
            V[s] = max(Q_s)
            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break

    # 策略提取
    policy = np.zeros(num_states, dtype=int)
    for s in range(num_states):
        Q_s = np.zeros(num_actions)
        for a in range(num_actions):
            for i in range(0, P.shape[0]):
                Q_s[a] += sum([P[i].transpose((1, 0, 2))[s, a, s_prime] * (R[i][s, a] + gamma
                    * V[s_prime]) for s_prime in range(num_states)])
                Q_s[a] = Q_s[a] / P.shape[0]
```

```
        policy[s] = np.argmax(Q_s)

    return V, policy

def fill_P(matrix):
    for num in range(0, matrix.shape[0]):
        P = matrix[num]
        for index in range(0, P.shape[0]):
            row = P[index]
            if np.all(row == 0):
                matrix[num][index][index] = 1
    return matrix


def get_PS(num_states, num_actions, p1, p2, p3, pc):
    n = len(p1)
    Ps = np.zeros((n, num_actions, num_states, num_states))
    for i in range(0, n):
        p1_h = p1[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        p2_h = p2[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        p3_h = p3[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        P = get_P(num_states, num_actions, p1[i], p2[i], p3[i], pc[i], p1_h, p2_h, p3_h)
        Ps[i] = P
    return Ps

def get_RS(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r,
    t):
    n = len(p1)
    Rs = np.zeros((n, num_states, num_actions))
    for i in range(0, n):
        p1_h = p1[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        p2_h = p2[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        p3_h = p3[i] / (1 - (1-p1[i]) * (1-p2[i]) * (1-p3[i]) * (1-pc[i]))
        R = get_R(num_states, num_actions, p1[i], w1, d1, p2[i], w2, d2, p3[i], w3, d3,
            pc[i], wc, dc, s, r, t, p1_h, p2_h, p3_h)
        Rs[i] = R
    return Rs


def bellmans(num_states, num_actions, params:list, n, size):
    p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s, r, t = params
    p1 = get_ps(n, p1, size)
    p2 = get_ps(n, p2, size)
    p3 = get_ps(n, p3, size)
    pc = get_ps(n, pc, size)

    P = get_PS(num_states, num_actions, p1, p2, p3, pc)
```

```
    R = get_RS(num_states, num_actions, p1, w1, d1, p2, w2, d2, p3, w3, d3, pc, wc, dc, s,
        r, t)

    V, best_policy = value_iterations(num_states, num_actions, P, -R, gamma=0.99)
    V, best_policy = value_iterations_1(num_states, num_actions, P, -R, gamma=0.99)

    print("状态价值函数: ", V)
    print("最优策略: ", best_policy)
    return best_policy

if __name__ == '__main__':
    params = [[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]]
    # for param in params:
    #     # best_policy = bellmans(19, 8, param, 1475, 10)
    #     best_policy = bellmans(19, 8, param, 2436, 10)
    #     bellman(19, 8, param)
    param1 = [0.1, 2, 1, 0.1, 8, 1, 0.1, 12, 2, 0.1, 8, 4, 46, 500, 6]
    param2 = [0, 46, 4, 0, 46, 4, 0, 42, 4, 0.1, 8, 6, 200, 40, 10]
    best_policy = bellmans(67, 10, param2, 3458, 100)
    # bellman(67, 10, param1)
    solve_c(param2, best_policy)
```

附录 H　绘制右尾检验的正态分布图

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

# 定义均值和标准差
mu = 0 # 均值
sigma = 1 # 标准差

# 创建正态分布曲线的 x 值
x = np.linspace(-4, 4, 1000) #用4表示右尾区域
y = norm.pdf(x, mu, sigma)

# 添加字体路径
font_path = "SimHei.ttf" # 替换为实际的字体文件路径
fm.fontManager.addfont(font_path)
```

```
prop = fm.FontProperties(fname=font_path)

plt.rcParams['font.sans-serif'] = [prop.get_name()]
plt.rcParams['axes.unicode_minus'] = False # 正常显示负号

# 创建图形
plt.figure(figsize=(10, 6))
plt.rc('font', family='SimHei')
# 绘制正态分布曲线
plt.plot(x, y, label='标准正态分布', color='blue')

# 设置单侧检验的 Z 值 (95% 单侧检验)
z_value = 1.645
x_fill = np.linspace(z_value, 4, 1000)
y_fill = norm.pdf(x_fill, mu, sigma)

# 填充单侧检验区域
plt.fill_between(x_fill, y_fill, color='orange', alpha=0.5, label=f'右侧检验区域 (Z =
    {z_value})')

# 标注 Z 值
plt.axvline(x=z_value, color='orange', linestyle='--', label=f'Z = {z_value}')

# 设置标题和标签
plt.title('右尾检验的标准正态分布图')
plt.xlabel('Z 值')
plt.ylabel('概率密度')

# 显示图例
plt.legend()

# 显示图形
plt.grid(True)
plt.savefig('figure/a.pdf')
```

附录 I　绘制左尾检验的正态分布图

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

# 定义均值和标准差
mu = 0 # 均值
sigma = 1 # 标准差
```

```
# 创建正态分布曲线的 x 值
x = np.linspace(-4, 4, 1000) #用4表示右尾区域
y = norm.pdf(x, mu, sigma)

# 添加字体路径
font_path = "SimHei.ttf" # 替换为实际的字体文件路径
fm.fontManager.addfont(font_path)
prop = fm.FontProperties(fname=font_path)

plt.rcParams['font.sans-serif'] = [prop.get_name()]
plt.rcParams['axes.unicode_minus'] = False # 正常显示负号

# 创建图形
plt.figure(figsize=(10, 6))
plt.rc('font', family='SimHei')
# 绘制正态分布曲线
plt.plot(x, y, label='标准正态分布', color='blue')

# 设置单侧检验的 Z 值 (90% 单侧检验)
z_value = 1.28
x_fill = np.linspace(-4, z_value, 1000)
y_fill = norm.pdf(x_fill, mu, sigma)

# 填充单侧检验区域
plt.fill_between(x_fill, y_fill, color='orange', alpha=0.5, label=f'左侧检验区域 (Z =
    {z_value})')

# 标注 Z 值
plt.axvline(x=z_value, color='orange', linestyle='--', label=f'Z = {z_value}')

# 设置标题和标签
plt.title('左尾检验的标准正态分布图')
plt.xlabel('Z 值')
plt.ylabel('概率密度')

# 显示图例
plt.legend()

# 显示图形
plt.grid(True)
plt.savefig('figure/b.pdf')
```

附录 J　绘制不同估计误差 E 在不同信度下的抽样次数

```python
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.stats import norm
import matplotlib.font_manager as fm

def calculate_size(E, Z, p):
    """
    计算单侧检验所需的样本量 n

    参数:
    E -- 允许误差
    Z -- 单侧置信度对应的 Z 值 (如 1.645 对应 95% 单侧置信度)
    p -- 标称值

    返回:
    所需的样本量 n (向上取整)
    """
    n = (pow(Z,2) * p * (1 - p)) / pow(E,2)
    return np.ceil(n) # 取整

Z1 = 1.645 # 对应95%单侧置信度的Z值(上侧单侧区间, 右尾检验)
Z2 = 1.28 # 对应90%单侧置信度的Z值 (下侧单侧区间, 左尾检验)
p = 0.1 # 给出的标称值为10%

# 生成 允许估计误差E 值的数组
E = np.linspace(0.001, 0.2, 400)

y1 = calculate_size(E, Z1, p)
y2 = calculate_size(E, Z2, p)

# 添加字体路径
font_path = "2024B/SimHei.ttf" # 替换为实际的字体文件路径
fm.fontManager.addfont(font_path)
prop = fm.FontProperties(fname=font_path)

plt.rcParams['font.sans-serif'] = [prop.get_name()]
plt.rcParams['axes.unicode_minus'] = False # 正常显示负号

# 绘制函数图
plt.figure(figsize=(8, 6)) # 设置图像大小
plt.plot(E, y1) # 绘制函数图
plt.xlabel('E') # x轴标签
plt.ylabel('n') # y轴标签
plt.title('不同估计误差 E 在信度为95%情况下抽样次数') # 图像标题
plt.grid(True) # 显示网格
```

```
# 保存图像到文件
plt.savefig('2024B/figure/sen_95.pdf') # 可以保存为.png, .pdf, .svg等格式

# 绘制函数图
plt.figure(figsize=(8, 6)) # 设置图像大小
plt.plot(E, y2) # 绘制函数图
plt.xlabel('E') # x轴标签
plt.ylabel('n') # y轴标签
plt.title('不同估计误差 E 在信度为90%情况下抽样次数') # 图像标题
plt.grid(True) # 显示网格

# 保存图像到文件
plt.savefig('2024B/figure/sen_90.pdf') # 可以保存为.png, .pdf, .svg等格式
```

附录 K　绘制不同信度下的次品率后验分布

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

# 观测数据
n = 3458 # 样本总数
k = 345.8 # 次品数

# 先验分布参数 (无信息先验 Beta(1, 1))
alpha_prior = 1
beta_prior = 1

# 计算后验分布参数
alpha_post = alpha_prior + k
beta_post = beta_prior + n - k

# 生成后验分布的样本
x = np.linspace(0, 1, 1000) # 次品率的可能取值范围 (0到1之间)
posterior_pdf = beta.pdf(x, alpha_post, beta_post)

# 绘制次品率的后验分布
plt.plot(x, posterior_pdf, label=f'Beta({alpha_post}, {beta_post})', color='blue')
plt.fill_between(x, posterior_pdf, alpha=0.3, color='blue')

# 输出后验分布的均值和 95% 置信区间
mean_posterior = beta.mean(alpha_post, beta_post)
ci_lower, ci_upper = beta.ppf([0.025, 0.975], alpha_post, beta_post)
```

```
# 在图中绘制均值和置信区间
plt.axvline(mean_posterior, color='red', linestyle='--', label=f'均值:
    {mean_posterior:.4f}')
plt.axvline(ci_lower, color='green', linestyle='--', label=f'95% 置信区间下限:
    {ci_lower:.4f}')
plt.axvline(ci_upper, color='green', linestyle='--', label=f'95% 置信区间上限:
    {ci_upper:.4f}')

# 添加字体路径
font_path = "SimHei.ttf" # 替换为实际的字体文件路径
fm.fontManager.addfont(font_path)
prop = fm.FontProperties(fname=font_path)

plt.rcParams['font.sans-serif'] = [prop.get_name()]
plt.rcParams['axes.unicode_minus'] = False # 正常显示负号

# 设置图的标题和标签
plt.title('次品率的后验分布')
plt.xlabel('次品率 (p)')
plt.ylabel('概率密度')

# 显示图例
plt.legend()
plt.grid(True)
plt.savefig('figure/c.pdf')

# 打印均值和 95% 置信区间
print(f"后验分布的均值为: {mean_posterior:.4f}")
print(f"95% 置信区间为: [{ci_lower:.4f}, {ci_upper:.4f}]")
```