

**软件可靠性课程实验报告**

**题 目:** JM模型和GO模型及比较

**院 系:** 计算机科学与技术学院

**专 业:** 软件工程

**学生姓名:**

**学 号:**

**二零二四 年 十一月 二十三日**

目录

[1.编写目的 3](#_Toc185777268)

[2.算法原理 5](#_Toc185777269)

[2.1 JM模型假设和推导 5](#_Toc185777270)

[2.2 GO模型假设和推导 6](#_Toc185777271)

[3.算法实现 7](#_Toc185777272)

[3.1 算法流程 7](#_Toc185777273)

[3.2 算法实现 9](#_Toc185777274)

[3.3 数据来源 20](#_Toc185777275)

[3.4 结果展示 20](#_Toc185777276)

[4.算法结果分析 23](#_Toc185777277)

# 1.编写目的

1）JM模型

JM 模型，全称 Jelinski-Moranda 模型，是软件可靠性工程领域中备受瞩目的经典模型之一。它诞生于对软件系统故障规律探索的需求之下，旨在为软件开发者、测试人员以及维护团队提供一种科学、量化的方法，用以深入理解软件故障行为并有效管理软件可靠性。

在软件的开发与运行过程中，故障的出现是不可避免的，但准确把握故障的发生规律对于保障软件质量至关重要。JM 模型的核心关注点在于剖析软件系统中故障数量随时间的演变以及每个故障对整体故障率的影响。

其编写目的主要体现在以下几个方面：

首先，通过对软件运行过程中记录的故障时间数据进行分析，JM 模型试图估算出软件在初始状态下所包含的错误数量。这一信息对于软件开发者来说极具价值，它能够帮助他们在开发初期对软件的潜在风险有一个较为清晰的认识，从而在后续的开发流程中有针对性地加强代码审查、测试等环节，以减少初始错误的引入。

其次，JM 模型能够根据故障发生的时间间隔等数据，计算出每个故障所对应的故障率贡献比例。这使得测试人员可以依据这些结果，优先关注那些对故障率影响较大的故障区域，合理分配测试资源，提高测试效率，确保在软件发布前尽可能多地发现并修复关键故障。

再者，在软件维护阶段，JM 模型可用于预测在未来一段时间内软件系统可能出现的故障数量及频率。基于这些预测，维护团队能够提前规划维护工作，准备所需的资源，如人力、物力等，及时对软件进行修复和优化，保障软件系统的持续稳定运行，降低因故障而导致的业务损失风险。

2）GO模型

GO 模型，即 Goel-Okumoto 模型，同样是软件可靠性评估领域的重要工具，它以独特的视角和方法来描述软件故障的增长趋势，为软件可靠性管理提供了另一种有效的途径。

随着软件系统在各个领域的广泛应用，对其可靠性的要求也日益提高。GO 模型应运而生，旨在通过对软件实际运行过程中故障发生时间的观测和分析，构建一个能够准确反映软件累积故障数随时间变化规律的模型。

具体而言，GO 模型的编写目的包括以下几点：

其一，它着重于刻画软件系统中累积故障数的动态增长过程。通过对大量故障数据的拟合分析，GO 模型能够确定出与实际情况最为匹配的参数，进而生成一个精确的函数表达式来描述累积故障数与时间的关系。这使得软件项目团队能够直观地了解到软件故障随着时间推移是如何逐步累积的，为评估软件的可靠性状态提供了一个清晰的量化依据。

其二，基于所建立的累积故障数与时间的函数关系，GO 模型可以用于预测在未来特定时间段内软件系统可能出现的累积故障数量。这种预测能力对于软件运营方和维护团队来说意义重大。他们可以根据预测结果提前制定应对策略，比如安排系统升级、加强监控措施或者调整业务流程等，以确保软件在面对可能出现的故障增长时能够保持正常运行，满足用户的需求。

其三，GO 模型在评估软件可靠性时，更侧重于从整体的故障增长趋势角度出发，相对弱化了对单个故障具体影响的考量。这使得它在处理一些复杂的软件系统，尤其是那些故障发生机制较为复杂、难以精确界定单个故障影响的情况下，能够提供一种较为简洁、有效的可靠性评估方法，帮助团队快速把握软件的整体可靠性态势。

3）对JM模型和GO模型比较

对 JM 模型和 GO 模型进行全面深入的比较，具有多方面的重要意义，其目的不仅仅在于区分两者的差异，更在于挖掘它们在不同应用场景下的优势与局限性，从而为软件可靠性评估和管理提供更为精准、有效的指导。

从理论层面来看，比较两种模型的目的在于：

首先，深入剖析它们各自的理论基础、模型假设以及计算公式。JM 模型基于每个故障对整体故障率有明确贡献的假设，通过一系列复杂的数学推导得出其相关公式；而 GO 模型则是从累积故障数的增长模式出发，构建了与之对应的函数关系。通过对比，可以清晰地看到两种模型在描述软件故障行为时所依据的不同逻辑和侧重点，这有助于我们更深入地理解软件故障的本质以及不同视角下的量化描述方式。

其次，通过比较两种模型的理论特性，我们能够发现它们在处理不同类型软件故障数据时可能存在的优势与劣势。例如，JM 模型在处理那些能够较为清晰地界定单个故障影响的数据时，可能会表现出较高的准确性；而 GO 模型在面对大量故障数据且更关注整体故障增长趋势的情况下，或许能更有效地捕捉到数据中的规律。这种理论层面的比较为我们在实际应用中根据具体数据特点选择合适的模型奠定了基础。

从实际应用角度出发，比较 JM 模型和 GO 模型的目的还包括：

其一，根据具体的软件项目特点、可获取的数据类型和精度等因素，为选择更为合适的模型进行软件可靠性评估提供依据。不同的软件项目具有各自独特的故障发生模式和数据收集情况。比如，对于一些小型、功能相对单一且故障发生与单个模块紧密相关的软件项目，JM 模型可能因其对单个故障分析的精细程度而更具优势；而对于大型、复杂的软件系统，其故障数据往往呈现出复杂的增长趋势且难以精确区分单个故障影响，此时 GO 模型可能更能满足对整体可靠性评估的需求。通过对两种模型的比较，我们可以根据实际情况做出更为明智的模型选择决策。

其二，通过比较两种模型在实际应用中的表现，我们可以发现它们可能存在的不足和局限性。例如，JM 模型在处理复杂的软件故障数据时，可能会因为其对单个故障假设的严格性而导致模型拟合效果不佳；GO 模型虽然在捕捉整体故障增长趋势方面表现出色，但可能在对单个故障的细节分析上有所欠缺。识别这些不足为进一步改进和优化模型提供了方向，有助于推动软件可靠性理论和实践的不断发展。

其三，比较两种模型还能够为综合运用它们提供思路。在某些情况下，我们可以考虑将 JM 模型和 GO 模型结合使用，充分发挥它们各自的优势。比如，利用 JM 模型对软件系统中关键故障区域进行精细分析，再结合 GO 模型对整体故障增长趋势的把握，从而实现对软件可靠性更为全面、准确的评估。这种综合运用的方式能够在一定程度上弥补单一模型的局限性，为软件可靠性管理提供更为完善的解决方案。

综上所述，对 JM 模型和 GO 模型进行比较的目的是多方面的，涵盖了从理论理解到实际应用的各个环节，旨在通过深入剖析两者的特点、优势和局限性，为软件可靠性评估和管理提供更为科学、有效的方法和策略。

# 2.算法原理

## 2.1 JM模型假设和推导

1. 软件中的固有错误数是一个未知的常数
2. 软件中的各个错误是相互独立的，每个错误导致系统发生失效的可能性大致相同，各次失效间隔时间也是相互独立的
3. 测试过程中检测到的错误，都被排除，每次排错只排除一个错误，排错时间可以忽略不计，在排错过程中不引入新的错误
4. 软件的失效率在每个失效间隔时间内是常数，其数值正比于程序中残留的错误数，比例系数为:
5. 软件的运行方式与预期的运行方式相似

JM模型给出的失效率函数为：其中和是未知参数，也就是我们要进行估计的参数，以第次失效为起点的第次失效发生的时间是一个随机变量，它服从以为参数的指数分布，其概率密度函数为

我们用最大似然法对上述两个未知参数进行估计，最终得到的估计值是从下列两个方程中求解的：

由上述两个方程我们得知在编程实现中，我们就需要计算如下两个值：

同时这里设为对自变量的误差精度控制值，为对自变量的误差精度控制值。

接下来根据如下算法步骤进行求解和：

步骤1：如果，则，转步骤2；如果，则中止计算；

步骤2：如果，则，重复步骤2；若，则，转步骤5；若，转步骤3；

步骤3：如果，则，转步骤5；如果，则，转步骤4；

步骤4：如果，则，转步骤3；如果，则转步骤5；如果，则，转步骤3；

步骤5：，终止计算。

## 2.2 GO模型假设和推导

1. 程序在同实际执行环境相差不大的条件下执行
2. 在软件测试过程中，所有检测到的故障是相互独立的
3. 测试未运行时的软件失效为0；当测试进行时，软件失效服从均值为的非齐次泊松过程（NHPP）
4. t 时刻累积检测到的故障数量变化率与当前剩余的故障数量成正比例, 比例系数为
5. 每次只修正一个错误，当软件故障出现时，引发故障的错误被立即排除，并不会引入新的错误

GO 模型在测试区内的累计失效期望函数为：，为软件累计测量时间；可靠性函数为

按假设，若时刻累计故障数为，则得到概率密度为:

;

则的联合概率密度函数为：

则在给定时，关于参数，的似然函数为：

得到以下方程：

所以该方程的解就是参数和的估计值。

# 3.算法实现

## 3.1 算法流程

JM模型算法流程：

首先，输入数据，然后通过计算得到。接着判断是否成立，如果成立，则令且；若不成立，则流程结束。

之后进入一个循环，判断是否成立，若成立，则令且，继续循环判断；若不成立，则进一步判断是否成立。如果成立，则判断是否成立，若成立，则令；若不成立，则同样令，然后继续下一步判断。

当确定后，判断是否成立，若成立，则令；若不成立，则进一步判断是否成立，若成立，则令；若成立，则令，然后通过计算得到，最后输出和。

JM模型算法流程图如图1：

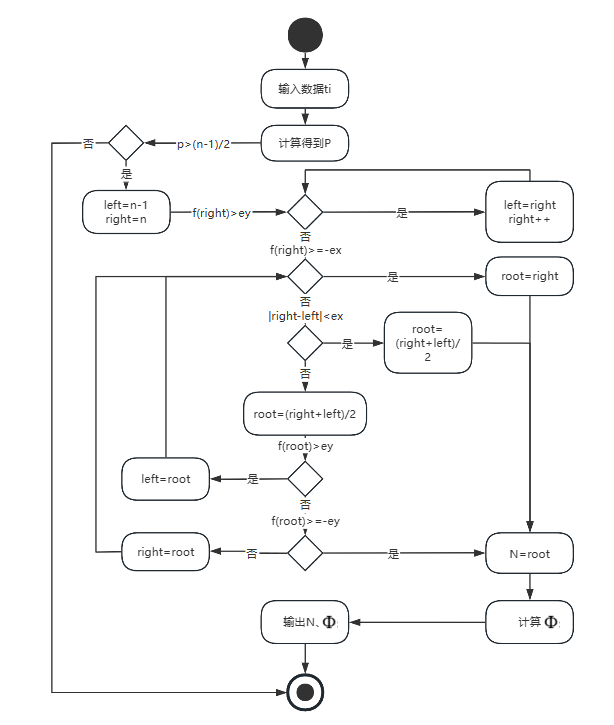


图1 JM模型算法流程图

GO模型算法流程：

首先，输入数据，然后进行计算得到。接着判断的值：如果，则进一步判断是否成立，若成立，则计算；若不成立，则进入计算的模块，在该模块中判断是否成立，若成立，则令，若不成立，则令。如果，则计算和，接着计算，之后同样判断是否成立，若成立，则计算；若不成立，则进入计算的模块进行上述相同操作。

GO模型算法流程图如图2：

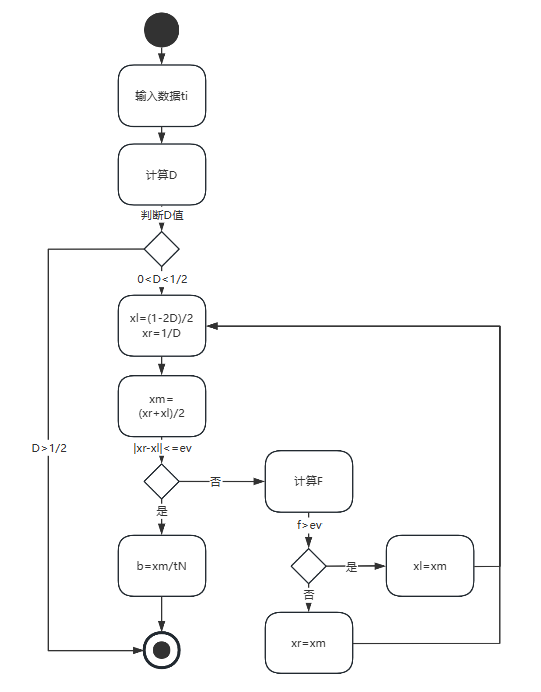


图2 GO模型算法流程图

## 3.2 算法实现

本次实验采用python语言编写，运行的IDE是VScode。

|  |
| --- |
| JM模型代码如下 |
| from typing import List  import matplotlib.pyplot as plt  class JMModel:      def \_\_init\_\_(self):          """          初始化JMModel类的实例。          该方法设置了类的初始状态，包括时间列表t、参数ex和ey、根值root、初始故障数N0和故障强度Fi。          """          self.t = []  # 存储故障时间的列表，初始为空列表          self.ex = None  # 参数ex，初始值为None          self.ey = None  # 参数ey，初始值为None          self.root = 0  # 用于计算的根值，初始值为0          self.N0 = 0  # 初始故障数，初始值为0          self.Fi = 0  # 故障强度，初始值为0      def getP(self, lst: List[float]) -> float:          """          计算故障时间的累积故障数。          :param lst: 故障时间列表。          :return: 累积故障数。          """          result = 0          n = len(lst)          for i in range(1, n):              t1 = lst[i]              t2 = lst[i - 1]              result += (i - 1) \* (t1 - t2)          tn = lst[-1]          result /= tn          return result      def function(self, lst: List[float], N: float) -> float:          """          计算给定N值下的函数值。          :param lst: 故障时间列表。          :param N: 待计算的N值。          :return: 函数值。          """          result = 0          n = len(lst)          P = self.getP(lst)          for i in range(1, n):              result += 1 / (N - i + 1)          result -= (n - 1) / (N - P)          return result      def read\_data(self):          """          从文件中读取故障数据。          该方法从名为"test.txt"的文件中读取故障数据，并将故障时间存储在t列表中。          """          with open("test.txt", 'r') as input\_file:              for line in input\_file:                  failure\_Number, Timetofailure = map(int, line.strip().split())                  self.t.append(float(Timetofailure))      def set\_ex\_ey(self):          """          设置ex和ey的值。          该方法通过用户输入获取ex和ey的值，并将其存储在类的属性中。          """          self.ex = float(input("请输入ex的值："))          self.ey = float(input("请输入ey的值："))      def calculate(self):          """          计算N0和Fi的值。          该方法根据读取的故障数据和设置的ex、ey值，计算N0和Fi的值。          """          n = len(self.t)          P = self.getP(self.t)          # print(f"当ex={self.ex},ey={self.ey}时")          # 以下是根据P值与((n - 1) / 2)的比较结果来确定后续计算的初始边界值left和right          if P > ((n - 1) / 2):              left = n - 1              right = n          else:              return  # 如果P不满足大于((n - 1) / 2)的条件，则直接返回，不进行后续复杂计算。          # 以下循环用于通过不断调整right的值，使得function(self.t, right) <= self.ey          while self.function(self.t, right) > self.ey:              left = right              right += 1          # 经过上述循环后，根据function(self.t, right)与self.ex的比较结果，确定self.root的值          if self.function(self.t, right) >= self.ex:              self.root = right          else:              while True:                  # 以下是一个二分查找类似的循环，用于更精确地确定self.root的值，使得function(self.t, self.root)在一定误差范围内满足条件                  if abs(right - left) < self.ex:                      self.root = (right + left) / 2                      break                  self.root = (right + left) / 2                  if self.function(self.t, self.root) > self.ey:                      left = self.root                  else:                      if self.function(self.t, self.root) < (-self.ey):                          right = self.root                      else:                          break          # 至此，已经确定了self.root的值，接下来根据已确定的值以及之前读取的故障时间数据来计算N0和Fi          self.N0 = self.root          sum\_val = 0          tn = self.t[-1]          for i in range(1, n):              t1 = self.t[i]              t2 = self.t[i - 1]              sum\_val += (i - 1) \* (t1 - t2)          self.Fi = (n - 1) / (self.N0 \* tn - sum\_val)      def print\_results(self):          """          打印计算结果。          该方法打印出ex、ey、N0和Fi的值。          """          print(f"当ex={self.ex},ey={self.ey}时")          print(f"N0={self.N0}")          print(f"Fi={self.Fi}")      def calculate\_MTBF(self):          """          计算JM模型的平均故障间隔时间（MTBF）。          :return: 平均故障间隔时间（MTBF）。          """          MTBF = 1 / self.Fi          # print(f"MTBF: {MTBF}")          return MTBF      def calculate\_failure\_rate\_mean(self):          """          计算JM模型的失效率均值。          :return: 失效率均值。          """          n = len(self.t)          failure\_rate\_sum = 0          for i in range(1, n):              t1 = self.t[i]              t2 = self.t[i - 1]              failure\_rate\_sum += self.Fi \* (n - (i - 1)) \* (t1 - t2)          failure\_rate\_mean = failure\_rate\_sum / (self.t[-1])          # print(f"失效率均值: {failure\_rate\_mean}")          return failure\_rate\_mean |

|  |
| --- |
| GO模型代码如下 |
| from typing import List  import math  import matplotlib.pyplot as plt  class GOModel:      def \_\_init\_\_(self):          """          初始化GOModel类的实例。          该方法设置了类的初始状态，包括时间列表t、精度参数epslv、参数a和b。          """          self.t = []  # 存储故障时间的列表，初始为空列表          self.epslv = None  # 精度参数，初始值为None          self.a = 0  # 参数a，初始值为0          self.b = 0  # 参数b，初始值为0      def Dfunction(self, lst: List[float]) -> float:          """          计算故障时间的累积故障数。          :param lst: 故障时间列表。          :return: 累积故障数。          """          result = 0          n = len(lst)          for i in range(1, n):              t1 = lst[i]              result += t1          tn = lst[-1]          result /= tn          result /= n          return result      def read\_data(self):          """          从文件中读取故障数据。          该方法从名为"test.txt"的文件中读取故障数据，并将故障时间存储在t列表中。          """          try:              with open("test.txt", 'r') as input\_file:                  for line in input\_file:                      failure\_Number, Timetofailure = map(int, line.strip().split())                      self.t.append(float(Timetofailure))          except FileNotFoundError:              print("文件未找到，请检查文件路径是否正确。")      def set\_epslv(self):          """          设置精度参数epslv的值。          该方法通过用户输入获取epslv的值，并将其存储在类的属性中。          """          self.epslv = float(input("请输入epslv的值："))      def calculate(self):          """          计算模型的参数a和b。          该方法根据读取的故障数据和设置的精度参数epslv，计算模型的参数a和b。          """          N = len(self.t)          tn = self.t[-1]          xl = 0          xr = 0          xm = 0          D = self.Dfunction(self.t)          f = 0          # 步骤1: 判断D大小          if 0 < D < 0.5:              # 根据模型的要求，首先计算 `D` 值（通过调用 `self.Dfunction` 方法），然后判断 `D` 的范围。              # 如果 `D` 的值在 `0` 和 `0.5` 之间，就按照特定公式设置 `xl` 和 `xr` 的初始值，这两个值将作为后续迭代计算的边界。              # `xl` 和 `xr` 的设置是基于模型的理论和算法要求，为后续的迭代搜索确定一个初始的区间范围。              xl = (1 - 2 \* D) / 2              xr = 1 / D          else:              # 如果 `D` 不在 `0` 到 `0.5` 这个范围内，说明当前的数据情况可能不满足模型的有效计算条件，              # 所以直接返回，不进行后续的复杂计算流程。              return          # 转步骤2：          max\_iterations = 1000  # 设置一个最大迭代次数，防止无限循环          iteration\_count = 0          while True:              # 进入一个循环，这个循环的目的是通过不断迭代来逼近满足特定条件的 `xm` 值。              # 每次迭代都会更新 `xm` 的值，并根据 `xm` 计算相关的函数值 `f`，然后根据 `f` 与 `self.epslv` 的比较结果来调整 `xl` 和 `xr` 的值，              # 从而逐步缩小搜索范围，直到找到满足条件的 `xm` 值或者达到最大迭代次数。              xm = (xl + xr) / 2              if math.fabs(xr - xl) <= self.epslv:                  # 如果当前 `xr` 和 `xl` 的差值的绝对值小于等于用户设置的 `self.epslv` 值，说明已经找到了满足精度要求的 `xm` 值，                  # 此时就可以跳出循环，进入下一步的计算。                  break              else:                  y = math.exp(xm)                  f = (1 - D \* xm) \* y + (D - 1) \* xm - 1                  if f > self.epslv:                      # 如果计算得到的函数值 `f` 大于 `self.epslv`，说明当前的 `xm` 值使得函数值偏大，                      # 所以将 `xl` 更新为当前的 `xm` 值，以便在下一次迭代中缩小搜索范围，让函数值更接近目标范围。                      xl = xm                  elif f < -self.epslv:                      # 如果计算得到的函数值 `f` 小于负的 `self.epslv`，说明当前的 `xm` 值使得函数值偏小，                      # 所以将 `xr` 更新为当前的 `xm` 值，同样是为了在下一次迭代中调整搜索范围，使函数值更符合要求。                      xr = xm                  else:                      # 如果函数值 `f` 在 `[-self.epslv, self.epslv]` 范围内，说明已经找到了满足精度要求的 `xm` 值，                      # 直接跳出循环，不再进行后续的迭代调整。                      break              iteration\_count += 1              if iteration\_count >= max\_iterations:                  # 在每次迭代过程中，会记录迭代的次数。如果迭代次数达到了设置的最大迭代次数 `max\_iterations`（这里设置为1000次），                  # 说明可能由于数据问题或者算法本身的原因，没有在规定的迭代次数内找到满足精度要求的 `xm` 值。                  # 此时会打印提示信息，告知用户可能需要检查数据或算法，然后直接返回，不再继续计算。                  print("达到最大迭代次数，可能未收敛，请检查数据或算法。")                  return          # 步骤4          b = xm / tn          ahelp = math.exp(-b \* tn)          a = N / (1 - ahelp)          # 在通过上述迭代过程找到满足精度要求的 `xm` 值后，就可以根据模型的公式计算出关键参数 `a` 和 `b` 的值。          # 首先计算 `b` 值，它是 `xm` 与最后一个故障时间 `tn` 的比值。然后通过 `b` 值计算出 `ahelp`，再根据 `ahelp` 和数据长度 `N` 计算出 `a` 值。          # 最后将计算得到的 `a` 和 `b` 值分别赋值给 `self.a` 和 `self.b`，以便在类的其他方法中可以使用这些计算出来的关键参数。          self.a = a          self.b = b      def print\_results(self):          """          打印计算结果。          该方法打印出精度参数epslv、参数a和b的值。          """          print(f"当epslv={self.epslv}时，")          print(f"a={self.a}")          print(f"b={self.b}")      def calculate\_MTBF(self):          """          计算GO模型的平均故障间隔时间（MTBF）。          :return: 平均故障间隔时间（MTBF）。          """          MTBF = 1 / self.b          # print(f"MTBF: {MTBF}")          return MTBF      def calculate\_failure\_rate\_mean(self):          """          计算GO模型的失效率均值。          :return: 失效率均值。          """          N = len(self.t)          failure\_rate\_sum = 0          for i in range(0, N):              failure\_rate\_sum += self.a \* self.b \* math.exp(-self.b \* self.t[i])          failure\_rate\_mean = failure\_rate\_sum / N          # print(f"失效率均值: {failure\_rate\_mean}")          return failure\_rate\_mean |

|  |
| --- |
| 模型比较代码如下 |
| from JM import JMModel  from GO import GOModel  import os  if \_\_name\_\_ == "\_\_main\_\_":      os.chdir(os.path.dirname(os.path.abspath(\_\_file\_\_)))      while True:          # 创建JMModel实例          jm\_model = JMModel()          jm\_model.read\_data()          jm\_model.set\_ex\_ey()          jm\_model.calculate()          jm\_model.print\_results()          jm\_model.calculate\_MTBF()          jm\_model.calculate\_failure\_rate\_mean()          # 创建GOModel实例          go\_model = GOModel()          go\_model.read\_data()          go\_model.set\_epslv()          go\_model.calculate()          go\_model.print\_results()          go\_model.calculate\_MTBF()          go\_model.calculate\_failure\_rate\_mean()          # 计算并比较MTBF          jm\_MTBF = jm\_model.calculate\_MTBF()          go\_MTBF = go\_model.calculate\_MTBF()          print(f"JM模型MTBF: {jm\_MTBF}")          print(f"GO模型MTBF: {go\_MTBF}")          if jm\_MTBF > go\_MTBF:              print("在MTBF方面，JM模型表现更好。")          elif jm\_MTBF < go\_MTBF:              print("在MTBF方面，GO模型表现更好。")          else:              print("在MTBF方面，JM模型和GO模型表现相同。")          # 计算并比较失效率均值          jm\_failure\_rate\_mean = jm\_model.calculate\_failure\_rate\_mean()          go\_failure\_rate\_mean = go\_model.calculate\_failure\_rate\_mean()          print(f"JM模型失效率均值: {jm\_failure\_rate\_mean}")          print(f"GO模型失效率均值: {go\_failure\_rate\_mean}")          if jm\_failure\_rate\_mean > go\_failure\_rate\_mean:              print("在失效率均值方面，GO模型表现更好。")          elif jm\_failure\_rate\_mean < go\_failure\_rate\_mean:              print("在失效率均值方面，JM模型表现更好。")          else:              print("在失效率均值方面，JM模型和GO模型表现相同。") |

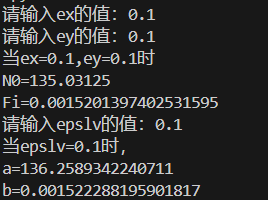
|  |
| --- |
| 模型可视化代码如下 |
| import numpy as np  import matplotlib.pyplot as plt  from JM import JMModel  from GO import GOModel  import os  plt.rcParams["font.sans-serif"] = ["SimHei"]  # 设置字体  plt.rcParams["axes.unicode\_minus"] = False  # 该语句解决图像中的"-"负号的乱码问题  def visualize\_models():      """      可视化模型拟合对比的函数。      该函数初始化了JM模型和GO模型实例，读取数据，设置模型参数，计算相关参数，      并绘制了原始数据、JM模型和GO模型的故障数量随时间的变化曲线。      """      # 初始化JM模型和GO模型实例      jm\_model = JMModel()      go\_model = GOModel()      # 读取数据      jm\_model.read\_data()      go\_model.read\_data()      # 设置JM模型参数      jm\_model.set\_ex\_ey()      # 设置GO模型参数      go\_model.set\_epslv()      # 计算JM模型和GO模型的相关参数      jm\_model.calculate()      go\_model.calculate()      # 时间范围      time\_range = np.linspace(0, 6000, 1000)      # JM模型预测的故障数量      jm\_cumulative\_failures = jm\_model.N0 \* (1 - np.exp(-jm\_model.Fi \* time\_range))      # GO模型预测的故障数量      go\_cumulative\_failures = go\_model.a \* (1 - np.exp(-go\_model.b \* time\_range))      # 实际故障数量      actual\_cumulative\_failures = np.arange(1, len(jm\_model.t) + 1)      # 绘图      plt.figure(figsize=(10, 6))      plt.plot(jm\_model.t, actual\_cumulative\_failures, 'o', label='原始数据', color='black')      plt.plot(time\_range, jm\_cumulative\_failures, '-', label='JM模型', color='blue')      plt.plot(time\_range, go\_cumulative\_failures, '--', label='GO模型', color='red')      # 设置图例和标签      plt.title('模型拟合对比')      plt.xlabel('数据x值')      plt.ylabel('数据y值')      plt.legend()      plt.grid()      plt.show()  # 调用可视化函数  if \_\_name\_\_ == "\_\_main\_\_":      os.chdir(os.path.dirname(os.path.abspath(\_\_file\_\_)))      visualize\_models() |

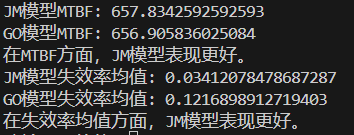
## 3.3 数据来源

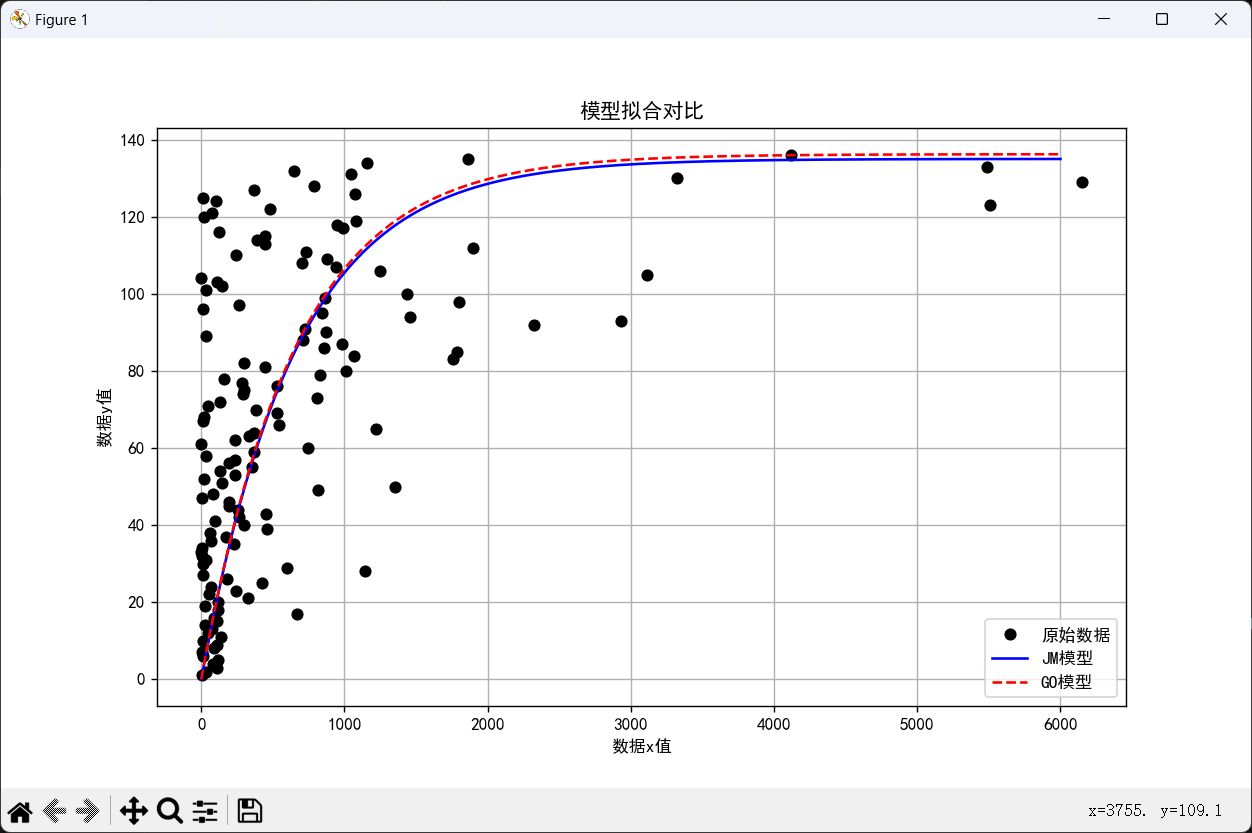
数据来源是自己编写的错误数据test.txt，已放入了工程目录下。

## 3.4 结果展示

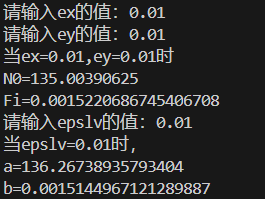
JM模型ex=0.1，ey=0.1，GO模型epslv=0.1，拟合结果如下：

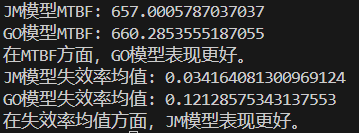


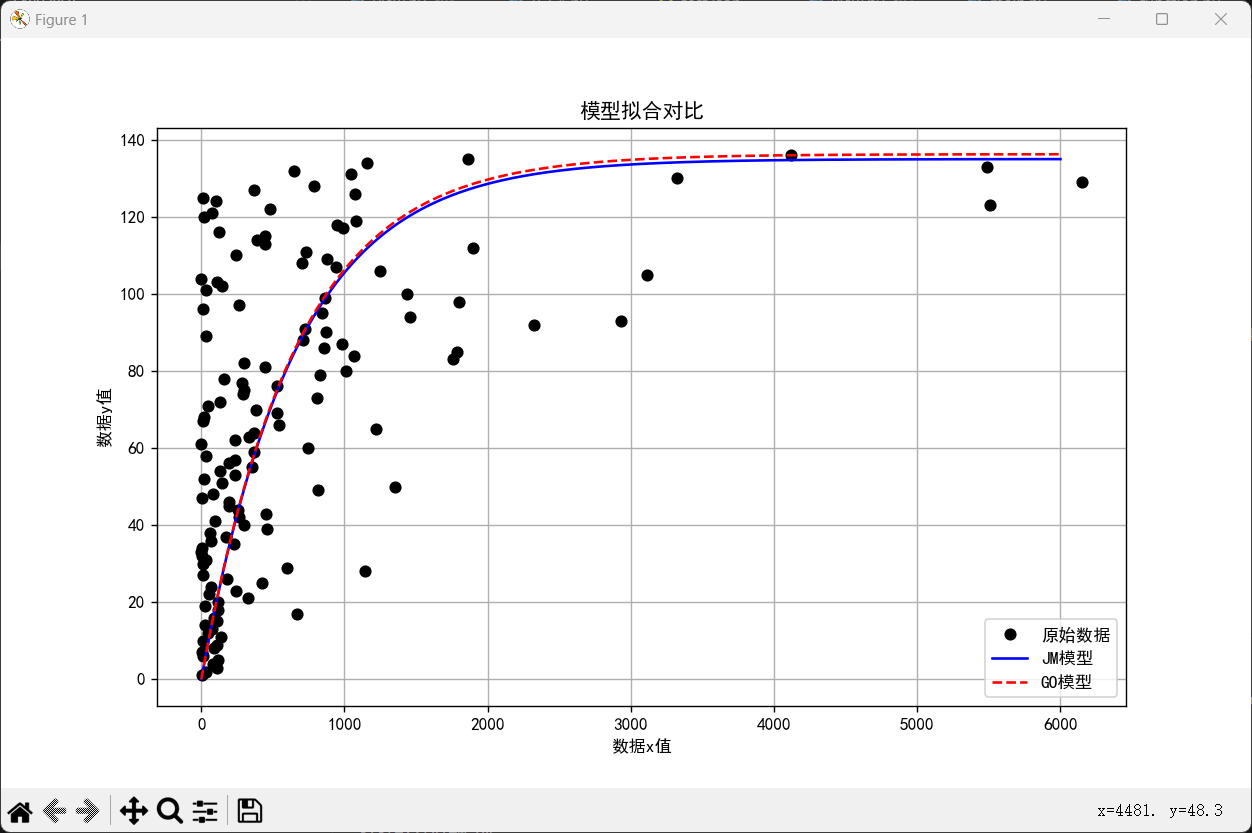




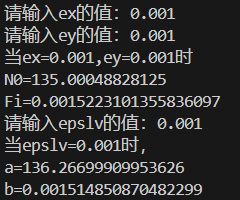
JM模型ex=0.01，ey=0.01，GO模型epslv=0.01，拟合结果如下：

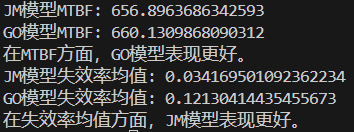


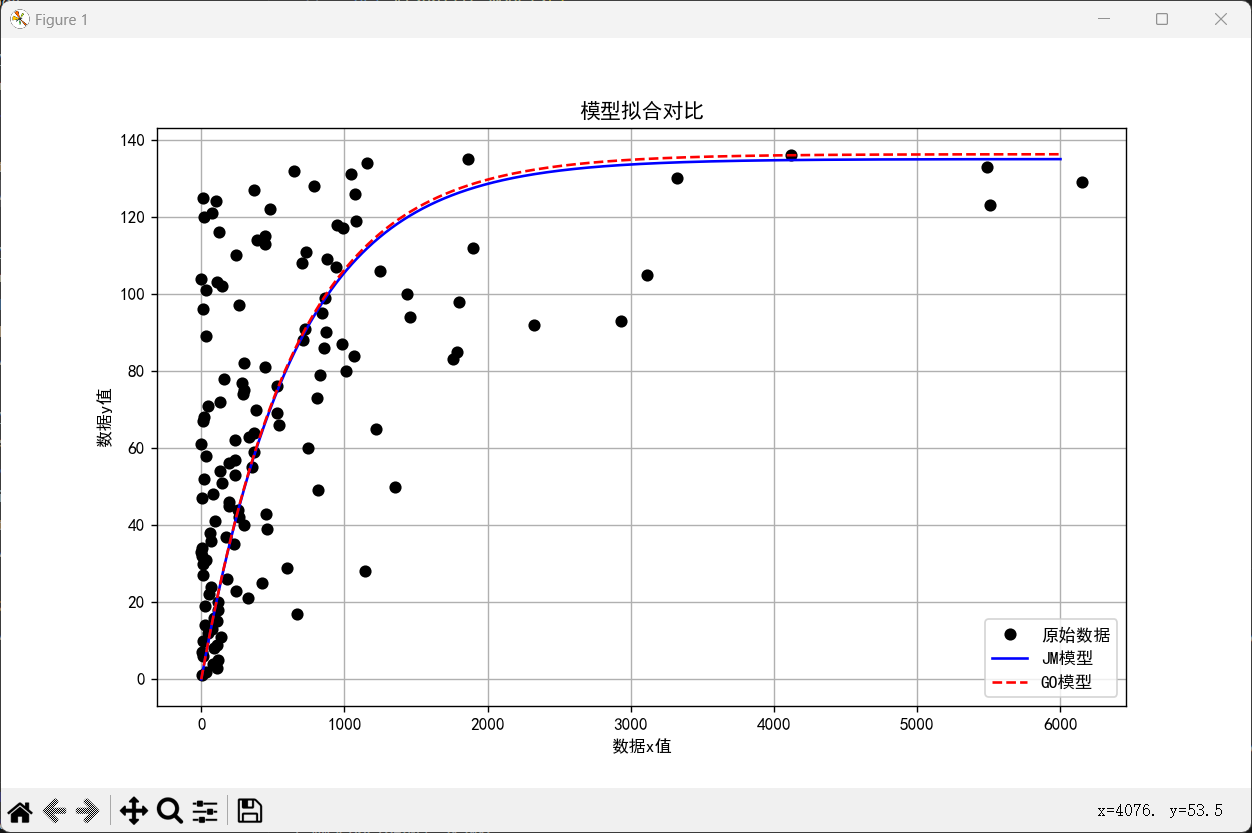




JM模型ex=0.001，ey=0.001，GO模型epslv=0.001，拟合结果如下：







# 4.算法结果分析

我们可以根据JM模型估计出该程序的固有错误数，该程序的失效率函数的比例常数；可以根据GO模型估计出该程序的参数，。

对于同一数据集比较两个算法的优劣：

当JM模型ex=0.1，ey=0.1，GO模型epslv=0.1时，对于MTBF和失效率均值方面，JM模型表现更好；

当JM模型ex=0.01，ey=0.01，GO模型epslv=0.01时，在MTBF方面，GO模型表现更好，在失效率均值方面，JM模型表现更好；

当JM模型ex=0.001，ey=0.001，GO模型epslv=0.001时，在MTBF方面，GO模型表现更好，在失效率均值方面，JM模型表现更好。