

**软件可靠性课程实验报告**

**题 目:** GM模型和EMD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 三十日**

目录

[1.编写目的 3](#_Toc185777510)

[2.算法原理 4](#_Toc185777511)

[2.1 GM模型理论 4](#_Toc185777512)

[2.2 EMD模型原理 5](#_Toc185777513)

[3.算法实现 7](#_Toc185777514)

[3.1 算法流程 7](#_Toc185777515)

[3.2 算法实现 8](#_Toc185777516)

[3.3 数据来源 23](#_Toc185777517)

[3.4 结果展示 23](#_Toc185777518)

[4.算法结果分析 24](#_Toc185777519)

1.编写目的

在现代软件开发和维护过程中，软件的可靠性是衡量其质量的重要指标之一。尤其是在一些对系统稳定性要求极高的领域，如金融、医疗、航天等行业，软件系统的故障往往会带来巨大的损失。因此，如何准确预测软件的故障发生频率、寿命和潜在的风险，成为了软件工程中的一个关键问题。为了更好地解决这一问题，科学家和工程师们提出了多种数学模型来分析软件的可靠性，其中灰色模型（GM模型）和经验模态分解（EMD模型）是常用的两种方法。这两种模型在数据处理和故障预测中各有其独特的优势，尤其适用于面对复杂、不确定和不完整数据时的可靠性分析。

灰色模型（GM模型）是一种适用于少量、不完全或不确定数据的建模方法。它在对数据进行建模时并不要求大量的历史数据，而是通过对现有少量数据的累积生成和差分处理来揭示数据的趋势。这使得GM模型在许多实际应用中具有显著的优势，尤其是在数据有限或者数据采集困难的情况下，能够依然提供合理的预测结果。对于软件可靠性分析，GM模型能够基于历史故障数据或软件缺陷记录，建立一个较为精确的预测模型，进而预测未来系统故障的发生频率和软件的寿命周期。其优点在于无需大量的数据支持，通过对现有数据的高效处理，便可快速得出结论。尤其适用于那些需要快速决策且数据有限的情境。

另一方面，经验模态分解（EMD模型）作为一种非线性、非平稳时间序列数据的分析方法，近年来在软件可靠性分析领域得到了广泛的应用。EMD的核心思想是将一个复杂的时间序列信号分解为多个本征模态函数（IMF）和一个残差项，每个IMF代表信号的局部特征。通过这种方式，EMD能够将原始信号中含有的复杂信息提取出来，从而揭示出信号的内在规律。对于软件故障预测，EMD能够有效地从软件系统的故障数据中提取出不同故障模式的特征，并根据这些特征预测系统的未来故障趋势。由于EMD具有很强的自适应能力，它可以从含有噪声和非线性成分的数据中提取出有用的信息，因此在面对复杂的、噪声较多的故障数据时，EMD能够提供更为精确的分析结果。

尽管GM和EMD各有独特的优势，但在实际应用中，它们也存在一些局限性。GM模型虽然在数据较少的情况下表现出色，但它依赖于数据的平稳性，并且在数据波动较大的情况下，其预测结果可能不够精确。EMD模型则擅长从非平稳和非线性的时间序列数据中提取信息，但它也存在分解过程计算量大的问题，且分解结果的解释较为复杂。因此，在进行软件可靠性分析时，单一模型可能无法完全满足所有需求。为了克服这些局限性，我们可以尝试将这两种模型结合起来，通过比较它们在不同场景下的表现，找出最适合特定软件系统的分析方法。

在本研究中，我们通过对比灰色模型和经验模态分解模型在软件可靠性分析中的应用，旨在为软件故障预测提供更加准确和高效的分析方法。我们首先使用灰色模型对历史数据进行建模，通过累积生成的方式来预测软件系统的未来故障发生频率及寿命。然后，使用EMD模型对数据进行分解，提取不同故障模式的特征，从而进一步分析软件系统的稳定性和潜在风险。通过比较这两种方法的优劣，我们能够发现它们各自的优点，并结合实际应用需求，选择最合适的分析方法。

此外，比较这两种模型在不同类型数据上的表现，能够帮助我们更好地理解它们在实际应用中的适用场景。对于那些数据较为稳定且历史记录较为充分的软件系统，灰色模型可能是一个较好的选择。它能够在数据有限的情况下提供较为精确的预测。而对于那些数据波动较大、含有噪声的软件系统，EMD模型则可以更好地从复杂的故障数据中提取出有用的信息，为软件可靠性分析提供更加细致的分析结果。因此，本研究的最终目标是找到一种适合不同类型软件系统的可靠性分析方法，帮助开发人员更好地进行故障预测和风险管理。

通过对灰色模型和经验模态分解模型的综合比较，我们希望能够为软件开发和维护提供更为可靠的数据支持和决策依据。无论是在软件设计初期，还是在软件的运行和维护阶段，提前预测和识别系统的潜在问题，都是确保软件质量和系统稳定性的重要步骤。最终，结合这两种方法，我们将能够为软件可靠性分析提供更加精确、全面的工具，从而提高软件的可靠性和稳定性，确保其在实际应用中的高效运行。

# 2.算法原理

## 2.1 GM模型理论

灰色系统模型是利用较少的或不确切的表示灰色系统行为特征的原始数据序列作生成变换后建立的，用以描述灰色系统内部事物连续变化过程的模型。灰色模型通过累加操作(AGO)的方法，对无规则的原始数据进行处理，从而弱化原始数据的随机性和波动性，生成有规则的准指数规律的数据，然后对这些生成的数据进行建模、预测。常用的灰色预测模型主要是经典的一阶模型。

模型由一个单变量的一阶微分方程构成:

式中，为待定的参数，为原始数据序列的累加生成值，公式也被称为时间响应函数。设为原始数据的元序列，即，做一次累加后得到一次累加(1-AGO)序列:

其中。

令为的紧邻均值生成序列为:

其中。

若为待定参数列，记

，

在第一个公式中由差分代替徽分，可得模型的最小二乘估计参数列满足:

由以上条件可以得到时间响应函数的解为:

又由，可得

令, 由上式可得到模型的预测值:

得到参数值后，需要对模型进行精度检验。相应的残差序列为:

经过上述生成与建模之后，为分析预测模型的有效性，使用残差检验法进行残差检验，以了解预测值和实际值间之残差

若平均精度大于0.01，则代表此模型的预测效果良好。

## 2.2 EMD模型原理

首先利用经验模态分解的方法提取出能描述软件失效行为的具有不同特征的子序列，然后结合支持向量回归和灰色预测理论对上述子序列进行预测，最后将得到的若干子序列预测结果进行重构得出最终的预测结果。

经验模态分解(Empirical Mode Decomposition,EMD)是黄锷(N.E.Huang)在美国国家宇航局与其他人于1998年创造性地提出的一种新型自适应信号时频处理方法，它的优点是不会运用任何已经定义好的函数作为基底，而是根据所分析的信号而自适应生成固有模态函数。可以用于分析非线性、非平稳的信号序列，具有很高的信噪比和良好的时频聚焦性特别适用于非线性非平稳信号的分析处理。

经验模态分解是将信号分解成一些固有模态函数(Intrinsic Mode Function,IMF)分量,使得各IMF 分量是窄带信号，即IMF分量必须满足下面两个条件:在整个信号长度上，极值点和过零点的数目必须相等或者至多只相差一个;在任意时刻，由极大值点定义的上包络线和由极小值点定义的下包络线的平均值为零，即信号的上下包络线关于时间轴对称。简单的说就是将一个复杂信号分解成多个简单信号的过程。同小波变换相比，EMD方法是完全根据信号数据本身来确定需要分解出多少个IMF，因此更加的具有自适应性。

EMD方法是基于如下假设基础上的:

1. 信号至少有两个极值点，一个极大值和一个极小值;
2. 特征时间尺度通过两个极值点之间的时间定义;
3. 若数据缺乏极值点但有形变点，则可通过数据微分一次或几次获得极值点，然后再通过积分来获得分解结果。

EMD分解的目的是将一个信号分解为个固有模态函数(IMF)和一个残差(residual)。其中，每个IMF需要满足以下两个条件:

1. 在整个数据范围内，局部极值点和过零点的数目必须相等，或者相差数目最多为1。
2. 在任意时刻，局部最大值的包络(上包络线)和局部最小值的包络(下包络线)的平均值必须为零。

EMD分解的实现过程主要包括如下几步:

第一步：寻找信号全部极值点，通过三次样条曲线将局部极大值点连成上包络线，将局部极小值点连成下包络线。上、下包络线包含所有的数据点。

第二步：由上包络和下包络线的平均值，得出：

若h1(t)满足IMF的条件，则可认为是的第一个IMF分量。

第三步：若不符合IMF条件，则将作为原始数据，重复步骤1、步骤2，得到上、下包络的均值，通过计算是否适合IMF分量的必备条件,若不满足，重复如上两步次，直到满足前提下得到。第1个IMF表示如下:

停止条件可以用标准差SD控制(筛分门限值，一般取值，小于门限值时才停止这样得到的第一个满足条件的就是第一个IMF。

标准差SD的求法:

第四步：将从信号中分离得到:

将作为原始信号重复上述三个步骤，循环n次，得到第二个IMF分量直到第个IMF分量, 则会得出:

第五步：当变成单调函数后，剩余的成为残余分量。所有IMF分量和残余分量之和为原始信号:

EMD算法实现描述如图1所示。

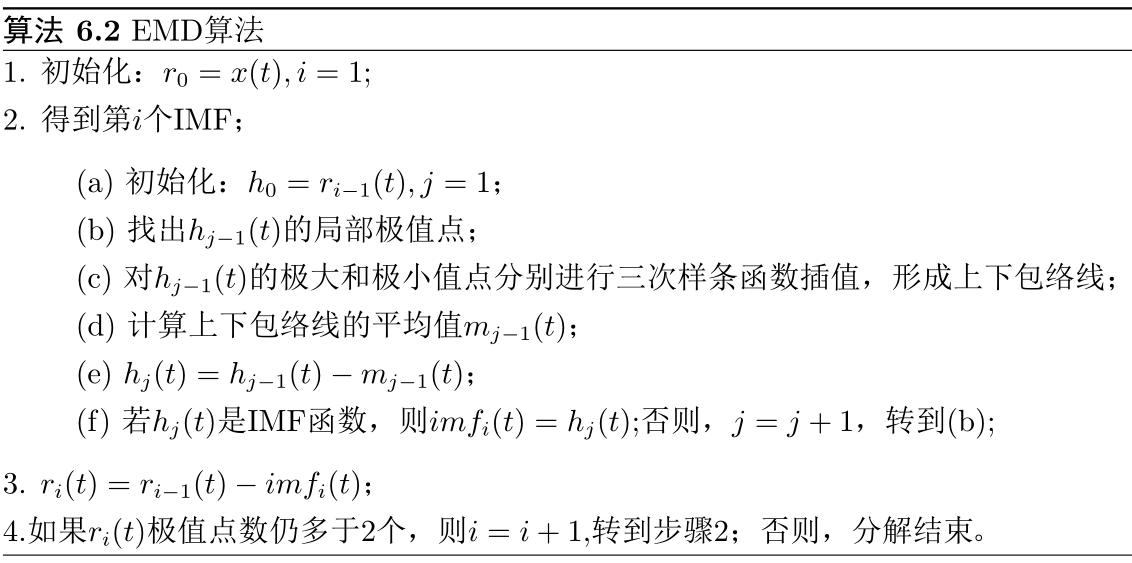


图1 EMD算法实现图

非平稳的失效数据序列经过EMD分解以后，各IMF分量都基本趋于平稳，这对预测是有利的。而且每一IMF分量都是对软件失效样本数据特征的一种真实反应，通过对失效数据序列的各个特征分别进行预测处理，然后再对各预测结果分量进行重构，就可以得到更加理想的预测效果。

# 3.算法实现

## 3.1 算法流程

GM模型流程如下：

首先从 GM (1,1) 开始，进行级比检验。若未通过级比检验，则进行平移变换或者换模型；若通过级比检验，则进行累加数据制造规律。接着构造微分方程，然后利用最小二乘法求参数，求解微分方程得到预测值。之后对预测值进行检验，若未通过检验，则改进模型或者换模型，再次进行检验；若通过检验，则得到可靠的预测值。

GM模型流程图如图2：

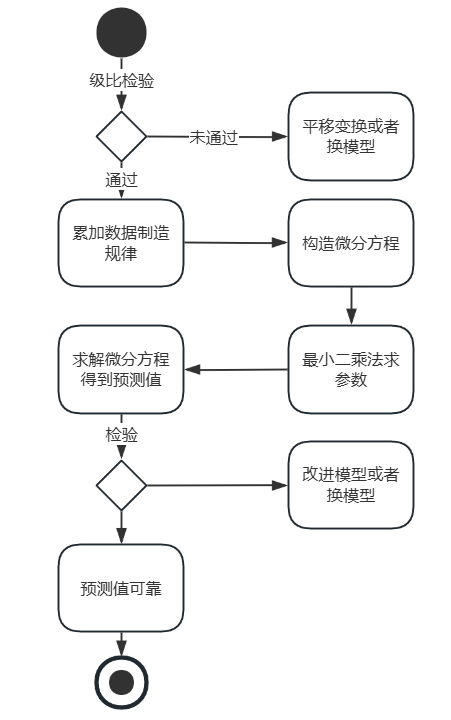


图2 GM模型流程图

EMD模型流程如下：

首先输入侧线原始信号，令且。接着确定的局部极大值点和局部极小值点，然后拟合上包络线和下包络线，计算和。

之后判断是否满足条件，如果不满足，则并返回继续确定的局部极大值点和局部极小值点；如果满足，则令，，，接着判断是否为单调函数，如果不是，则并返回继续确定的局部极大值点和局部极小值点；如果是，则流程结束。

EMD模型流程图如图3：

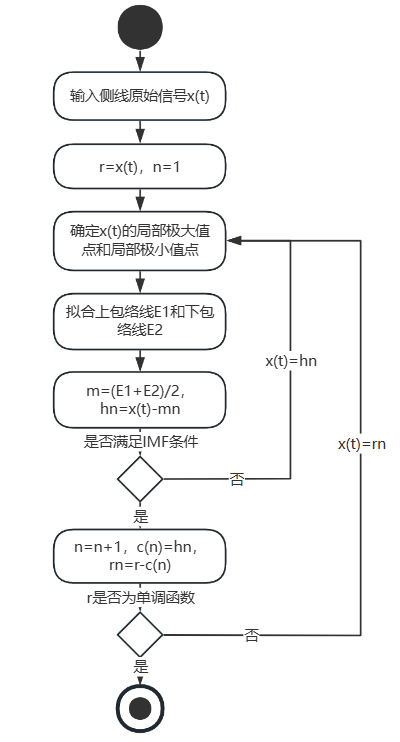


图3 EMD模型流程图

## 3.2 算法实现

本次实验采用python语言编写，运行的IDE是VScode。

|  |
| --- |
| GM模型代码如下 |
| import numpy as np  class GMModel:      def \_\_init\_\_(self):          """          初始化GMModel类的实例。          在这里定义了一系列用于存储模型相关数据和参数的属性，初始化为空的numpy数组或零值。          """          self.test\_data = np.array(())  # 用于存储实验数据集，初始为空数组          self.add\_data = np.array(())  # 存储一次累加产生的数据，初始为空数组          self.argu\_a = 0  # 模型的参数a，初始化为0          self.argu\_b = 0  # 模型的参数b，初始化为0          self.MAT\_B = np.array(())  # 矩阵B，用于后续的计算，初始为空数组          self.MAT\_Y = np.array(())  # 矩阵Y，用于后续的计算，初始为空数组          self.modeling\_result\_arr = np.array(())  # 存储对实验数据的拟合值，初始为空数组          self.P = 0  # 小误差概率，初始化为0          self.C = 0  # 后验方差比值，初始化为0      def set\_model(self, arr:list):          """          设置GM模型的参数并进行相关计算。          该方法接收一个列表作为输入，通常是实验数据的列表形式。          它会依次调用内部的几个私有方法来完成数据处理、参数计算和拟合值的获取。          """          self.\_\_acq\_data(arr)  # 调用私有方法获取数据并构建矩阵B和矩阵Y          self.\_\_compute()  # 调用私有方法计算模型的参数a和b          self.\_\_modeling\_result()  # 调用私有方法获取对实验数据的拟合值      def \_\_acq\_data(self, arr:list):          """          私有方法：获取并处理数据，构建矩阵B和矩阵Y。          该方法将输入的列表数据转换为numpy数组，并进行一次累加操作，然后根据累加后的数据构建矩阵B和矩阵Y。          参数:          arr (list): 实验数据的列表形式。          """          self.test\_data = np.array(arr).flatten()  # 将输入的列表转换为一维的numpy数组，作为实验数据集          add\_data = []          sum\_val = 0          for i in range(len(self.test\_data)):              sum\_val = sum\_val + self.test\_data[i]  # 对实验数据进行累加              add\_data.append(sum\_val)          self.add\_data = np.array(add\_data)  # 将累加后的数据转换为numpy数组          ser = []          for i in range(len(self.add\_data) - 1):              temp = (-1) \* ((1 / 2) \* self.add\_data[i] + (1 / 2) \* self.add\_data[i + 1])              ser.append(temp)          B = np.vstack((np.array(ser).flatten(), np.ones(len(ser), ).flatten()))          self.MAT\_B = np.array(B).T  # 构建矩阵B并转置          Y = np.array(self.test\_data[1:])          self.MAT\_Y = np.reshape(Y, (len(Y), 1))  # 构建矩阵Y并重塑为列向量形式      def \_\_compute(self):          """          私有方法：计算模型的参数a和b。          通过之前构建的矩阵B和矩阵Y，利用矩阵运算来计算出模型的两个重要参数a和b。          """          temp\_1 = np.dot(self.MAT\_B.T, self.MAT\_B)  # 先计算矩阵B的转置与自身的点积          temp\_2 = np.matrix(temp\_1).I  # 对上述结果求逆          temp\_3 = np.dot(np.array(temp\_2), self.MAT\_B.T)  # 再与矩阵B的转置做点积          vec = np.dot(temp\_3, self.MAT\_Y)  # 最后与矩阵Y做点积得到包含参数a和b的向量          self.argu\_a = vec.flatten()[0]  # 从向量中获取参数a的值          self.argu\_b = vec.flatten()[1]  # 从向量中获取参数b的值      def \_\_predict(self, k:int) -> float:          """          私有方法：根据模型参数进行预测计算。          基于已经计算出的参数a和b，以及给定的预测步数k，计算出相应的预测值。          参数:          k (int): 预测的步数。          返回:          float: 预测的值。          """          part\_1 = 1 - np.exp(self.argu\_a)  # 计算预测公式的第一部分          part\_2 = self.test\_data[0] - self.argu\_b / self.argu\_a  # 计算预测公式的第二部分          part\_3 = np.exp((-1) \* self.argu\_a \* k)  # 计算预测公式的第三部分          return part\_1 \* part\_2 \* part\_3  # 返回预测结果      def \_\_modeling\_result(self):          """          私有方法：获取对实验数据的拟合值。          通过调用\_\_predict方法，依次计算出每个实验数据点对应的拟合值，并存储在modeling\_result\_arr中。          """          ls = [self.\_\_predict(i + 1) for i in range(len(self.test\_data) - 1)]          ls.insert(0, self.test\_data[0])          self.modeling\_result\_arr = np.array(ls)  # 将计算出的拟合值转换为numpy数组并存储      def predict(self, number:int) -> list:          """          根据模型进行外部预测。          基于已经训练好的模型（即已经计算出参数a和b），对后续指定个数的数据进行预测。          参数:          number (int): 需要预测的数据个数。          返回:          list: 预测结果的列表形式。          """          prediction = [self.\_\_predict(i + len(self.test\_data)) for i in range(number)]          return prediction      def precision\_evaluation(self):          """          对模型的精度进行评估。          通过计算实验数据与拟合数据之间的误差，进而计算出平均误差、平均实验数据值等，最终得出小误差概率P和后验方差比值C。          """          error = [              self.test\_data[i] - self.modeling\_result\_arr[i]              for i in range(len(self.test\_data))          ]  # 计算每个实验数据点与对应拟合值的误差          aver\_error = np.mean(error)  # 计算平均误差          aver\_test\_data = np.mean(self.test\_data)  # 计算平均实验数据值          temp1 = 0          temp2 = 0          for i in range(len(error)):              temp1 = temp1 + (self.test\_data[i] - aver\_test\_data) \*\* 2              temp2 = temp2 + (error[i] - aver\_error) \*\* 2          square\_S\_1 = temp1 / len(self.test\_data)  # 计算实验数据的方差          square\_S\_2 = temp2 / len(error)  # 计算误差的方差          self.C = np.sqrt(square\_S\_2) / np.sqrt(square\_S\_1)  # 计算后验方差比值          ls = [i                for i in range(len(error))                if np.abs(error[i] - aver\_error) < (0.6745 \* np.sqrt(square\_S\_1))                ]          self.P = len(ls) / len(error)  # 计算小误差概率          # print("精度指标P,C值为：", self.P, self.C)      def get\_predicted\_data(self):          """          获取模型对实验数据的拟合值。          返回:          numpy.ndarray: 模型对实验数据的拟合值数组。          """          return self.modeling\_result\_arr |

|  |
| --- |
| EMD模型代码如下 |
| import numpy as np  from scipy import interpolate  class EMDModel:      def \_\_init\_\_(self, datax, datay):          """          初始化EMD模型的对象          此方法在创建EMDModel类的实例时被调用，用于初始化对象的属性。          参数:          datax (numpy.ndarray或类似可迭代对象): 数据的x值序列。          datay (numpy.ndarray或类似可迭代对象): 数据的y值序列，与datax中的元素一一对应。          """          self.datax = datax  # 存储输入数据的x值序列，用于后续各种计算和操作          self.datay = datay  # 存储输入数据的y值序列，与datax相对应，是主要的分析对象          self.H = []  # 用于储存通过后续处理得到的IMF（本征模态函数）分量，初始化为空列表      def FindMax(self, Datax, Datay):          """          寻找数据中的极大值点          该方法遍历输入的数据序列，按照一定的条件判断并找出其中的极大值点及其对应的x值。          参数:          Datax (numpy.ndarray或类似可迭代对象): 待分析数据的x值序列。          Datay (numpy.ndarray或类似可迭代对象): 待分析数据的y值序列，与Datax中的元素一一对应。          返回:          tuple: 包含两个列表，第一个列表是极大值点对应的x值，第二个列表是极大值点对应的y值。          """          x, y = [], []  # 初始化用于存储极大值点的x值和y值的空列表          if Datay[0] >= Datay[1]:  # 如果第一个数据点的y值大于等于第二个数据点的y值              x.append(1)  # 将x值设为1（可能是数据序列的索引从1开始的一种约定，具体需根据数据含义确定）              y.append(Datay[0])  # 将第一个数据点的y值作为极大值点的y值          for i in range(1, len(Datax) - 2):  # 遍历数据序列中除了首尾几个特殊点之外的部分              if Datay[i] >= Datay[i + 1] and Datay[i] >= Datay[i - 1]:  # 如果当前数据点的y值大于等于前后相邻数据点的y值                  x.append(Datax[i])  # 将当前数据点的x值添加到极大值点的x值列表中                  y.append(Datay[i])  # 将当前数据点的y值添加到极大值点的y值列表中          if Datay[len(Datax) - 1] >= Datay[len(Datax) - 2]:  # 如果最后一个数据点的y值大于等于倒数第二个数据点的y值              x.append(Datax[len(Datax) - 1])  # 将最后一个数据点的x值添加到极大值点的x值列表中              y.append(Datay[len(Datax) - 1])  # 将最后一个数据点的y值添加到极大值点的y值列表中          return x, y  # 返回极大值点对应的x值和y值的列表      def FindMin(self, Datax, Datay):          """          寻找数据中的极小值点          与FindMax方法类似，该方法用于遍历输入的数据序列，找出其中的极小值点及其对应的x值。          参数:          Datax (numpy.ndarray或类似可迭代对象): 待分析数据的x值序列。          Datay (numpy.ndarray或类似可迭代对象): 待分析数据的y值序列，与Datax中的元素一一对应。          返回:          tuple: 包含两个列表，第一个列表是极小值点对应的x值，第二个列表是极小值点对应的y值。          """          x, y = [], []  # 初始化用于存储极小值点的x值和y值的空列表          if Datay[0] <= Datay[1]:  # 如果第一个数据点的y值小于等于第二个数据点的y值              x.append(Datax[0])  # 将第一个数据点的x值添加到极小值点的x值列表中              y.append(Datay[0])  # 将第一个数据点的y值添加到极小值点的y值列表中          for i in range(1, len(Datax) - 1):  # 遍历数据序列中除了首尾特殊点之外的部分              if Datay[i] <= Datay[i + 1] and Datay[i] <= Datay[i - 1]:  # 如果当前数据点的y值小于等于前后相邻数据点的y值                  x.append(Datax[i])  # 将当前数据点的x值添加到极小值点的x值列表中                  y.append(Datay[i])  # 将当前数据点的y值添加到极小值点的y值列表中          if Datay[len(Datax) - 1] <= Datay[len(Datax) - 2]:  # 如果最后一个数据点的y值小于等于倒数第二个数据点的y值              x.append(Datax[len(Datax) - 1])  # 将最后一个数据点的x值添加到极小值点的x值列表中              y.append(Datay[len(Datax) - 1])  # 将最后一个数据点的y值添加到极小值点的y值列表中          return x, y  # 返回极小值点对应的x值和y值的列表      def getCubicLine(self, Datax, Datay, length):          """          获取极大值和极小值点的包络线          该方法首先找到数据的极大值点和极小值点，然后利用插值函数生成通过这些极值点的三次样条曲线，作为包络线。          参数:          Datax (numpy.ndarray或类似可迭代对象): 待分析数据的x值序列。          Datay (numpy.ndarray或类似可迭代对象): 待分析数据的y值序列，与Datax中的元素一一对应。          length (int): 生成包络线时使用的x值序列长度，通常用于控制包络线的分辨率。          返回:          tuple: 包含两条曲线的y值序列，分别是极大值点包络线的y值序列和极小值点包络线的y值序列。          """          x, y = self.FindMax(Datax, Datay)  # 找到数据中的极大值点及其对应的x值和y值          tck = interpolate.splrep(x, y, k=3)  # 使用scipy的插值函数生成通过极大值点的三次样条函数的表示形式          xx = np.linspace(min(Datax), max(Datax), length)  # 生成在Datax的最小值和最大值之间等间距的x值序列，用于计算包络线的y值          ymax = interpolate.splev(xx, tck, der=0)  # 通过三次样条函数计算极大值点包络线的y值序列          x, y = self.FindMin(Datax, Datay)  # 找到数据中的极小值点及其对应的x值和y值          tck = interpolate.splrep(x, y, k=3)  # 使用scipy的插值函数生成通过极小值点的三次样条函数的表示形式          ymin = interpolate.splev(xx, tck, der=0)  # 通过三次样条函数计算极小值点包络线的y值序列          return ymax, ymin  # 返回极大值点包络线的y值序列和极小值点包络线的y值序列      def FindZeros(self, Datax, Datay):          """          找到过零点的数量          该方法遍历数据序列，检查相邻数据点的y值乘积是否小于零，以此来确定过零点的数量。          参数:          Datax (numpy.ndarray或类似可迭代对象): 待分析数据的x值序列。          Datay (numpy.ndarray或类似可迭代对象): 待分析数据的y值序列，与Datax中的元素一一对应。          返回:          int: 过零点的数量。          """          num = 0  # 初始化过零点数量为0          for i in range(0, len(Datax) - 1):  # 遍历数据序列中除了最后一个数据点之外的部分              if Datay[i] \* Datay[i + 1] < 0:  # 如果相邻两个数据点的y值乘积小于零，说明存在过零点                  num += 1  # 过零点数量加1          return num  # 返回过零点的数量      def is\_IMF(self, Hdata):          """          判断是否为IMF分量          该方法根据一系列条件判断输入的数据序列是否符合IMF（本征模态函数）的定义。          参数:          Hdata (numpy.ndarray或类似可迭代对象): 待判断的数据序列。          返回:          bool: 如果数据序列符合IMF的定义，则返回True；否则返回False。          """          x = np.linspace(1, len(Hdata), len(Hdata))  # 生成与输入数据序列长度相同的等间距x值序列，可能用于后续分析与数据序列的对应关系          xx, y1 = self.FindMax(x, Hdata)  # 找到数据序列中的极大值点及其对应的x值和y值          xx, y2 = self.FindMin(x, Hdata)  # 找到数据序列中的极小值点及其对应的x值和y值          num1 = len(y1) + len(y2)  # 计算极大值点和极小值点的总数          num2 = self.FindZeros(x, Hdata)  # 找到数据序列中的过零点数量          # 判断局部极值点的数量和过零点的数量是否相等（允许有一定的误差，这里误差范围设定为相差不超过1）          flag1 = abs(num1 - num2) <= 1          ymax, ymin = self.getCubicLine(x, Hdata, len(Hdata))  # 获取数据序列的极大值点和极小值点的包络线          yaver = (ymax + ymin) / 2  # 计算包络线的平均值          num3 = 0          error = 0.1          for i in  range(0, len(Hdata)):              if abs(yaver[i]) < error:  # 如果包络线平均值的绝对值小于设定的误差值                  num3 += 1  # 满足条件的点数量加1          flag2 = num3 / len(Hdata) >= 0.95  # 判断满足条件的点占总点数的比例是否达到设定的阈值          return flag1 and flag2  # 如果两个条件都满足，则返回True，表示是IMF分量；否则返回False      def CalSD(self, h\_k\_1, h\_k):          """          计算SD值          该方法根据输入的两个数据序列计算它们之间的SD（标准偏差）值。          参数:          h\_k\_1 (numpy.ndarray或类似可迭代对象): 第一个数据序列。          h\_k (numpy.ndarray或类似可迭代对象): 第二个数据序列。          返回:          float: 计算得到的SD值。          """          SD = h\_k\_1 - h\_k  # 计算两个数据序列对应元素的差值          SD = np.sum(SD \* SD)  # 计算差值的平方和          SD = SD / np.sum(h\_k \* h\_k)  # 将差值的平方和除以第二个数据序列的平方和，得到SD值          return SD  # 返回计算得到的SD值      def extract\_imf(self, N=3):          """          提取多个IMF分量          该方法通过迭代的方式从输入数据中提取指定数量的IMF分量。          参数:          N (int, 可选): 要提取的IMF分量数量，默认值为3。          返回:          tuple: 包含两个元素，第一个元素是提取得到的IMF分量列表，第二个元素是提取完IMF分量后剩余的数据。          """          InitalData = self.datay  # 将输入数据的y值序列作为初始数据          for j in range(1, N + 1):  # 进行指定次数的迭代提取              ymax, ymin = self.getCubicLine(self.datax, InitalData, len(self.datax))  # 获取当前数据的极大值点和极小值点的包络线              # self.draw(self.datax, InitalData, ymax, ymin, len(self.datax))  # 可能是用于绘制相关图形的函数，这里暂时注释掉              m = (ymax + ymin) / 2  # 计算包络线的平均值              h = InitalData - m  # 用初始数据减去包络线平均值得到中间数据              is\_accept = self.is\_IMF(h)  # 判断中间数据是否符合IMF分量的定义              num = 0              while not is\_accept:  # 如果不符合IMF分量的定义，则进行循环调整                  num += 1                  PreH = h  # 记录当前的中间数据                  ymax, ymin = self.getCubicLine(self.datax, PreH, len(self.datax))  # 获取新的包络线                  m = (ymax + ymin) / 2  # 计算新的包络线平均值                  h = PreH - m  # 用之前的中间数据减去新的包络线平均值得到新的中间数据                  is\_accept = self.is\_IMF(h)  # 再次判断新的中间数据是否符合IMF分量的定义              self.H.append(h)  # 将符合IMF分量定义的中间数据添加到IMF分量列表中              InitalData = InitalData - h  # 用初始数据减去已提取的IMF分量，得到剩余的数据用于下一次迭代          return self.H, InitalData  # 返回提取得到的IMF分量列表和剩余的数据      def fit\_residual(self, degree=3):          """          对余量进行多项式拟合          该方法使用多项式拟合的方式对剩余的数据（可能是提取完IMF分量后剩余的数据）进行拟合，得到拟合后的预测值。          参数:          degree (int, 可选): 多项式的次数，默认值为3。          返回:          numpy.ndarray: 拟合后的预测值序列，与输入数据的x值序列相对应。          """          z1 = np.polyfit(self.datax, self.datay, degree)  # 使用numpy的polyfit函数根据输入数据的x值和y值序列以及指定的多项式次数，拟合得到多项式的系数          p1 = np.poly1d(z1)  # 根据拟合得到的系数创建一个多项式对象          y\_pred = p1(self.datax)  # 使用多项式对象计算在输入数据的x值序列上的预测值          return y\_pred  # 返回拟合后的预测值序列 |

|  |
| --- |
| 模型比较代码如下 |
| import numpy as np  from GM import GMModel  from EMD import EMDModel  def compare\_models():      """      比较GM模型和EMD模型的性能。      该函数生成了数据，初始化了GM模型和EMD模型实例，设置了模型参数，计算了相关参数，      并返回了GM模型和EMD模型的均方根误差（RMSE）和均值绝对误差（MAE）。      """      # 生成数据      datax = np.linspace(1, 34, 34)  # 生成从1到34的等间隔的34个数据点作为x轴数据      datay = np.array([9, 12, 11, 4, 7, 2, 5, 8, 5, 7, 1, 6, 1, 9, 4, 1, 3, 8, 6, 1, 1, 33, 7, 91, 2, 1, 87, 47, 12, 9, 135, 258, 16, 35], float)  # 给定的y轴数据点      # GM模型      gm\_model = GMModel()  # 初始化GM模型实例      gm\_model.set\_model(datay.tolist())  # 设置GM模型的参数为给定的y轴数据点列表      gm\_predicted = gm\_model.modeling\_result\_arr  # 获取GM模型对实验数据的拟合值数组      gm\_residuals = datay - gm\_predicted  # 计算GM模型的残差      # EMD模型      emd\_model = EMDModel(datax, datay)  # 初始化EMD模型实例，传入x轴和y轴数据      imfs, residual = emd\_model.extract\_imf(N=2)  # 从输入数据中提取指定数量的IMF分量      emd\_predicted = emd\_model.fit\_residual(degree=3)  # 对剩余数据进行拟合，获取拟合值      emd\_residuals = datay - emd\_predicted  # 计算EMD模型的残差      # 计算均方根误差（RMSE）      gm\_rmse = np.sqrt(np.mean(gm\_residuals \*\* 2))  # 计算GM模型的均方根误差（RMSE）      emd\_rmse = np.sqrt(np.mean(emd\_residuals \*\* 2))  # 计算EMD模型的均方根误差（RMSE）      # 计算均值绝对误差（MAE）      gm\_mae = np.mean(np.abs(gm\_residuals))  # 计算GM模型的均值绝对误差（MAE）      emd\_mae = np.mean(np.abs(emd\_residuals))  # 计算EMD模型的均值绝对误差（MAE）      return gm\_rmse, emd\_rmse, gm\_mae, emd\_mae  # 返回GM模型和EMD模型的均方根误差（RMSE）和均值绝对误差（MAE）  if \_\_name\_\_ == "\_\_main\_\_":      gm\_rmse, emd\_rmse, gm\_mae, emd\_mae = compare\_models()      print("GM模型均方根误差（RMSE）:", gm\_rmse)      print("EMD模型均方根误差（RMSE）:", emd\_rmse)      # 比较均方根误差（MSE）并输出结果      if gm\_rmse < emd\_rmse:          print("在均方根误差（RMSE）指标上，GM模型表现更好。")      elif gm\_rmse > emd\_rmse:          print("在均方根误差（RMSE）指标上，EMD模型表现更好。")      else:          print("在均方根误差（RMSE）指标上，GM模型和EMD模型表现相同。")        print("GM模型均值绝对误差（MAE）:", gm\_mae)      print("EMD模型均值绝对误差（MAE）:", emd\_mae)      # 比较均值绝对误差（MAE）并输出结果      if gm\_mae < emd\_mae:          print("在均值绝对误差（MAE）指标上，GM模型表现更好。")      elif gm\_mae > emd\_mae:          print("在均值绝对误差（MAE）指标上，EMD模型表现更好。")      else:          print("在均值绝对误差（MAE）指标上，GM模型和EMD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 GM import GMModel  from EMD import EMDModel  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。      该函数生成了数据，初始化了GM模型和EMD模型实例，设置了模型参数，计算了相关参数，      并绘制了原始数据、GM模型和EMD模型的拟合曲线。      """      # 生成数据      datax = np.linspace(1, 34, 34)  # 生成从1到34的等间隔的34个数据点作为x轴数据      datay = np.array([9, 12, 11, 4, 7, 2, 5, 8, 5, 7, 1, 6, 1, 9, 4, 1, 3, 8, 6, 1, 1, 33, 7, 91, 2, 1, 87, 47, 12, 9, 135, 258, 16, 35], float)  # 给定的y轴数据点      # GM模型      gm\_model = GMModel()  # 初始化GM模型实例      gm\_model.set\_model(datay.tolist())  # 设置GM模型的参数为给定的y轴数据点列表      gm\_predicted = gm\_model.get\_predicted\_data()  # 获取GM模型对实验数据的拟合值数组      # EMD模型      emd\_model = EMDModel(datax, datay)  # 初始化EMD模型实例，传入x轴和y轴数据      imfs, residual = emd\_model.extract\_imf(N=2)  # 从输入数据中提取指定数量的IMF分量      emd\_predicted = emd\_model.fit\_residual(degree=3)  # 对剩余数据进行拟合，获取拟合值      plt.figure(figsize=(10, 6))  # 创建一个新的图形，设置图形大小为10x6英寸      plt.plot(datax, datay, 'o', label='原始数据', color='black')  # 绘制原始数据点，用黑色圆圈表示，添加标签“原始数据”      plt.plot(datax, gm\_predicted, label='GM模型拟合', linestyle='-', color='blue')  # 绘制GM模型的拟合曲线，用蓝色实线表示，添加标签“GM模型拟合”      plt.plot(datax, emd\_predicted, label='EMD模型拟合', linestyle='--', color='red')  # 绘制EMD模型的拟合曲线，用红色虚线表示，添加标签“EMD模型拟合”      plt.title('模型拟合对比')  # 设置图形标题为“模型拟合对比”      plt.xlabel('数据x值')  # 设置x轴标签为“数据x值”      plt.ylabel('数据y值')  # 设置y轴标签为“数据y值”      plt.legend()  # 显示图例      plt.show()  # 显示图形  if \_\_name\_\_ == "\_\_main\_\_":      visualize\_models() |

## 3.3 数据来源

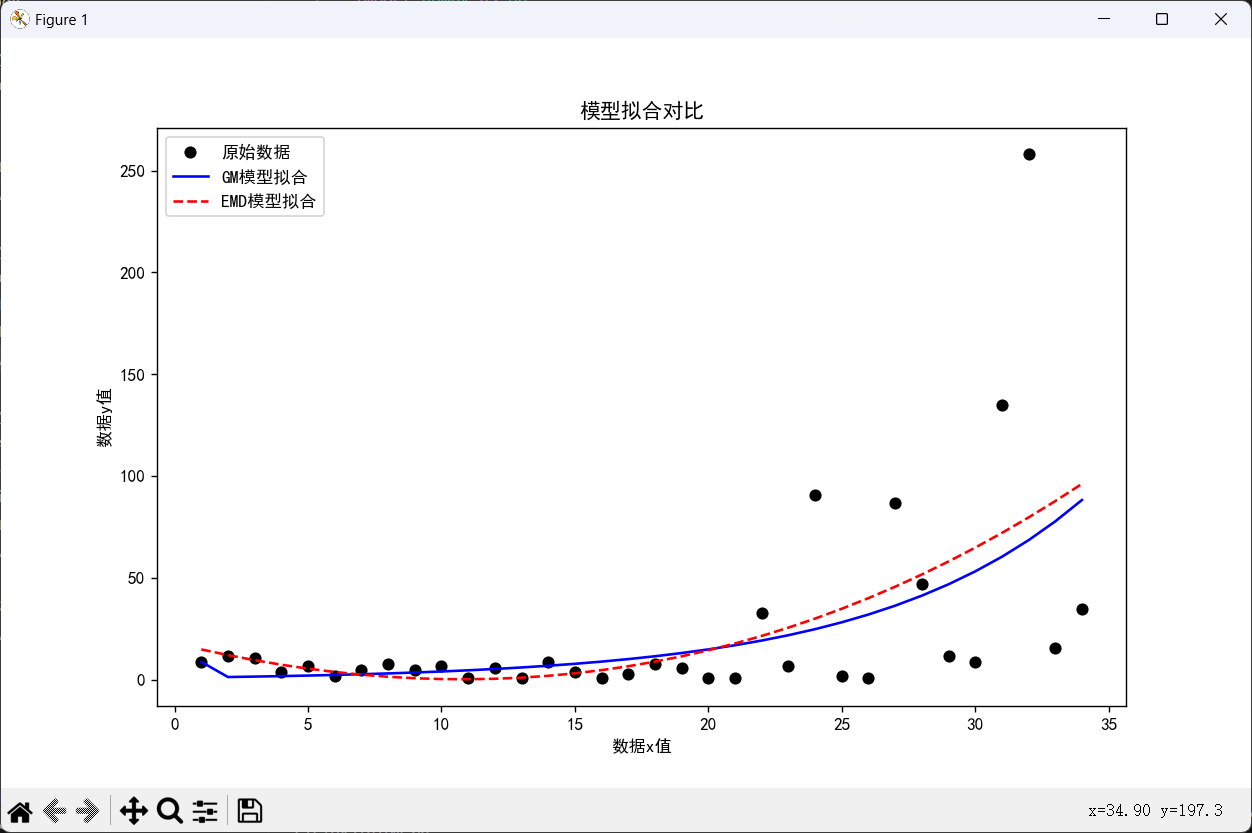
数据来源是自己编写的数据：



## 结果展示

GM模型和EMD模型拟合结果如下：





# 4.算法结果分析

在对模型预测效果进行评估时，均方根误差（RMSE）与均值绝对误差（MAE）是两个极为关键的衡量指标。通过深入分析这两个指标在不同模型上的表现情况，能够精准洞察模型的特性与优势。

就均方根误差（RMSE）指标而言，EMD 模型展现出了更为卓越的性能。RMSE 指标着重考量的是预测值与真实值之间偏差的平方和的均值的平方根。EMD 模型在这一指标上表现出色，意味着它在预测过程中，对于整体的误差分布有着较好的把控能力。其预测值与真实值的偏差平方和相对较小，从而在经过一系列数学运算后得到的均方根误差值较低。这反映出 EMD 模型能够较为稳定地贴近真实值进行预测，不会出现过大的偏差波动，使得预测结果在整体上与实际情况具有较高的一致性。

而在均值绝对误差（MAE）指标方面，GM 模型则脱颖而出。MAE 指标聚焦于预测值与真实值偏差的绝对值的平均值。GM 模型在此指标上的良好表现，表明它在处理单个数据点的预测偏差时具有独特的优势。该模型能够有效地减少单个预测值与真实值之间的绝对误差，即对于每一个预测数据点，其与对应的真实数据点的偏离程度在绝对值意义上相对较小。这使得 GM 模型在对数据细节的预测准确性上表现出色，能够较为精准地捕捉到数据的局部变化特征。

综合来看，尽管 EMD 模型和 GM 模型在不同指标上各擅胜场，但不可忽视的是，两种模型的预测效果均处于良好的范畴之内。它们之间的差异仅仅是略微的，这说明在实际应用场景中，这两种模型都能够为预测任务提供可靠且具有一定精度的结果。无论是对于需要考虑整体误差稳定性的大规模数据预测，还是对于注重局部数据点准确性的精细化预测，这两种模型都有其用武之地，可以根据具体的需求和数据特点灵活选择合适的模型，以实现最佳的预测效果与应用价值。