

数值分析上机研究报告

姓 名： 佟文轩

学 号： 1120240934

专 业： 计算机科学与技术

指导老师： 孙新

2025 年 11 月 7 日

摘要

在数值分析的第二章课程中我们学习了非线性方程的迭代解法；在第三章与第四章中我们学习了线性方程组的直接解法和迭代解法，这一类方法利用了计算机高效的计算能力，是为计算机量身定制的，不同于以往我们在学习中接触到的解析式解法和克莱姆法则。因此，本报告的目的是在计算机中实际应用上述迭代方法，以加深自己对相关知识的理解。

本报告选择使用 JAVA 语言来进行上机实现，原因有三：其一为 JAVA 还提供较为方便的 javafx 方便将我的程序更好的可视化；其二 JAVA 可以将我的程序方便的转化为软件，方便他人一键安装使用；其三本学期新增 JAVA 选修课可以借此机会增加对 JAVA 语言的熟练度。

在软件的子功能方程的迭代解法中，我手动编写了有关方法——牛顿法、艾特肯迭代法、单点弦截法等 8 种方法，其余部分借助 AI 完成相关内容的编写。该软件展现了方程迭代在二维坐标系和一维坐标轴上的变化，方便我们从不同的角度去观察映射对迭代行为的影响。此外，本程序记录每次迭代产生的相关数据，包括 x 轴坐标值、 $f(x)$ 值、相邻两次迭代 x 轴坐标差值等内容。

在软件的另一个子功能线性方程组的解法中，我编写了直接法——高斯消元法、克劳特消元法、列主元素法和全主元素法，这部分内容通过矩阵的实时变换来进行可视化；同时我编写了迭代法——雅可比迭代法、高斯赛德尔迭代法、松弛迭代法，这部分内容通过残差变化曲线图来进行迭代效率的可视化。

本报告展示了不同方法的代码实现、对比了不同方法之间的差异，最后提供了一个安装程序，以便他人可以方便的使用迭代方法的演示功能。以下为我的 GitHub 仓库：<https://github.com/twx145/Numerical-Analysis>，其中记录了所有相关代码和 latex^[1]文案。

关键词：方程迭代解法；线性方程组的数值解法；JAVA 方程（组）求解软件

目录

1	引言	1
2	方程的迭代解法	1
2.1	理论基础	1
2.1.1	压缩映射	1
2.1.2	压缩映射的性质	2
2.1.3	如何判断压缩映射	2
2.1.4	方程迭代求根	2
2.2	详细迭代方法	3
2.2.1	普通迭代法	3
2.2.2	牛顿迭代法	4
2.2.3	简化切线法	6
2.2.4	修正切线法	7
2.2.5	牛顿下山法	8
2.2.6	单点弦截法	9
2.2.7	双点弦截法	10
2.2.8	艾特肯迭代法	11
3	线性方程组的直接解法	13
3.1	消元法综述	13
3.1.1	消元法介绍	13
3.1.2	消元法核心理论	13
3.1.3	消元法步骤	13
3.1.4	优缺点	13
3.2	详细消元法	14
3.2.1	高斯消元法	14
3.2.2	克劳特消元法	15
3.2.3	高斯列主元素消元法	16
3.2.4	高斯全主元素消元法	17
4	线性方程组的迭代解法	18
4.1	线性方程组的迭代法综述	18
4.1.1	线性方程组的迭代法介绍	18
4.1.2	消元法步骤	18
4.1.3	优缺点	18
4.2	详细迭代方法	19
4.2.1	雅可比迭代法	19
4.2.2	高斯-赛德尔迭代法	20

目录

4.2.3 松弛迭代法	21
5 java 软件实现	22
5.1 总体设计	22
5.1.1 线性方程组求解模块	23
5.1.2 非线性方程求根模块	23
5.2 核心功能实现	24
5.2.1 非线性方程求根实现	24
5.2.2 线性方程组求解实现	24
5.2.3 函数字符串识别与函数求值	25
5.3 软件界面介绍	26
6 总结	28
6.1 工作总结	28
6.2 展望	28
致谢	2
附录	3
A 方程的迭代解法完整代码	3
B 线性方程组的直接解法完整代码	8
C 线性方程组的迭代解法完整代码	12
D 函数识别与求值部分完整代码	15

1 引言

非线性方程因其通常能比线性方程更好的描述事物的规律而在学习生活中很常见，但是却鲜有非线性方程可以求得解析解，因此，为计算机计算方程解而量身打造的迭代法变得很重要，迭代法通过一系列逐步逼近可以得到预期精度的解，具有高效精准的特点。

线性方程组当今在深度学习大模型训练领域应用广泛，利用高效精准的方程组求解算法可以显著增加模型训练的效率和成功率。不同于线性代数中经典解法——克莱姆法则的多次矩阵运算带来的低效，方程组的直接解法为小型稠密矩阵提供了求解方案，方程组的迭代解法则给大型稀疏矩阵的求解带来了巨大的便利。

本报告的核心任务包含以下三点：一是研究课内外常见的数值解法^[2]；二为利用 JAVA 打造一个方程（组）求解可视化软件。该软件允许用户自主输入不同的方程（组）并选择不同策略来执行求解操作，此外我的软件还提供了相关的指标来分析各个迭代方法；三是通过数值实验的方法来比较不同迭代方法在收敛速度和稳定性上的表现。

2 方程的迭代解法

本节着重介绍迭代法的原理，同时详细的介绍 8 种迭代法各自的基本思想、数学公式、算法步骤、收敛性分析与优缺点。

2.1 理论基础

这里我们先讨论一下何为压缩映射，以及压缩映射的性质，稍后再讨论压缩映射与其性质是如何帮助我们通过迭代法算出方程解的。

2.1.1 压缩映射

压缩映射是针对一个函数映射在某一个闭区间上而言的。若在一个区间内每经过一次映射变换，各个点依旧在这个区间内并且它们之间的距离都缩小，我们则称该映射在该区间是一个压缩映射。

更加严谨的来说，我们有一个映射 $g(x)$ 和一个区间 $I = [x_1, x_2]$ 。若在该区间 I 内任意两点 a, b ，在经过映射后的 $g(a)$ 和 $g(b)$ 依旧在区间 I 内，并且距离缩小——即存在一个 $L < 1$ 使得：

$$|g(a) - g(b)| \leq L|a - b| \quad (1)$$

则我们称该映射为压缩映射。这个常数 L 被称为压缩因子。 $L < 1$ 这个条件保证了“收缩”的发生，压缩映射示例见图1。

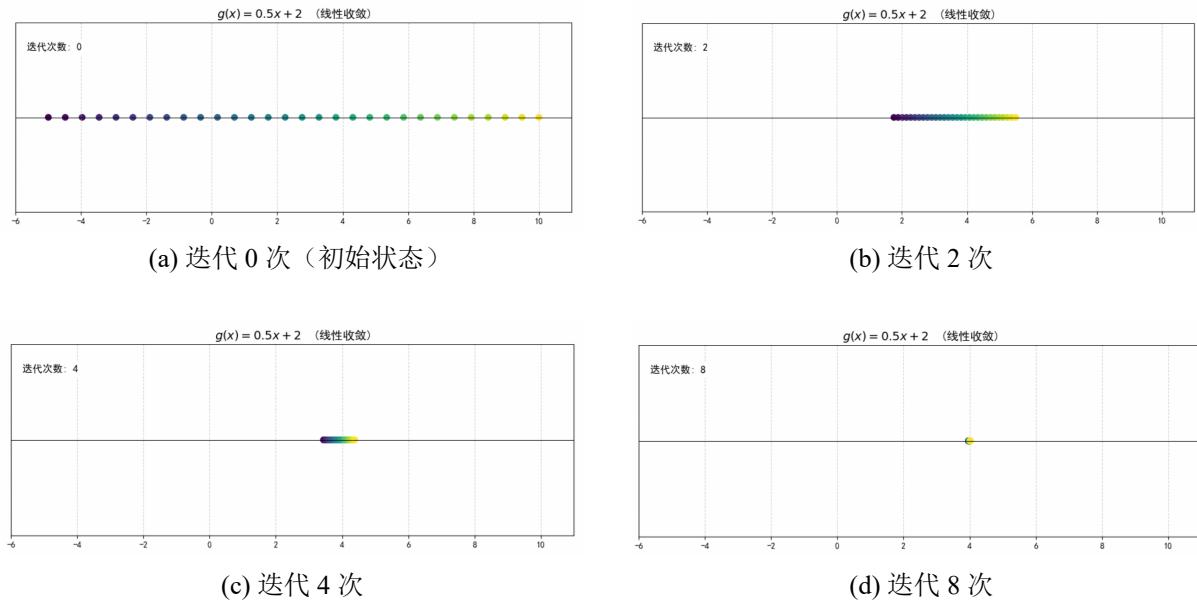


图 1 压缩映射示意图 (方程 $g(x)=0.5x+2$ 在不同迭代次数下压缩程度)

2.1.2 压缩映射的性质

一个压缩映射可以在多次迭代后将区间内的点汇集到一个称为不动点的位置。

严谨的来说，我们可以构建一个压缩多次后两点间距离上界的等比数列，由于 $L < 1$ 这个条件的存在，我们不难得知多次迭代后这个数列必将收敛于 0。

2.1.3 如何判断压缩映射

根据微积分中的中值定理，我们知道对于任意的 x 和 y ，在它们之间必定存在一个点 c ，使得：

$$g(x) - g(y) = g'(c) \cdot (x - y) \quad (2)$$

两边取绝对值得到：

$$|g(x) - g(y)| = |g'(c)| \cdot |x - y| \quad (3)$$

现在将这个公式与压缩映射的定义 $|g(x) - g(y)| \leq L|x - y|$ 我们不难发现： $|g'(x)|$ 的大小决定了映射是压缩还是发散的。如果我们能在一包含根的区间 I 上找到一个常数 $L < 1$ ，使得对区间 I 内所有的 x ，都有 $|g'(x)| \leq L$ ，那么 $g(x)$ 在该区间上就是一个压缩映射。

2.1.4 方程迭代求根

现在我们知道了一个压缩映射 $g(x)$ 通过多次迭代可以收敛于一个不动点 x_0 ($x_0 = g(x_0)$)，那我们就可以将原始方程 $f(x) = 0$ 变形为 $x = g(x)$ ，随后通过多次迭代算出不动点 $x_0 = g(x_0)$ ，这里 x_0 就是方程的解。

2.2 详细迭代方法

2.2.1 普通迭代法

基本思想

普通迭代法的主要想法就是朴素的将方程 $f(x) = 0$ 等价变形为 $x = g(x)$ 从而直接求得这个新函数的不动点。选定一个初始近似值 x_0 后，通过迭代公式 $x_{k+1} = g(x_k)$ 产生一个序列 x_0, x_1, x_2, \dots 。如果这个序列收敛于某个值 x_0 ，那么 x_0 就是方程的解。

数学公式

普通迭代法的迭代公式为：

$$x_{k+1} = g(x_k), \quad k = 0, 1, 2, \dots \quad (4)$$

算法步骤

1. 将方程 $f(x) = 0$ 转换为 $x = g(x)$ 。
2. 选取一个初始近似值 x_0 和一个允许误差 ϵ 。
3. 根据迭代公式 $x_{k+1} = g(x_k)$ 计算下一个近似值。
4. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。若满足，则停止迭代， x_{k+1} 即为所求解的近似根。
5. 若不满足收敛条件，则令 $x_k = x_{k+1}$ ，返回第 3 步继续迭代。

收敛性分析

不动点迭代法的收敛性与迭代函数 $g(x)$ 的性质密切相关。根据上述的压缩映射相关知识，我们知道只有迭代函数 $g(x)$ 满足以下两个条件迭代过程才收敛：

1. 对于任意 $x \in [a, b]$ ，都有 $g(x) \in [a, b]$ 。
2. 存在一个常数 $0 \leq L < 1$ ，使得对于任意 $x \in (a, b)$ ，都有 $|g'(x)| \leq L$ 。

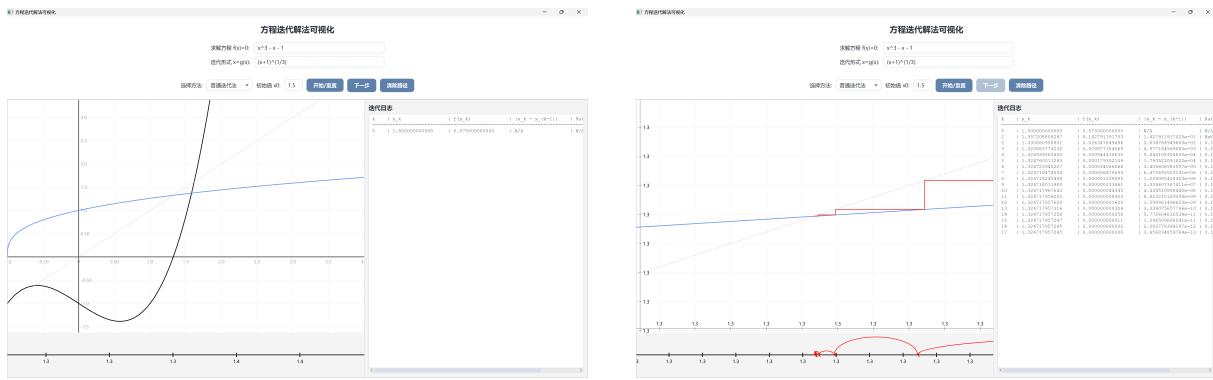
优缺点

- 优点：算法简单朴素，易于理解。
- 缺点：
 - 迭代函数的构造不唯一，不同的构造方式可能导致收敛性不同，甚至发散。
 - 收敛速度通常较慢，仅为一阶收敛。
 - 对初值的选取比较敏感，需要选取在根附近的初值才能保证收敛。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 和迭代函数 $g(x) = \sqrt[3]{x + 1}$ 时的软件运行截图（详细代码见附录 A）。

2 方程的迭代解法



(a) 普通迭代法软件运行截图 1

(b) 普通迭代法软件运行截图 2

图 2 普通迭代法软件运行截图

2.2.2 牛顿迭代法

基本思想

从普通迭代法的收敛性理论我们知道，迭代函数的导数绝对值越小，收敛速度就越快。牛顿法正是基于这一思想，改造迭代函数将导数的绝对值变为零以快速收敛。

我们将原始方程 $f(x) = 0$ 转化为等价的不动点形式 $x = g(x)$ 。为了引入可调节的参数，我们构造一个更普遍的迭代函数：

$$g(x) = x + \alpha(x)f(x) \quad (5)$$

其中 $\alpha(x)$ 是一个待定的函数。只要 $f(x) = 0$ ，这个形式就等价于 $x = g(x)$ 。由上述推理可知我们希望在根 x_0 附近有 $|g'(x)| \approx 0$ 。对 $g(x)$ 求导可得：

$$g'(x) = 1 + \alpha'(x)f(x) + \alpha(x)f'(x) \quad (6)$$

在根 x_0 附近，由于 $f(x) \approx 0$ ，上式简化为：

$$g'(x) = 1 + \alpha(x)f'(x) \quad (7)$$

我们令 $g'(x) = 0$ ，解得：

$$\alpha(x) = -\frac{1}{f'(x)} \quad (8)$$

由此，我们选择 $\alpha(x) = -\frac{1}{f'(x)}$ 作为我们的迭代函数，便得到了牛顿迭代法。它通过每一步都选取能让迭代函数导数趋近于零的方向，从而实现了局部二阶收敛。

数学公式

我们将 $\alpha(x_k) = -\frac{1}{f'(x_k)}$ 代入迭代公式 $x_{k+1} = x_k + \alpha(x_k)f(x_k)$ ，即可得到牛顿法的

迭代公式:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)} \quad (9)$$

算法步骤

1. 选取一个初始近似值 x_0 和一个允许误差 ϵ 。
2. 计算函数值和导数值，计算 $f(x_k)$ 和一阶导数 $f'(x_k)$ 。
3. 根据迭代公式 $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ 计算下一个近似值。
4. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。若满足，则停止迭代， x_{k+1} 即为所求解的近似根。
5. 若不满足收敛条件，则令 $x_k = x_{k+1}$ ，返回第 3 步继续迭代。

收敛性分析

牛顿法有局部收敛特性，且在一定情况收敛速度很快。

1. 若 x_0 为方程单根且选定的初值距离根较近则收敛阶为 2。
2. 若 x_0 为方程重根则收敛阶为 1。
3. 迭代函数是否收敛严重依赖于初值的选择，只有初值距离根足够接近才会收敛。

优缺点

- 优点：

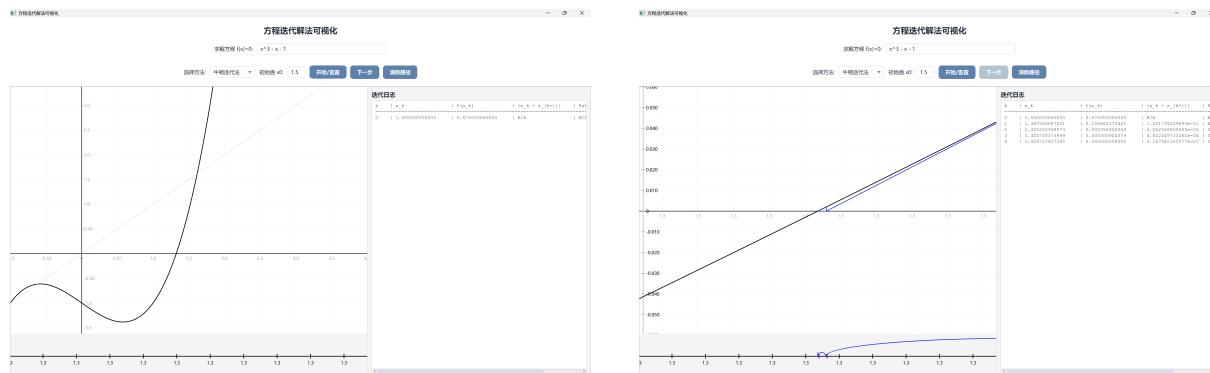
- 收敛速度快，对于单根，具有二阶收敛性。
- 算法简洁，迭代公式形式简单易于理解。

- 缺点：

- 初始值敏感，只有在跟附近才可能收敛到根处。
- 需要计算导数，对复杂函数求导计算量大，该方法受限。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 时的软件运行截图（详细代码见附录 A）。



(a) 牛顿迭代法软件运行截图 1

(b) 牛顿迭代法软件运行截图 2

图 3 牛顿迭代法软件运行截图

2.2.3 简化切线法

基本思想

牛顿法需要在每次迭代时都计算导数 $f'(x_k)$, 当导数函数 $f'(x)$ 的形式复杂时, 会出现计算量大、耗时等问题。简化切线法因此诞生。该方法只在初始点 x_0 计算一次导数 $f'(x_0)$, 并在后续的所有迭代中都使用这个固定的斜率来代替牛顿迭代法种变化的 $f'(x_k)$ 。

数学公式

迭代公式为:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_0)} \quad (10)$$

算法步骤

1. 选取一个初始近似值 x_0 和一个允许误差 ϵ 。
2. 计算一次导数值作为固定斜率 $m = f'(x_0)$ 。
3. 根据迭代公式 $x_{k+1} = x_k - \frac{f(x_k)}{m}$ 计算下一个近似值。
4. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。若满足, 则停止迭代。
5. 若不满足, 则令 $x_k = x_{k+1}$, 返回第 3 步。

收敛性分析

该方法线性收敛, 收敛速度慢于牛顿法。当初始值 x_0 离根足够近并且在根的邻域内满足条件 $|1 - f'(x)/f'(x_0)| < 1$, 该方法是收敛的。

优缺点

- 优点: 极大地减少了计算量 (只计算一次导数)。
- 缺点:
 - 收敛速度从牛顿法的二阶下降到一阶, 收敛速度变慢。
 - 对初始值的选取较为敏感。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 时的软件运行截图。

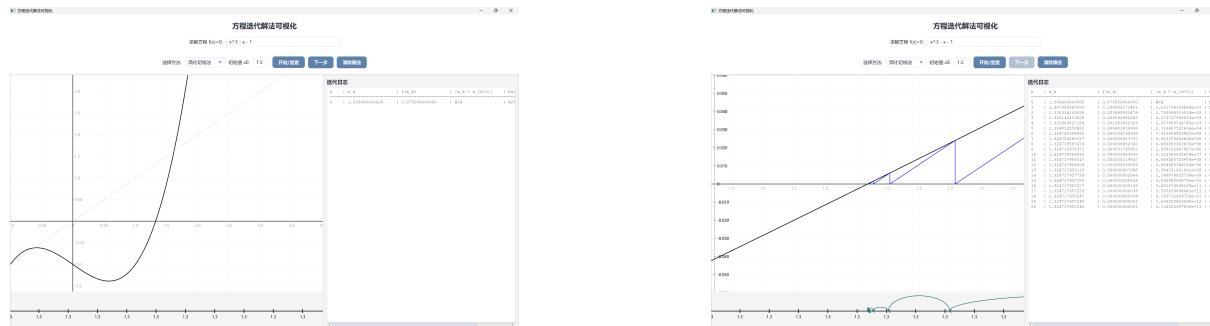


图 4 简化切线法软件运行截图

2.2.4 修正切线法

基本思想

修正切线法是牛顿法和简化切线法的一种折中方案。这种方法会定期执行一次导数计算，这既避免了牛顿法多次大量的导数计算又增加了简化切线法的收敛速度，在计算效率和收敛速度上取得了平衡。

数学公式

迭代公式为：

$$x_{k+1} = x_k - \frac{f(x_k)}{m_j} \quad (11)$$

其中斜率 m_j 会定期更新。这里我的程序设置了一个更新间隔 N ，当迭代次数 k 是 N 的倍数时才更新斜率 $m_j = f'(x_k)$ 。

算法步骤

1. 选取初始值 x_0 , 误差 ϵ , 以及更新间隔 N 。
2. 初始化斜率 $m = f'(x_0)$ 。
4. 根据公式 $x_{k+1} = x_k - \frac{f(x_k)}{m}$ 计算下一个近似值。
5. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。若满足，则停止迭代。
6. 如果 $(k + 1) \pmod N = 0$, 则更新斜率 $m = f'(x_{k+1})$ 。
7. 若不满足收敛条件, 则令 $x_k = x_{k+1}$, 返回第三步。

收敛性分析

收敛性介于牛顿法和简化切线法之间。更新频率 N 越小, 收敛速度越快, 但单步计算成本也越高。

优缺点

- 优点：计算成本和收敛速度相对平衡。
- 缺点：需要额外设定一个更新频率参数 N , 选择不当会影响效率。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 时的软件运行截图。

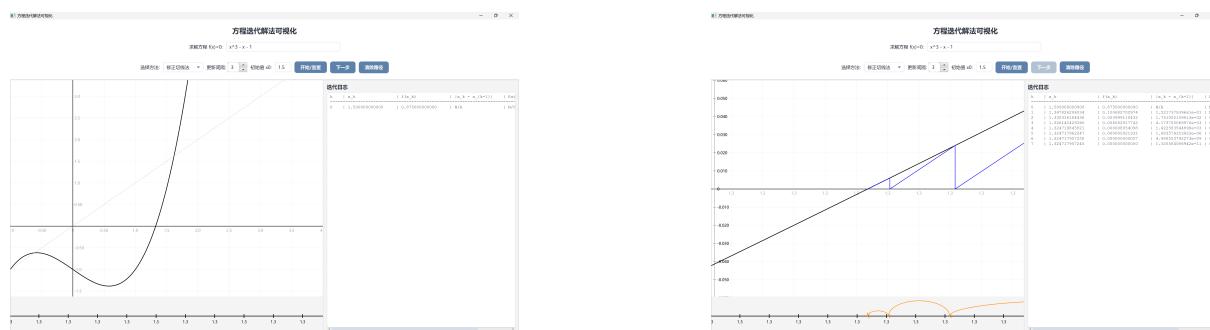


图 5 修正切线法软件运行截图

2.2.5 牛顿下山法

基本思想

牛顿法非常依赖合适的初值，一旦选择不当就会不收敛，这也导致了其有局部收敛的特征。牛顿下山法通过引入一个下山因子 λ 来扩大收敛范围。下山法强制要求每次迭代都必须满足“下山”条件——即 $|f(x_{k+1})| < |f(x_k)|$ 。如果标准的牛顿步长 ($\lambda = 1$) 不满足此条件，就缩小步长（将 λ 减半）再尝试，直到满足下山条件为止。

数学公式

迭代公式为：

$$x_{k+1} = x_k - \lambda \frac{f(x_k)}{f'(x_k)} \quad (12)$$

算法步骤

1. 选取初始值 x_0 和误差 ϵ 。
2. 计算牛顿步 $d_k = \frac{f(x_k)}{f'(x_k)}$ 。
3. 初始化下山因子 $\lambda = 1$ 。
4. 计算候选点 $x_{next} = x_k - \lambda d_k$ 。
5. 判断是否满足下山条件 $|f(x_{next})| < |f(x_k)|$ 。
6. 若不满足，则 $\lambda = \lambda/2$ ，返回第 4 步。若 λ 过小则认为方法失败。
7. 若满足，则接受该点，令 $x_{k+1} = x_{next}$ 。
8. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ ，若满足则停止迭代否则返回第 2 步。

收敛性分析

下山法相比牛顿法能够更大范围的收敛。

优缺点

- 优点：相比牛顿法扩大了收敛域，对初值选择不那么敏感。
- 缺点：算法更复杂，增加了单次迭代的计算量。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 时的软件运行截图。

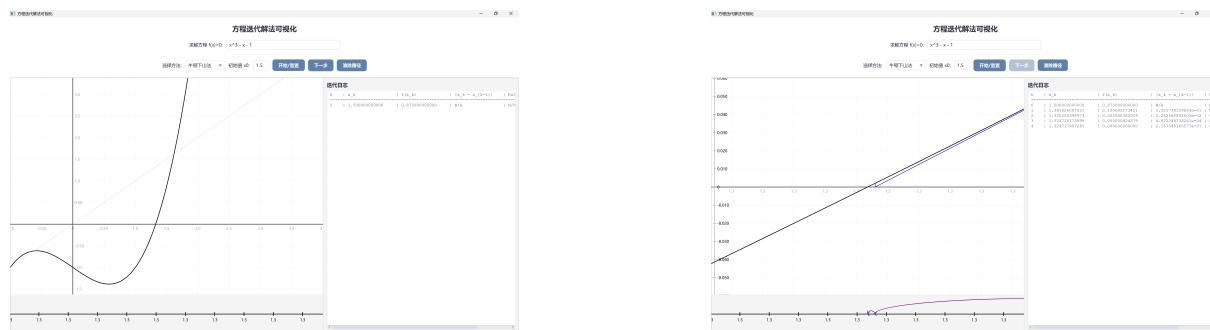


图 6 牛顿下山法软件运行截图

2.2.6 单点弦截法

基本思想

该方法是牛顿法的另一种近似，它使用一个固定点 $(x_0, f(x_0))$ 和当前迭代点 $(x_k, f(x_k))$ 连接形成的割线与 x 轴的交点，来作为下一个近似根 x_{k+1} 。这等价于在牛顿法公式中用差商 $\frac{f(x_k) - f(x_0)}{x_k - x_0}$ 来近似导数 $f'(x_k)$ 。

数学公式

迭代公式为：

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_0)}{f(x_k) - f(x_0)} \quad (13)$$

算法步骤

1. 选取一个固定点 x_0 和一个初始迭代点 x_1 ，以及误差 ϵ 。
2. 计算并存储 $f(x_0)$ 。
3. 对于 $k = 1, 2, \dots$ ，根据上述公式计算下一个近似值 x_{k+1} 。
4. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。
5. 若不满足，则令 $x_k = x_{k+1}$ ，返回第 3 步。

收敛性分析

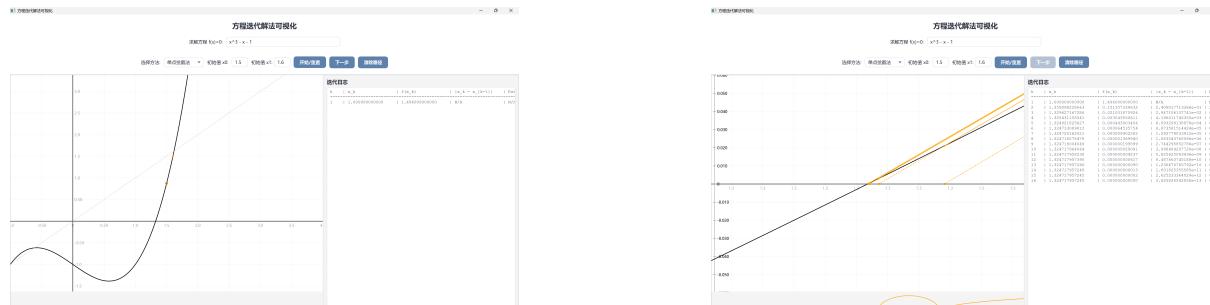
该方法线性收敛，收敛速度通常慢于牛顿法。

优缺点

- 优点：无需计算导数。
- 缺点：
 - 收敛速度为一阶，较慢。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 时的软件运行截图（详细代码见附录 A）。



(a) 单点弦截法软件运行截图 1

(b) 单点弦截法软件运行截图 2

图 7 单点弦截法软件运行截图

2.2.7 双点弦截法

基本思想

双点弦截法是单点弦截法的一个改进，该方法使用最近的两个迭代点 $(x_{k-1}, f(x_{k-1}))$ 和 $(x_k, f(x_k))$ 来构造一条割线，并用该割线与 x 轴的交点作为新的近似根 x_{k+1} 。

数学公式

迭代公式为：

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})} \quad (14)$$

算法步骤

1. 选取两个初始近似值 x_0, x_1 和允许误差 ϵ 。
2. 对于 $k = 1, 2, \dots$ ，根据迭代公式计算 x_{k+1} 。
3. 判断是否满足收敛条件 $|x_{k+1} - x_k| < \epsilon$ 。
4. 若不满足，则更新迭代点：令 $x_{k-1} = x_k, x_k = x_{k+1}$ ，然后返回第 2 步。

收敛性分析

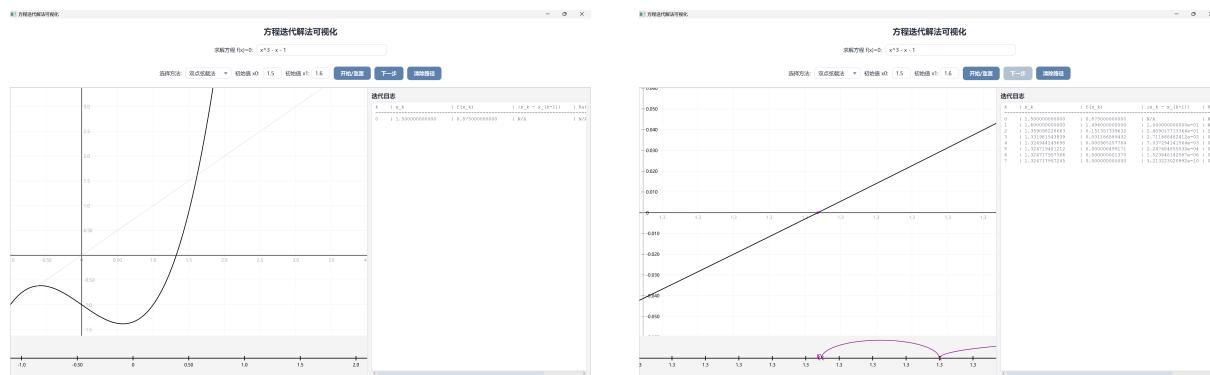
该方法具有超线性收敛性，收敛阶约为 1.618。其收敛速度快于线性收敛方法，但慢于牛顿法的二阶收敛。

优缺点

- 优点：无需计算导数，同时保持了较快的收敛速度。
- 缺点：与牛顿法一样是局部收敛的，不能保证对任意初值都收敛。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 和迭代函数 $g(x) = \sqrt[3]{x + 1}$ 时的软件运行截图（详细代码见附录 A）。



(a) 双点弦截法软件运行截图 1

(b) 双点弦截法软件运行截图 2

图 8 双点弦截法软件运行截图

2.2.8 艾特肯迭代法

基本思想

艾特肯法可以看作是双点弦截法的一种变形，艾特肯法利用由不动点迭代 $x_{k+1} = g(x_k)$ 产生的一系列点来执行双点弦截法。双点弦截法的迭代公式是利用两个点 $(x_k, f(x_k))$ 和 $(x_{k+1}, f(x_{k+1}))$ 构造割线来求根：

$$x_{new} = x_{k+1} - \frac{f(x_{k+1})(x_{k+1} - x_k)}{f(x_{k+1}) - f(x_k)} \quad (15)$$

将 x_k 、 x_{k+1} 两个点代入函数 $f(x) = x - g(x)$ ，可得：

$$\begin{aligned} f(x_k) &= x_k - g(x_k) = x_k - x_{k+1} \\ f(x_{k+1}) &= x_{k+1} - g(x_{k+1}) = x_{k+1} - x_{k+2} \end{aligned} \quad (16)$$

现在，我们将上述结果代回双点弦截法的公式中，求出的 x_{new} 即为加速后的 new 迭代值 \hat{x}_k ：

$$\hat{x}_k = x_{k+1} - \frac{(x_{k+1} - x_{k+2})(x_{k+1} - x_k)}{(x_{k+1} - x_{k+2}) - (x_k - x_{k+1})} \quad (17)$$

对上式分母进行化简： $(x_{k+1} - x_{k+2}) - (x_k - x_{k+1}) = -(x_{k+2} - 2x_{k+1} + x_k)$ 。随后我们对整个表达式进行代数变换：

$$\begin{aligned} \hat{x}_k &= \frac{x_{k+1}((2x_{k+1} - x_k - x_{k+2}) - (x_{k+1} - x_{k+2})) - x_k(x_{k+1} - x_{k+2})}{2x_{k+1} - x_k - x_{k+2}} \\ &= \frac{x_k x_{k+2} - x_{k+1}^2}{x_{k+2} - 2x_{k+1} + x_k} \\ &= \frac{x_k(x_{k+2} - 2x_{k+1} + x_k) - (x_k^2 - 2x_k x_{k+1} + x_{k+1}^2)}{x_{k+2} - 2x_{k+1} + x_k} \\ &= x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k} \end{aligned} \quad (18)$$

最终得到的这个公式就是艾特肯加速法的标准形式。它利用基础迭代产生的连续三点 (x_k, x_{k+1}, x_{k+2}) 来构造一个收敛更快的迭代值。

数学公式

给定一个由基础迭代 $x_{i+1} = g(x_i)$ 产生的序列，艾特肯加速序列 $\{\hat{x}_k\}$ 的计算公式为：

$$\hat{x}_k = x_k - \frac{(x_{k+1} - x_k)^2}{x_{k+2} - 2x_{k+1} + x_k} \quad (19)$$

算法步骤

- 选取初始值 x_0 和误差 ϵ 。

2. 使用基础迭代公式 $x_{i+1} = g(x_i)$ 计算两步: $x_1 = g(x_0)$, $x_2 = g(x_1)$ 。
3. 使用艾特肯公式计算加速后的值 \hat{x}_0 。
4. 判断是否满足收敛条件 $|\hat{x}_0 - x_0| < \epsilon$ 。
5. 若不满足, 则令 $x_0 = \hat{x}_0$, 返回第 2 步继续迭代。

收敛性分析

如果原始的不动点迭代法是线性收敛的, 那么经过艾特肯法加速后形成的序列通常具有二阶收敛性。

优缺点

- 优点: 显著提高线性收敛迭代法的收敛速度。
- 缺点:
 - 每次迭代的计算复杂性增加。
 - 分母 $x_{k+2} - 2x_{k+1} + x_k$ 接近于零时会出现数值不稳定问题。

软件示例

下图展示了我们选择原始函数 $f(x) = x^3 - x - 1$ 和迭代函数 $g(x) = \sqrt[3]{x+1}$ 时的软件运行截图 (详细代码见附录 A)。

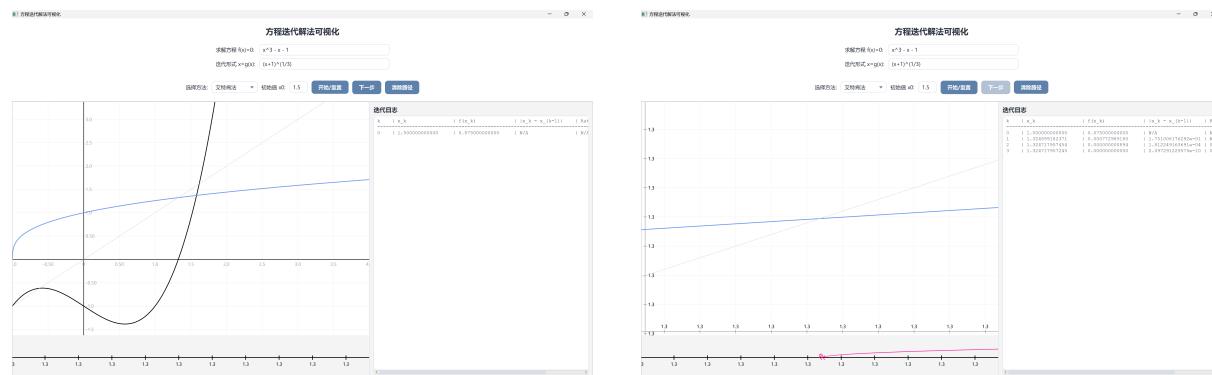


图 9 艾特肯法软件运行截图

3 线性方程组的直接解法

3.1 消元法综述

3.1.1 消元法介绍

消元法是将方程组中的一方程的未知数用含有另一未知数的代数式表示，并将其带入到另一方程中，这就消去了一未知数，得到一解；或将方程组中的一方程倍乘某个常数加到另外一方程中去，也可达到消去一未知数的目的。

更本质的来看，消元法通过一系列的初等行变换，将线性方程组的增广矩阵逐步转化为一个更容易求解的等价矩阵，然后通过回代求解出未知数。

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \xrightarrow{\text{高斯消元}} \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix} \quad (20)$$

3.1.2 消元法核心理论

- 两方程互换，解不变
- 一方程乘以非零数 k 解不变
- 一方程乘以数 k 加上另一方程，解不变

3.1.3 消元法步骤

- 消元
 - 将线性方程组写成增广矩阵的形式。
 - 通过一系列的初等行变换（在上文中提及，教材中采用1作为每一行需要除的分母，以此完成行变换），从第一列开始，逐列将主元下方的元素变为零。
 - 重复执行第二步直到矩阵变为上三角矩阵。
- 回代
 - 从阶梯形矩阵的最后一行开始（该行可以直接求解）。将求得的解代入上一行，解出另一个未知数。
 - 依次向上回代，直到求出所有未知数的解。

3.1.4 优缺点

- 优点：对于任何有唯一解或有无穷多解的线性方程组，理论上总能求得其精确解。
- 缺点：
 - 计算量大， $O(n^3)$ 级别。
 - 舍入误差会出现累积并影响精度。

3.2 详细消元法

3.2.1 高斯消元法

介绍

高斯消元法在每次消元的时候本行不做处理，即当前行保持原始状态，不会同下面即将要说的克劳特消元法一样进行对角元归一。

矩阵变换公式

$$[A|\mathbf{b}] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array} \right] \xrightarrow{\text{正向消元}} [U|\mathbf{z}] = \left[\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & z_1 \\ 0 & u_{22} & u_{23} & z_2 \\ 0 & 0 & u_{33} & z_3 \end{array} \right] \quad (21)$$

收敛性

- 在没有舍入误差且对角元不为 0 的理想情况下，高斯消元法总能通过有限步运算得到精确解。

- 对角线元素为 0 或绝对值很小时，会导致计算产生巨大的舍入误差。

程序运行截图

下图为程序在利用高斯消元法求解线性方程组时的截图，详细代码见附录 B。

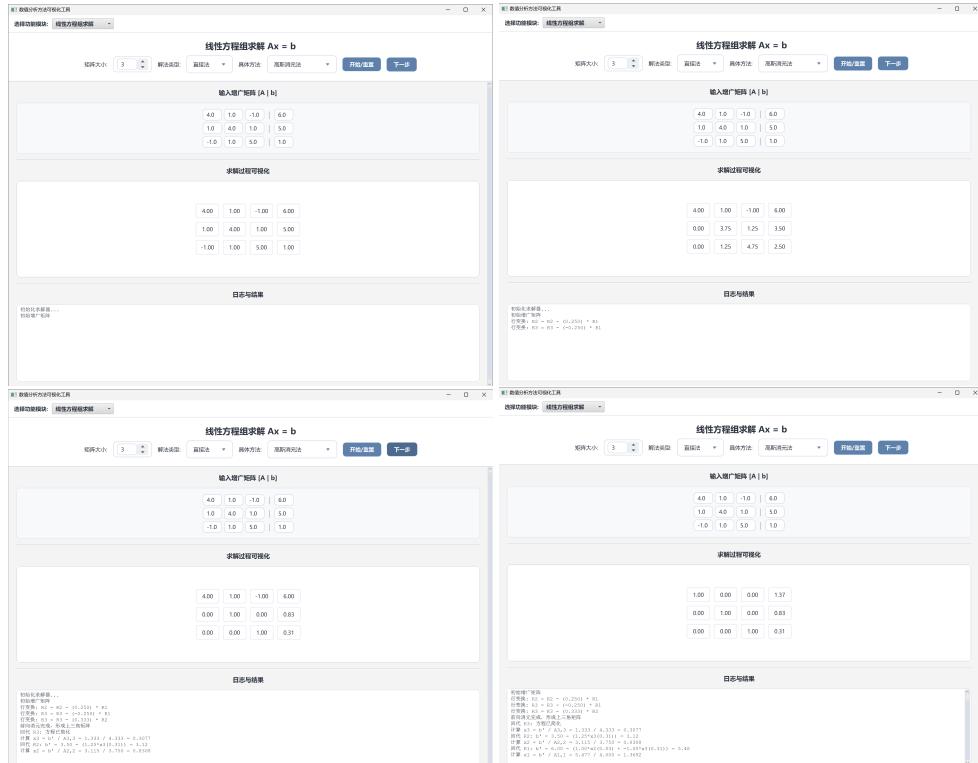


图 10 高斯消元法程序运行截图

3.2.2 克劳特消元法

介绍

克劳特消元法在每次进行消元的时候会首先将本行的对角元素进行除法，使得对角元素值为1。这样在得到上三角矩阵后方便进行回代操作。

矩阵变换公式

$$[A|\mathbf{b}] = \left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array} \right] \xrightarrow{\text{正向消元}} [U|\mathbf{z}] = \left[\begin{array}{ccc|c} 1 & u_{12} & u_{13} & z_1 \\ 0 & 1 & u_{23} & z_2 \\ 0 & 0 & 1 & z_3 \end{array} \right] \quad (22)$$

收敛性

- 与高斯消元法类似，克劳特法同样存在数值不稳定的问题。

程序运行截图

下图为程序在利用克劳特消元法求解线性方程组时的截图，详细代码见附录 B。

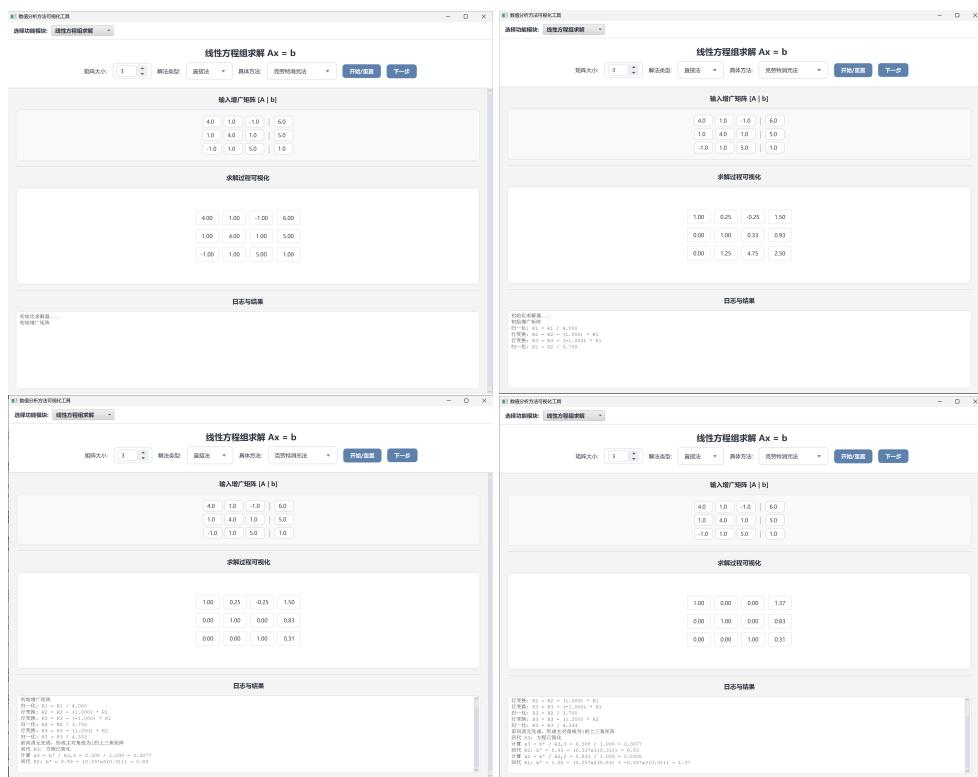


图 11 克劳特消元法程序运行截图

3.2.3 高斯列主元素消元法

介绍

该方法总体思路与高斯消元法一致，只是在对每行处理的时候都会遍历该行对角元素所在的列并把找到的最大元素所在的行与当前行进行交换，保证当前行对角元素最大。

矩阵变换公式

$$Ax = b \xrightarrow{\text{行交换}} PAx = P\mathbf{b} \xrightarrow{\text{消元}} [PU|P\mathbf{z}] = \left[\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & z_1 \\ 0 & u_{22} & u_{23} & z_2 \\ 0 & 0 & u_{33} & z_3 \end{array} \right] \quad (23)$$

收敛性

- 通过选取较大的主元，可以有效避免除以一个很小的数，从而抑制舍入误差的增长。

- 只要矩阵是非奇异的，列主元法即可求出结果。
- 这是目前求解小型稠密线性方程组最常用、最稳健的方法。

程序运行截图

下图为程序在利用列主元素消元法求解线性方程组时的截图，详细代码见附录 B。

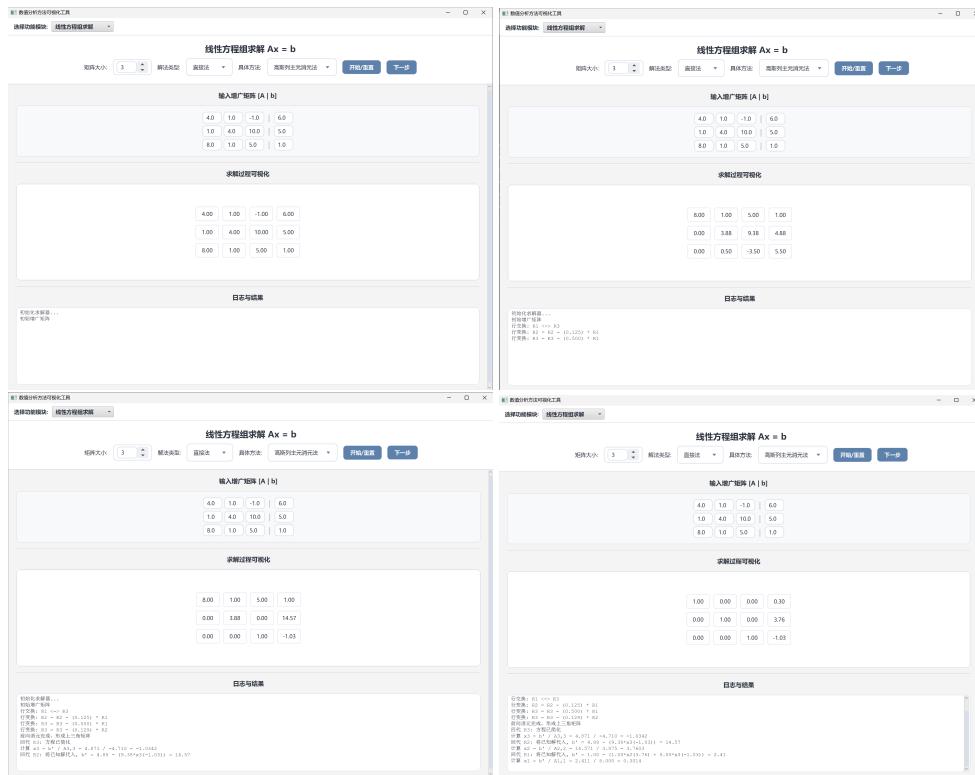


图 12 高斯列主元消元法程序运行截图

3.2.4 高斯全主元素消元法

介绍

该方法总体思路与高斯消元法一致，只是在对每行处理的时候都会遍历未处理的整个矩阵元素并把找到的最大元素所在的位置通过行交换与列交换换到当前处理的对角元处，保证当前行对角元素全局最大。

矩阵变换公式

$$Ax = b \xrightarrow{\text{行列交换}} PQAx = Pb \xrightarrow{\text{消元}} [PQU|P\mathbf{z}] = \left[\begin{array}{ccc|c} u_{11} & u_{12} & u_{13} & z_1 \\ 0 & u_{22} & u_{23} & z_2 \\ 0 & 0 & u_{33} & z_3 \end{array} \right] \quad (24)$$

收敛性

- 在所有高斯消元法的变种中，全主元素法对舍入误差的控制效果最好。
- 因为全主元素法需要多次寻找全局最大值，其搜索开销远大于列主元法。

程序运行截图

下图为程序在利用全主元素消元法求解线性方程组时的截图，详细代码见附录 B。

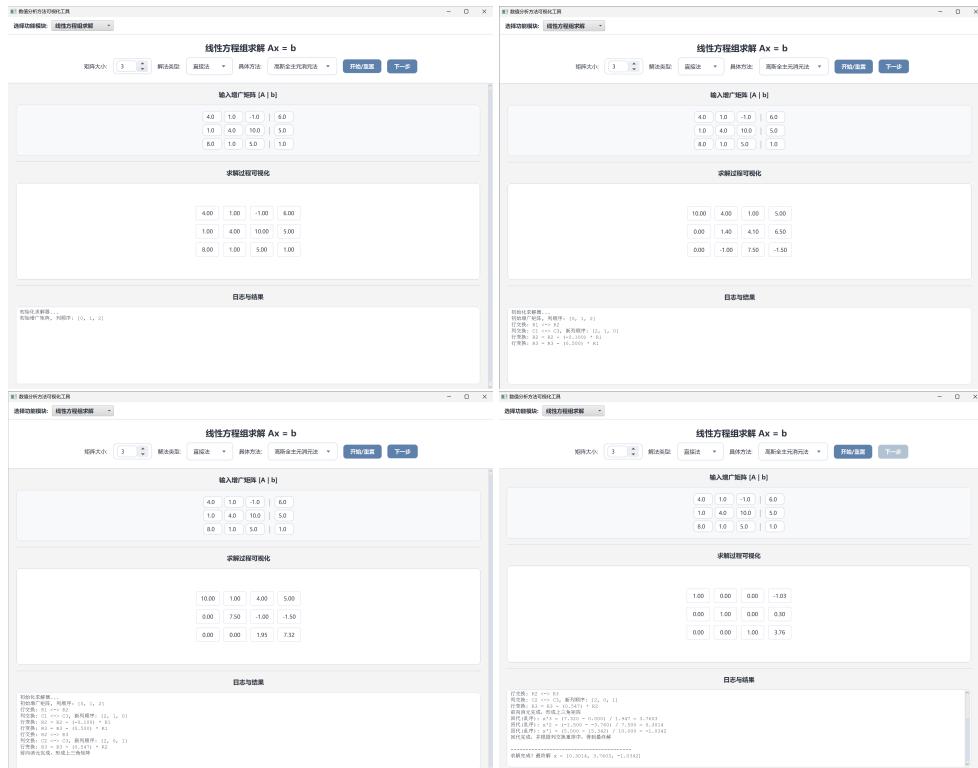


图 13 高斯全主元素消元法程序运行截图

4 线性方程组的迭代解法

4.1 线性方程组的迭代法综述

4.1.1 线性方程组的迭代法介绍

迭代法从一个初始的近似解出发，通过迭代公式反复计算，逐步逼近方程组精确解。这里面确保收敛的判定依据依旧是上文在方程的迭代解法中提到的压缩映射，只要一个矩阵的谱半径小于 1 就可以保证方法收敛。这里的谱半径衡量的是矩阵映射变换完成后各个维度的变化最大值，只要该值小于 1 就表明该矩阵对于所有维度的映射都是压缩的，也就说明空间内的所有点都在向一个点靠近。

$$\mathbf{x}^{(k)} = \begin{pmatrix} x_1^{(k)} \\ x_2^{(k)} \\ \vdots \\ x_n^{(k)} \end{pmatrix} \xrightarrow{\text{迭代公式 } \mathbf{x}^{(k+1)} = \mathbf{B}\mathbf{x}^{(k)} + \mathbf{f}} \mathbf{x}^{(k+1)} = \begin{pmatrix} x_1^{(k+1)} \\ x_2^{(k+1)} \\ \vdots \\ x_n^{(k+1)} \end{pmatrix} \quad (25)$$

4.1.2 消元法步骤

- 构造迭代格式：将原方程组 $Ax = b$ 变形为一个等价的形式 $x = Bx + f$ ，其中 B 称为迭代矩阵。
- 选取初始向量：选择一个向量并从此作为迭代的起点。
- 进行迭代：根据迭代公式 $x(k+1) = Bx(k) + f$ ，从 $x(0)$ 开始，依次计算 $x(1), x(2), x(3), \dots$
- 判断收敛：每次迭代后检查是否满足收敛条件并选择停机或者继续执行第三步。

4.1.3 优缺点

- 优点：
 - 计算量小，迭代的计算量远小于直接法。
 - 舍入误差不会累积。
- 缺点：
 - 不保证收敛，收敛性与迭代矩阵 B 的性质密切相关。
 - 得到的是近似解，而非理论上的精确解。

4.2 详细迭代方法

4.2.1 雅可比迭代法

介绍

雅可比迭代法是一种基本的迭代方法。它的思想很类似前文讲到的方程的迭代解法，在每次计算新的向量 x_{k+1} 过程中，它会利用上一次的迭代结果 x_k ，并将这个向量输入矩阵进行映射。由于各个分量的计算是相互独立的，因此该方法非常适合并行计算。

矩阵迭代公式

- 矩阵形式:

$$\mathbf{x}^{(k+1)} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} + \mathbf{D}^{-1}\mathbf{b} \quad (26)$$

- 分量形式:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \dots, n \quad (27)$$

收敛性

- 充要条件: 迭代法收敛的充要条件是迭代矩阵的谱半径小于 1。对于雅可比法，即 $\rho(\mathbf{B}_J) < 1$ ，其中 $\mathbf{B}_J = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ 。
- 充分条件: 如果系数矩阵 \mathbf{A} 是严格对角占优矩阵，则雅可比迭代法必定收敛。
- 充分条件: 如果系数矩阵 \mathbf{A} 是对角占优矩阵且不可约，则雅可比迭代法必定收敛。
- 充分条件: 如果迭代矩阵 \mathbf{B}_J 的范数小于 1 则雅可比迭代法必定收敛。

程序运行截图

下图为程序在利用雅克比迭代法求解线性方程组时的截图，

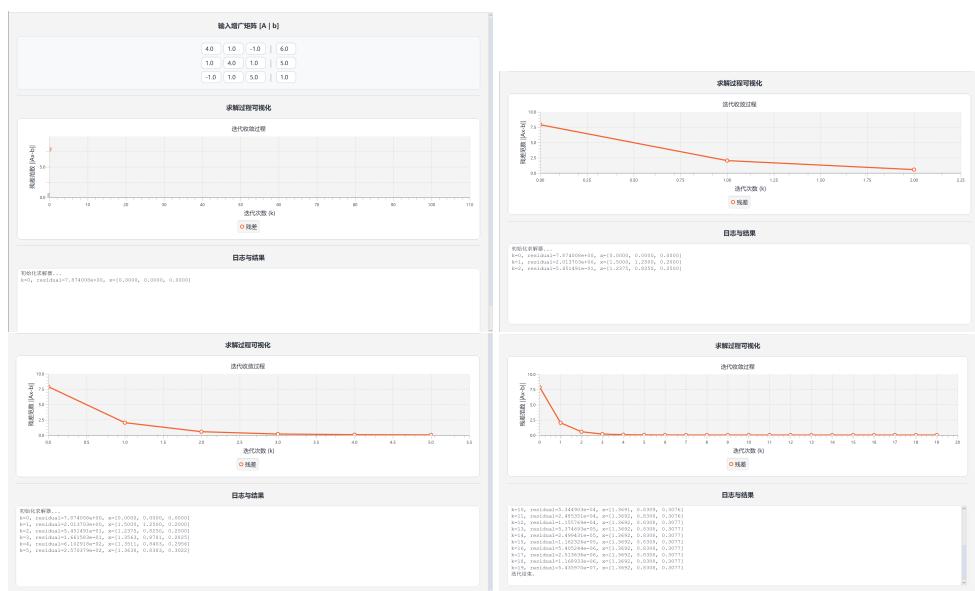


图 14 雅可比迭代法程序运行截图

4.2.2 高斯-赛德尔迭代法

介绍

高斯-赛德尔法是雅可比法的一种改进型。它在计算第 $k+1$ 次迭代的第 i 个分量 $x_i^{(k+1)}$ 时，会立即使用在同一次迭代中已经计算出的新分量 $x_1^{(k+1)}, \dots, x_{i-1}^{(k+1)}$ 。这种策略使得它通常比雅可比法收敛得更快，但也因为计算存在依赖关系所以不适合并行处理。

矩阵迭代公式

- 矩阵形式:

$$\mathbf{x}^{(k+1)} = -(\mathbf{D} + \mathbf{L})^{-1} \mathbf{U} \mathbf{x}^{(k)} + (\mathbf{D} + \mathbf{L})^{-1} \mathbf{b} \quad (28)$$

- 分量形式:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \dots, n \quad (29)$$

收敛性

- 充要条件: 迭代法收敛的充要条件是迭代矩阵的谱半径小于 1。对于雅可比法，即 $\rho(\mathbf{B}_{GS}) < 1$ ，其中 $\mathbf{B}_{GS} = -(\mathbf{D} + \mathbf{L})^{-1} \mathbf{U}$ 。

- 充分条件: 如果系数矩阵 \mathbf{A} 是严格对角占优矩阵，则高斯-赛德尔迭代法必定收敛。
- 充分条件: 如果系数矩阵 \mathbf{A} 是对角占优矩阵且不可约，高斯-赛德尔迭代法必定收敛。
- 充分条件: 如果迭代矩阵 \mathbf{B}_{GS} 的范数小于 1 高斯-赛德尔迭代法必定收敛。
- 充分条件: 如果系数矩阵 \mathbf{A} 是对称正定矩阵，则高斯-赛德尔迭代法必定收敛。

程序运行截图

下图为程序利用高斯-赛德尔迭代法求解线性方程组时的截图，详细代码见附录 C。

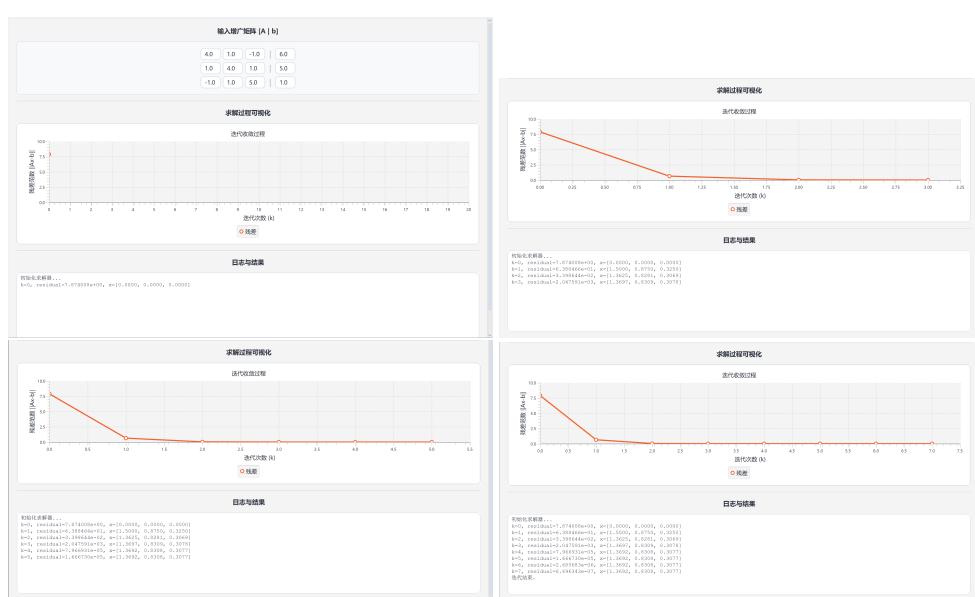


图 15 高斯-赛德尔迭代法程序运行截图

4.2.3 松弛迭代法

介绍

松弛迭代法是高斯-赛德尔法的一种改进。它在计算出高斯-赛德尔迭代值后将其与上一次的旧值进行加权平均作为新值。

矩阵迭代公式

- 分量形式: 设 $\tilde{x}_i^{(k+1)}$ 为高斯-赛德尔法计算出的中间值, 则

$$\begin{aligned}\tilde{x}_i^{(k+1)} &= \frac{1}{a_{ii}} \left(b_i - \sum_{j < i} a_{ij} x_j^{(k+1)} - \sum_{j > i} a_{ij} x_j^{(k)} \right) \\ x_i^{(k+1)} &= (1 - \omega)x_i^{(k)} + \omega\tilde{x}_i^{(k+1)}\end{aligned}\quad (30)$$

收敛性

- 必要条件: SOR 方法收敛的一个必要条件是松弛因子 ω 必须在 $(0, 2)$ 的范围内。

- 分类:

- 当 $0 < \omega < 1$ 时, 称为低松弛法。
- 当 $\omega = 1$ 时, SOR 方法为高斯-赛德尔法。
- 当 $1 < \omega < 2$ 时, 称为超松弛法, 用于加速收敛。

程序运行截图

下图为程序在利用松弛迭代法求解线性方程组时的截图, 详细代码见附录 C。

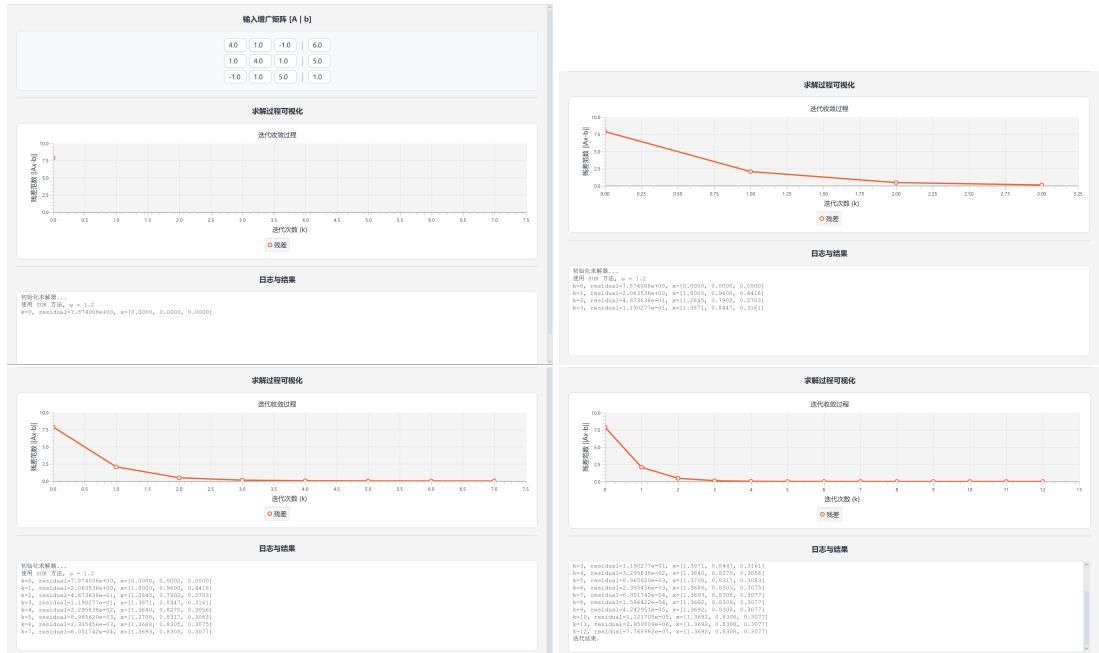
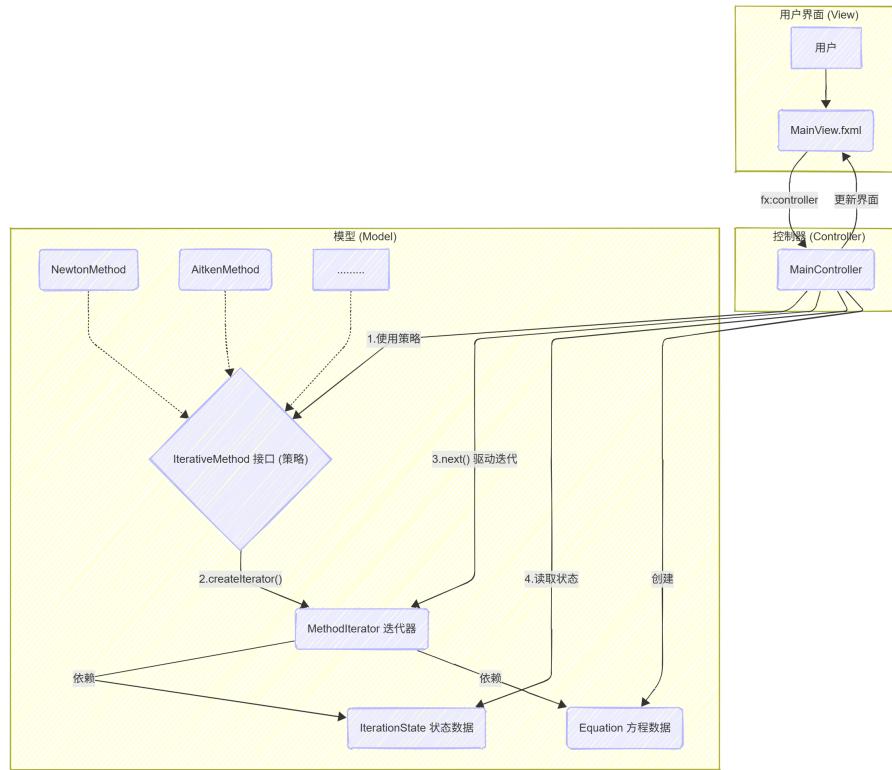


图 16 松弛迭代法程序运行截图

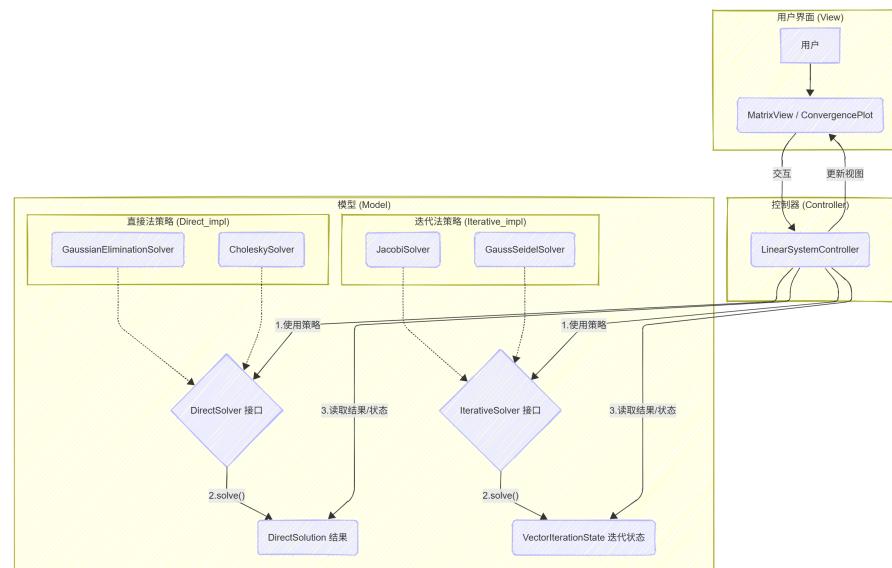
5 java 软件实现

5.1 总体设计

我的软件采用了模型-视图-控制器（MVC）的设计模式，下方为我的软件第一部分结构图，这种模式将主要的功能解耦，在日后较易进行进一步的扩展。



(a) 非线性方程求根模块架构



(b) 线性方程组求解模块架构

图 17 软件核心模块的 MVC 架构图

5.1.1 线性方程组求解模块

- 模型:

- `Direct_impl` 包: 封装了直接解法, 如高斯消元法、带主元选择的高斯消元法、克劳特法。

- `Iterative_impl` 包: 封装了迭代解法, 如雅可比法、高斯-赛德尔法和松弛法。

- `LinearSystemSolver` 作为统一的策略接口, `DirectSolver` 和 `IterativeSolver` 是其子接口, 分别定义了直接法和迭代法的行为。

- `MatrixState`, `VectorIterationState` 等类用于封装计算的中间状态和最终解。

- 视图:

- `MatrixView`: 用于输入和显示系数矩阵 A 和向量 b。

- `ConvergencePlot`: 用于可视化展示迭代法的收敛过程。

- 控制器:

- `LinearSystemController`: 响应用户的算法选择和“求解”命令, 创建对应的求解器策略实例, 执行求解过程, 并从返回的状态对象中提取数据更新到 `MatrixView` 和 `ConvergencePlot` 中。

5.1.2 非线性方程求根模块

- 模型:

- `impl` 包: 包含了所有非线性求根算法的策略实现, 如牛顿法、简化牛顿法、各种割线法、简单迭代法和艾特肯加速法。

- `IterativeMethod` 是所有这些算法共同实现的策略接口。

- `MethodIterator` 是迭代器实现, 它持有一个 `IterativeMethod` 策略实例, 并管理 `IterationState`。

- `Equation` 类用于封装和解析用户输入的数学表达式。

- 视图:

- `OneDimPlot` / `TwoDimPlot`: 用于绘制函数图像, 并动态地在图上标出每一次迭代的点位, 使用户可以直观地看到逼近过程。

- 控制器:

- `MainController`: 根据用户选择的算法策略, 创建 `MethodIterator`。通过控制 `Iterator` 的 `next()` 方法执行单步或连续迭代, 每次迭代后从 `IterationState` 读取数据, 并命令 `View` 刷新绘图。

5.2 核心功能实现

软件的核心计算功能主要围绕两大模块展开：非线性方程求根和线性方程组求解。两者都采用了策略模式来封装具体算法，但其内部的执行和控制流程根据问题的特性有所不同。

5.2.1 非线性方程求根实现

控制器驱动的迭代流程如下：

- 初始化：当用户点击“开始”时，控制器首先根据用户在下拉菜单中的选择，从 `IterativeMethod` 策略数组中获取对应的算法对象。随后，它调用该策略对象的 `createIterator()` 方法，传入 `Equation` 实例和初始值，从而创建一个与具体算法绑定的 `MethodIterator` 迭代器实例。这个过程完美体现了策略模式和工厂方法的思想。
- 单步执行：每次用户点击“下一步”时，控制器只需调用当前迭代器实例的 `next()` 方法。该方法会执行一次完整的迭代计算，并返回一个封装了所有中间数据的 `IterationState` 对象。
- 状态更新与可视化：控制器接收到 `IterationState` 对象后，将其存入历史记录列表，并调用 `View` 层的相应方法（如更新日志、重绘函数图像），从而在界面上实时反馈计算的进展。当迭代器的 `hasNext()` 方法返回 `false` 时，迭代过程终止。

5.2.2 线性方程组求解实现

该模块区分了直接法和迭代法两种求解思路。

1. 双策略体系：控制器 (`LinearSystemController`) 内部维护了两个独立的策略：一个用于直接法 (`DirectSolver`)，另一个用于迭代法 (`IterativeSolver`)。控制器根据用户的选择来决定使用哪一套策略。
2. 直接法：当用户选择一种直接法时，其实现流程如下：
 - 预处理计算：控制器调用所选 `DirectSolver` 策略的 `solve()` 方法。该方法会一次性地完成所有计算步骤，然后返回一个 `DirectSolution` 对象。
 - 历史状态封装：这个 `DirectSolution` 对象内部不仅包含了最终解，还存储了一个包含所有中间步骤状态 (`MatrixState`) 的历史记录列表。
 - 可视化：用户的“下一步”操作，实际上是从这个预先计算好的历史记录列表中逐个取出 `MatrixState`，并交由 `MatrixView` 进行渲染。
3. 迭代法：当用户选择一种迭代法时，流程与非线性模块类似，流程如下：

- 创建迭代器: 控制器调用所选 IterativeSolver 策略的 `createIterator()` 方法来获取一个迭代器。特别地, 对于 SOR 方法, 控制器会额外读取用户输入的 ω 值并传入。
- 实时单步计算: 用户的“下一步”操作会调用迭代器的 `next()` 方法, 实时执行一次迭代计算, 并返回一个包含当前步结果 (`VectorIterationState`) 的状态对象。
- 可视化: 控制器将新的状态对象传递给 `ConvergencePlot`, 后者会在收敛图中增量地添加一个新的数据点, 实时描绘解的收敛轨迹。

5.2.3 函数字符串识别与函数求值

相关的代码我附在附录 D 处, 下面介绍我所用到的算法与逻辑。

1. 词法分析

这里我们在所有的运算符、括号和逗号前后添加空格, 然后基于空格进行分割, 得到初步的符号流, 例如 ["-", "x", "*", "sin", "(", "pi", "*", ")"]。

2. 调度场算法

这里我们利用调度长算法将中缀表达式的符号流转换为后缀表达式, 即逆波兰表达式 (**RPN**)。

我们严格遵守调度场算法的相关要求, 设置辅助栈来协助转换字符串, 高优先级先压栈, 低优先级则出栈加入输出队列 (实际情况更加复杂, 此处我自己实现了普通的调度场算法, 更多的细致内容如特殊函数、左右结合的处理由 AI 辅助完成)。

我们

3. 求值

这里我们使用一个栈来进行函数求值。首先我们遍历 RPN 列表: 遇到数字、常数或变量值, 则压入栈中; 遇到运算符或函数, 则从栈中弹出所需数量的操作数, 进行计算, 并将结果压回栈中。遍历结束后, 栈中唯一剩下的元素即为表达式的最终计算结果。

5.3 软件界面介绍

下图展示了软件中方程迭代法子功能的用户界面。可以看到整个界面分为三部分：上侧为用户输入区，在这里用户输入自己想求解的方程、选择想使用的迭代方法，并重置或单步运行迭代；左下方为可视化区域，在这里用户可以看到二维平面和一维数轴上迭代点的变化；右下侧为迭代日志，在这里用户可以看到每一次迭代的详细信息。

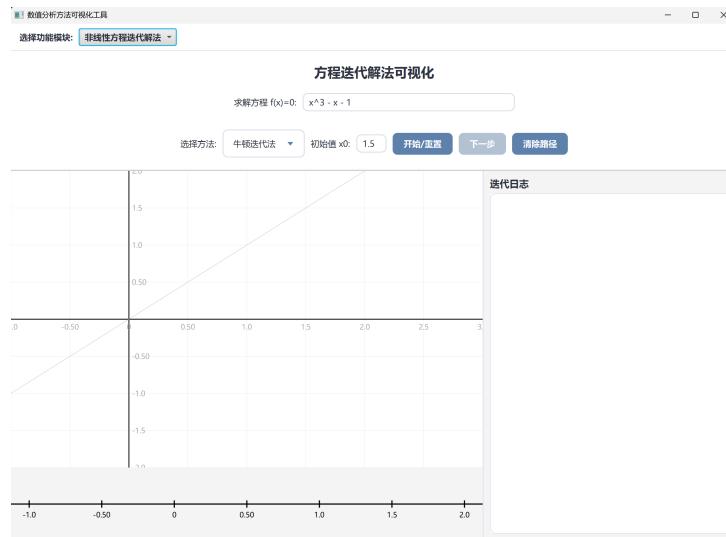


图 18 方程的迭代法用户界面

下图展示了线性方程组直接法软件的用户界面。可以看到这个页面被分成了上中下三个部分，上部为矩阵输入处，用于用户输入要进行求解的增广矩阵；中间部分为程序求解过程的可视化处，在这里用户可以看到矩阵是如何一步步变化为最终答案的；最下侧展现了每一次迭代过程的详细操作和记录。

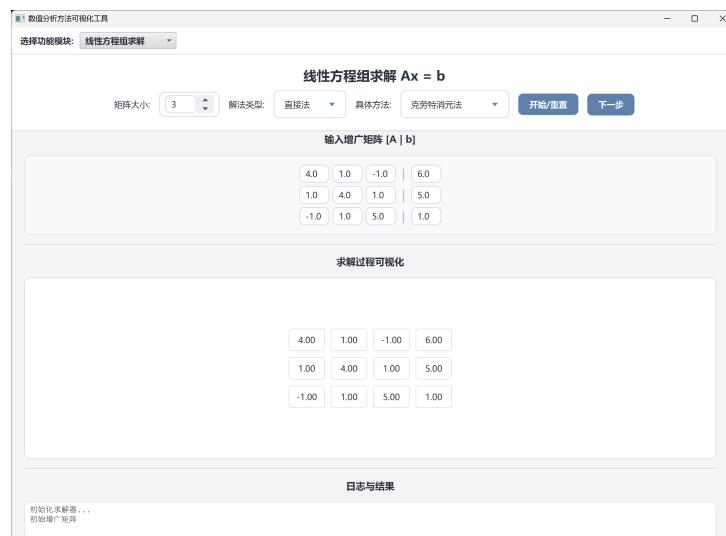


图 19 线性方程组直接法软件用户界面

下图展示了线性方程组迭代法软件的用户界面。同样的这个图被分为了三个部分，在上侧依旧是矩阵输入处用于输入要求解的矩阵；中间展示每一次迭代产生的向量与答案向量之间范数的大小用于衡量误差大小；最下侧记录迭代过程中的相关数据。



图 20 线性方程组迭代法软件用户界面

6 总结

6.1 工作总结

总的来说本报告较为详细的分析与展示了不同算法各自的数学原理和收敛性分析。就迭代法本质来看，所有相关迭代方法均是由最初的普通迭代法改造而来（无论是方程还是方程组），其本质无外乎修改映射方式使映射的收缩效果更好来加快收敛速度。

此外，本报告还提供了一款基于 java 开发的可视化软件，并以此来详细的展示多种方法不同的运行模式。本软件采用了 MVC 布局，便于日后的拓展任务。

通过将抽象的公式转化为直观的动画和图，本报告连接了编程实践和理论学习，一定程度上减轻了数值分析的学习任务，达到了预期的目标。

最后，本人建议您可以从 GitHub(<https://github.com/twx145/Numerical-Analysis/releases/tag/v2.5.0>) 上下载最新软件。

6.2 展望

为了进一步提升本软件的广度和深度，未来可以从以下几部分展开研究

- 扩充算法库：集成更多高级算法，如处理多项式根的拉盖尔法。
- 增强交互与智能：开发如拖拽初始点、用户缩放图像等功能，使探索分析过程更加便捷、智能。

参考文献

- [1] KNUTH D E. The TeXbook[M]. Addison-Wesley Professional, 1984.
- [2] AMAT S, BUSQUIER S. Advances in Iterative Methods for Nonlinear Equations[M]. Springer, Cham. DOI: [10.1007/978-3-319-39228-8](https://doi.org/10.1007/978-3-319-39228-8).

致谢

衷心感谢孙新老师在《数值分析》课程中的指导与帮助！报告不足之处，恳请指正。

附录

A 方程的迭代解法完整代码

普通迭代法

```

1   if (k == 0) {
2       double fx0 = equation.getF().apply(x_curr);
3       IterationState initialState = IterationState.initial(x_curr, fx0);
4       x_prev = x_curr;
5       k++;
6       return initialState;
7   }
8   // --- 执行一次普通迭代 ---
9   x_curr = equation.getG().apply(x_prev);
10

```

Listing 1 普通迭代法单步迭代核心逻辑

牛顿迭代法

```

1   if (k == 0) {
2       double fx0 = equation.getF().apply(x_curr);
3       IterationState initialState = IterationState.initial(x_curr, fx0);
4       x_prev = x_curr;
5       k++;
6       return initialState;
7   }
8   // --- 执行一次牛顿法迭代 ---
9   double fx = equation.getF().apply(x_prev);
10  double dfx = equation.getDf().apply(x_prev);
11
12  if (Math.abs(dfx) < 1e-12) {
13      x_curr = Double.NaN;
14  } else {
15      x_curr = x_prev - fx / dfx;
16  }
17

```

Listing 2 牛顿迭代法单步迭代核心逻辑

简化切线法

```

1  if (k == 0) {
2      double fx0 = equation.getF().apply(x_curr);
3      IterationState initialState = IterationState.initial(x_curr, fx0);
4      x_prev = x_curr;
5      k++;
6      return initialState;
7  }
8  // --- 执行一次简化切线法迭代 ---
9  x_curr = x_prev - equation.getF().apply(x_prev) / dfx0;
10

```

Listing 3 简化切线法单步迭代核心逻辑

修正切线法

```

1  if (k == 0) {
2      double fx0 = equation.getF().apply(x_curr);
3      IterationState initialState = IterationState.initial(x_curr, fx0);
4      x_prev = x_curr;
5      k++;
6      return initialState;
7  }
8
9  double fx = equation.getF().apply(x_prev);
10
11 if (k == 1 || (k - 1) % this.updateInterval == 0) {
12     this.derivative_approx = (equation.getF().apply(x_prev + H) - fx) / H
13 ;
14 }
15 // --- 执行一次修正切线法迭代 ---
16 if (Math.abs(this.derivative_approx) < 1e-12) {
17     x_curr = Double.NaN;
18 } else {
19     x_curr = x_prev - fx / this.derivative_approx;
20 }

```

Listing 4 修正切线法单步迭代核心逻辑

单点弦截法

```

1  if (k == 1) {
2      double error_abs = Math.abs(x_curr - x_prev);
3      IterationState firstState = new IterationState(k, x_curr, x_prev,
4          equation.getF().apply(x_curr), error_abs, Double.NaN);
5      x_prev_prev = x_prev;
6      x_prev = x_curr; k++;
7      return firstState;
8  }
9  // --- 执行一次单点割线法迭代 ---
10 double fx_prev = equation.getF().apply(x_prev);
11 double denominator = fx_prev - fx_fixed;
12
13 if (Math.abs(denominator) < 1e-12) {
14     x_curr = Double.NaN;
15 } else {
16     x_curr = x_prev - fx_prev * (x_prev - x_fixed) / denominator;
17 }
```

Listing 5 单点弦截法单步迭代核心逻辑

艾特肯迭代法

```

1  if (k == 0) {
2      double fx0 = equation.getF().apply(x_curr);
3      IterationState initialState = IterationState.initial(x_curr, fx0);
4      x_prev = x_curr;
5      k++;
6      return initialState;
7  }
8  // --- 执行一次艾特肯加速迭代 ---
9  double x0_k = x_prev;
10 double x1_k = g.apply(x0_k);
11 double x2_k = g.apply(x1_k);
12 double denominator = x2_k - 2 * x1_k + x0_k;
13 if (Math.abs(denominator) < 1e-12) {
14     x_curr = Double.NaN;
15 } else {
16     x_curr = x0_k - Math.pow(x1_k - x0_k, 2) / denominator;
17 }
```

Listing 6 艾特肯迭代法单步迭代核心逻辑

牛顿下山法

```
1  if (k == 0) {
2      double fx0 = equation.getF().apply(x_curr);
3      IterationState initialState = IterationState.initial(x_curr, fx0);
4      x_prev = x_curr;
5      k++;
6      return initialState;
7  }
8
9  // --- 执行一次下山法迭代 ---
10 double fx_prev = equation.getF().apply(x_prev);
11 double dfx_prev = equation.getDf().apply(x_prev);
12
13 if (Math.abs(dfx_prev) < 1e-12) {
14     x_curr = Double.NaN;
15 } else {
16     double lambda = 1.0;
17     int maxTries = 10;
18     boolean found = false;
19     while (maxTries-- > 0) {
20         double x_next_candidate = x_prev - lambda * (fx_prev / dfx_prev);
21         if (Math.abs(equation.getF().apply(x_next_candidate)) < Math.abs(
22             fx_prev)) {
23             x_curr = x_next_candidate;
24             found = true;
25             break;
26         }
27         lambda /= 2.0;
28     }
29     if (!found) {
30         x_curr = Double.NaN;
31     }
32 }
```

Listing 7 牛顿下山法单步迭代核心逻辑

双点弦截法

```
1  if (k == 0) {
2      k++;
3      return IterationState.initial(x_old, equation.getF().apply(x_old));
4  }
5  if (k == 1) {
6      double error_abs = Math.abs(x_curr - x_old);
7      IterationState state = new IterationState(k, x_curr, x_old, equation.
8          getF().apply(x_curr), error_abs, Double.NaN);
9      x_older = x_old;
10     x_old = x_curr;
11     k++;
12     return state;
13 }
14 // --- 执行一次双点割线法迭代 ---
15 double fx_old = equation.getF().apply(x_old);
16 double fx_older = equation.getF().apply(x_older);
17 double denominator = fx_old - fx_older;
18
19 if (Math.abs(denominator) < 1e-12) {
20     x_curr = Double.NaN;
21 } else {
22     x_curr = x_old - fx_old * (x_old - x_older) / denominator;
23 }
```

Listing 8 双点弦截法单步迭代核心逻辑

B 线性方程组的直接解法完整代码

克劳特消元法

```

1 // 消元
2 for (int i = 0; i < n; i++) {
3     double pivot = aug.getEntry(i, i);
4     if (Math.abs(pivot) < EPSILON) {
5         history.add(new MatrixState("错误: 主元 A(" + (i + 1) + ", " + (i +
6             1) + ") 为零或过小, 无法继续", aug.copy(), new int[] {i}));
7         return new DirectSolution(history, null);
8     }
9
10    // a. 归一化当前行, 使主元变为1
11    if (Math.abs(pivot - 1.0) > EPSILON) {
12        aug.setRowVector(i, aug.getRowVector(i).mapDivide(pivot));
13        String desc = String.format("归一化: R%d = R%d / %.3f", i + 1, i +
14            1, pivot);
15        history.add(new MatrixState(desc, aug.copy(), new int[] {i}));
16    }
17
18    // b. 对当前主元下方的所有行进行消元
19    for (int j = i + 1; j < n; j++) {
20        double factor = aug.getEntry(j, i);
21        if (Math.abs(factor) < EPSILON) continue;
22
23        // 执行行变换: Rj = Rj - factor * Ri
24        RealVector rowI = aug.getRowVector(i).mapMultiply(factor);
25        RealVector rowJ = aug.getRowVector(j).subtract(rowI);
26        aug.setRowVector(j, rowJ);
27
28        String desc = String.format("行变换: R%d = R%d - (%.3f) * R%d", j +
29            1, j + 1, factor, i + 1);
30        history.add(new MatrixState(desc, aug.copy(), new int[] {i, j}));
31    }
32
33    history.add(new MatrixState("前向消元完成, 形成主对角线为1的上三角矩阵",
34        aug.copy(), null));
35}

```

Listing 9 克劳特消元法核心逻辑

高斯列主元素消元法

```

1 // 消元
2 for (int i = 0; i < n; i++) {
3     int max = i;
4     for (int j = i + 1; j < n; j++) {
5         if (Math.abs(aug.getEntry(j, i)) > Math.abs(aug.getEntry(max, i)))
6             {
7                 max = j;
8             }
9     }
10    if (i != max) {
11        double[] temp = aug.getRow(i);
12        aug.setRow(i, aug.getRow(max));
13        aug.setRow(max, temp);
14        history.add(new MatrixState("行交换: R" + (i + 1) + " <-> R" + (max
+ 1), aug.copy(), new int[]{i, max}));
15    }
16    if (Math.abs(aug.getEntry(i, i)) < EPSILON) {
17        history.add(new MatrixState("错误: 主元过小, 矩阵奇异或接近奇异",
18        aug.copy(), new int[]{i}));
19        return new DirectSolution(history, null);
20    }
21    for (int j = i + 1; j < n; j++) {
22        double factor = aug.getEntry(j, i) / aug.getEntry(i, i);
23        if (Math.abs(factor) < EPSILON) continue;
24        RealVector rowI = aug.getRowVector(i).mapMultiply(factor);
25        RealVector rowJ = aug.getRowVector(j).subtract(rowI);
26        aug.setRowVector(j, rowJ);
27        String desc = String.format("行变换: R%d = R%d - (%.3f) * R%d", j +
28        1, j + 1, factor, i + 1);
29        history.add(new MatrixState(desc, aug.copy(), new int[]{i, j}));
30    }
31    history.add(new MatrixState("前向消元完成, 形成上三角矩阵", aug.copy(),
32    null));
33}

```

Listing 10 高斯列主元素消元法核心逻辑

高斯全主元素消元法

```

1 // 消元
2 for (int i = 0; i < n; i++) {
3     int pivotRow = i;
4     int pivotCol = i;
5     double maxVal = Math.abs(aug.getEntry(i, i));
6
7     for (int row = i; row < n; row++) {
8         for (int col = i; col < n; col++) {
9             if (Math.abs(aug.getEntry(row, col)) > maxVal) {
10                 maxVal = Math.abs(aug.getEntry(row, col));
11                 pivotRow = row;
12                 pivotCol = col;
13             }
14         }
15     }
16     // --- 行交换 ---
17     if (i != pivotRow) {
18         double[] temp = aug.getRow(i);
19         aug.setRow(i, aug.getRow(pivotRow));
20         aug.setRow(pivotRow, temp);
21         history.add(new MatrixState("行交换: R" + (i + 1) + " <-> R" + (
pivotRow + 1), aug.copy(), new int[]{i, pivotRow}));
22     }
23     // --- 列交换 ---
24     if (i != pivotCol) {
25         RealVector temp = aug.getColumnVector(i);
26         aug.setColumnVector(i, aug.getColumnVector(pivotCol));
27         aug.setColumnVector(pivotCol, temp);
28
29         int tempIndex = colIndices[i];
30         colIndices[i] = colIndices[pivotCol];
31         colIndices[pivotCol] = tempIndex;
32
33         history.add(new MatrixState("列交换: C" + (i + 1) + " <-> C" + (
pivotCol + 1) +
34         ", 新列顺序: " + Arrays.toString(colIndices), aug.copy(), null));
35     }
36
37     if (Math.abs(aug.getEntry(i, i)) < EPSILON) {
38         history.add(new MatrixState("错误: 主元过小, 矩阵奇异或接近奇异",
aug.copy(), new int[]{i}));
39         return new DirectSolution(history, null);
40     }

```

```

41 // 消元
42 for (int j = i + 1; j < n; j++) {
43     double factor = aug.getEntry(j, i) / aug.getEntry(i, i);
44     if (Math.abs(factor) < EPSILON) continue;
45     RealVector rowI = aug.getRowVector(i).mapMultiply(factor);
46     RealVector rowJ = aug.getRowVector(j).subtract(rowI);
47     aug.setRowVector(j, rowJ);
48     String desc = String.format("行变换: R%d = R%d - (%.3f) * R%d", j +
49     1, j + 1, factor, i + 1);
50     history.add(new MatrixState(desc, aug.copy(), new int[]{i, j}));
51 }
52 history.add(new MatrixState("前向消元完成, 形成上三角矩阵", aug.copy(),
53 null));

```

Listing 11 高斯全主元素消元法核心逻辑

高斯消元法

```

1 // 消元
2 for (int i = 0; i < n; i++) {
3     if (Math.abs(aug.getEntry(i, i)) < EPSILON) {
4         history.add(new MatrixState("错误: 主元 A(" + (i + 1) + "," + (i +
5         1) + ") 为零或过小, 无法继续", aug.copy(), new int[]{i}));
6         return new DirectSolution(history, null);
7     }
8     for (int j = i + 1; j < n; j++) {
9         double factor = aug.getEntry(j, i) / aug.getEntry(i, i);
10        if (Math.abs(factor) < EPSILON) continue;
11        RealVector rowI = aug.getRowVector(i).mapMultiply(factor);
12        RealVector rowJ = aug.getRowVector(j).subtract(rowI);
13        aug.setRowVector(j, rowJ);
14        String desc = String.format("行变换: R%d = R%d - (%.3f) * R%d", j +
15        1, j + 1, factor, i + 1);
16        history.add(new MatrixState(desc, aug.copy(), new int[]{i, j}));
17    }
18 history.add(new MatrixState("前向消元完成, 形成上三角矩阵", aug.copy(),
19 null));

```

Listing 12 高斯消元法核心逻辑

C 线性方程组的迭代解法完整代码

雅可比迭代法

```
1  @Override
2  public VectorIterationState next() {
3      if (k == 0) {
4          k++;
5          return new VectorIterationState(0, x0.copy(), residualNorm);
6      }
7
8      RealVector x_new = new ArrayRealVector(x.getDimension());
9      for (int i = 0; i < a.getRowDimension(); i++) {
10         double sigma = 0;
11         for (int j = 0; j < a.getColumnDimension(); j++) {
12             if (i != j) {
13                 sigma += a.getEntry(i, j) * x.getEntry(j);
14             }
15         }
16         x_new.setEntry(i, (b.getEntry(i) - sigma) / a.getEntry(i, i));
17     }
18     x = x_new;
19     residualNorm = a.operate(x).subtract(b).getNorm();
20     return new VectorIterationState(k++, x.copy(), residualNorm);
21 }
22 }
```

Listing 13 雅可比迭代法核心逻辑

高斯-赛德尔迭代法

```
1  @Override
2  public VectorIterationState next() {
3      if (k == 0) {
4          k++;
5          return new VectorIterationState(0, x0.copy(), residualNorm);
6      }
7
8      RealVector x_new = x.copy();
9      for (int i = 0; i < a.getRowDimension(); i++) {
10         double sigma1 = 0;
11         for (int j = 0; j < i; j++) {
12             sigma1 += a.getEntry(i, j) * x_new.getEntry(j);
13         }
14
15         double sigma2 = 0;
16         for (int j = i + 1; j < a.getColumnDimension(); j++) {
17             sigma2 += a.getEntry(i, j) * x.getEntry(j);
18         }
19
20         x_new.setEntry(i, (b.getEntry(i) - sigma1 - sigma2) / a.getEntry(i,
21 i));
22     }
23
24     x = x_new;
25     residualNorm = a.operate(x).subtract(b).getNorm();
26
27     return new VectorIterationState(k++, x.copy(), residualNorm);
28 }
```

Listing 14 高斯-赛德尔迭代法核心逻辑

松弛迭代法

```
1  @Override
2  public VectorIterationState next() {
3      if (k == 0) {
4          k++;
5          return new VectorIterationState(0, x0.copy(), residualNorm);
6      }
7
8      RealVector x_old = x.copy();
9
10     for (int i = 0; i < a.getRowDimension(); i++) {
11         double sigma1 = 0;
12         for (int j = 0; j < i; j++) {
13             sigma1 += a.getEntry(i, j) * x.getEntry(j);
14         }
15
16         double sigma2 = 0;
17         for (int j = i + 1; j < a.getColumnDimension(); j++) {
18             sigma2 += a.getEntry(i, j) * x_old.getEntry(j);
19         }
20
21         double gs_component = (b.getEntry(i) - sigma1 - sigma2) / a.
22         getEntry(i, i);
23
24         double new_xi = (1 - omega) * x_old.getEntry(i) + omega *
25         gs_component;
26         x.setEntry(i, new_xi);
27     }
28
29     residualNorm = a.operate(x).subtract(b).getNorm();
30     return new VectorIterationState(k++, x.copy(), residualNorm);
}
```

Listing 15 松弛迭代法核心逻辑

D 函数识别与求值部分完整代码

```
1  public class Equation {  
2  
3      private final ManualExpression f, g;  
4      private static final double H = 1e-7;  
5  
6      public Equation(String fStr, String gStr) {  
7          this.f = new ManualExpression(fStr);  
8          if (gStr != null && !gStr.isBlank()) {  
9              this.g = new ManualExpression(gStr);  
10         } else {  
11             this.g = new ManualExpression(null);  
12         }  
13     }  
14  
15     public Function<Double, Double> getF() {  
16         return f::evaluate;  
17     }  
18  
19     public Function<Double, Double> getG() {  
20         return g::evaluate;  
21     }  
22     public Function<Double, Double> getDf() {  
23  
24         return (x) -> {  
25             try {  
26                 double f_x_plus_h = f.evaluate(x + H);  
27                 double f_x_minus_h = f.evaluate(x - H);  
28                 return (f_x_plus_h - f_x_minus_h) / (2 * H);  
29             } catch (Exception e) {  
30                 return Double.NaN;  
31             }  
32         };  
33     }  
34 }
```

Listing 16 函数识别核心逻辑

```

1  private List<String> tokenize(String expression) {
2      List<String> tokens = new ArrayList<>();
3
4      expression = expression.replace("(", " ( ");
5      expression = expression.replace(")", ") ");
6      expression = expression.replace(",", ", , , ");
7      for (String op : PRECEDENCE.keySet()) {
8          if (!op.equals(UNARY_MINUS)) {
9              expression = expression.replace(op, " " + op + " ");
10         }
11     }
12
13     String[] rawTokens = expression.trim().split("\\s+");
14     Collections.addAll(tokens, rawTokens);
15
16     return tokens;
17 }
18

```

Listing 17 词法分析

```

1  private List<String> shuntingYard(List<String> tokens) {
2      List<String> outputQueue = new ArrayList<>();
3      Stack<String> operatorStack = new Stack<>();
4      String lastToken = null;
5      for (String token : tokens) {
6          if (isNumber(token) || CONSTANTS.containsKey(token.toLowerCase()))
7          {
8              outputQueue.add(token);
9          } else if (isVariable(token)) {
10              outputQueue.add(token);
11          } else if (FUNCTIONS.containsKey(token.toLowerCase())) {
12              operatorStack.push(token);
13          } else if (token.equals(",")) {
14              while (!operatorStack.isEmpty() && !operatorStack.peek().equals("(")) {
15                  outputQueue.add(operatorStack.pop());
16              }
17              if (operatorStack.isEmpty()) {
18                  throw new IllegalArgumentException("非法输入：逗号位置不当或括号不匹配。");
19              }
20          } else if (token.equals("-") && (lastToken == null || PRECEDENCE.
containsKey(lastToken) || lastToken.equals("("))) {
21              operatorStack.push(UNARY_MINUS);
22          } else if (PRECEDENCE.containsKey(token)) {
23              while (!operatorStack.isEmpty() && PRECEDENCE.containsKey(

```

```

operatorStack.peek())));
21     String op2 = operatorStack.peek();
22     if ((ASSOCIATIVITY.get(token) && PRECEDENCE.get(token) <=
PRECEDENCE.get(op2)) ||
23         (!ASSOCIATIVITY.get(token) && PRECEDENCE.get(token) <
PRECEDENCE.get(op2))) {
24         outputQueue.add(operatorStack.pop());
25     } else {
26         break;}}
27     operatorStack.push(token);
28 } else if (token.equals("(")) {
29     operatorStack.push(token);
30 } else if (token.equals(")")) {
31     while (!operatorStack.isEmpty() && !operatorStack.peek().equals(
32         "(")) {
33         outputQueue.add(operatorStack.pop());}
34     if (operatorStack.isEmpty()) {
35         throw new IllegalArgumentException("非法输入：括号不匹配。");}
36     operatorStack.pop();
37     if (!operatorStack.isEmpty() && FUNCTIONS.containsKey(
operatorStack.peek().toLowerCase())) {
38         outputQueue.add(operatorStack.pop());}
39     } else {
40         throw new IllegalArgumentException("非法输入：包含无法识别的符号
" + token + "');");
41     lastToken = token;}
42     while (!operatorStack.isEmpty()) {
43         String op = operatorStack.pop();
44         if (op.equals("(")) {
45             throw new IllegalArgumentException("非法输入：括号不匹配。");}
46         outputQueue.add(op);}
47     return outputQueue;
48 }

```

Listing 18 调度场算法

```

1  public double evaluate(double xValue) {
2      if (variables.size() > 1 || (variables.size() == 1 && !variables.
3          contains("x")))) {
4          throw new IllegalArgumentException("该表达式需要 'x' 之外的变量，无
5              法仅用 xValue 求值。");
6      Stack<Double> valueStack = new Stack<>();
7      for (String token : rpnExpression) {
8          if (isNumber(token)) {
9              valueStack.push(Double.parseDouble(token));
10         } else if (CONSTANTS.containsKey(token.toLowerCase())) {
11             valueStack.push(CONSTANTS.get(token.toLowerCase()));
12         } else if (token.equalsIgnoreCase("x")) {
13             valueStack.push(xValue);
14         } else if (PRECEDENCE.containsKey(token)) {
15             if (token.equals(UNARY_MINUS)) {
16                 if (valueStack.isEmpty()) throw new IllegalArgumentException("非法输入：运算符缺少操作数。");
17                 valueStack.push(-valueStack.pop());
18             } else {
19                 if (valueStack.size() < 2) throw new IllegalArgumentException("非法输入：运算符缺少操作数。");
20                 double b = valueStack.pop();
21                 double a = valueStack.pop();
22                 valueStack.push(applyOperator(token, a, b));
23             }
24         } else if (FUNCTIONS.containsKey(token.toLowerCase())) {
25             String func = token.toLowerCase();
26             int numArgs = FUNCTIONS.get(func);
27             if (valueStack.size() < numArgs) {
28                 throw new IllegalArgumentException("非法输入：函数 '" + func +
29                     "' 的参数数量不足。");
30             List<Double> args = new ArrayList<>();
31             for (int i = 0; i < numArgs; i++) {
32                 args.add(valueStack.pop());
33             Collections.reverse(args);
34             valueStack.push(applyFunction(func, args));
35         }
36         if (valueStack.size() != 1) {
37             throw new IllegalArgumentException("非法输入：表达式格式错误，导致
38 最终栈中元素不唯一。");
39         }
40     }
41     return valueStack.pop();
42 }

```

Listing 19 函数求值