数值分析方程的迭代解法研究报告

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 10 月 12 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，这一类方法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，因此，本报告的目的是在计算机中实际应用上述迭代方法，以加深自己对迭代解法的理解。

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；其二 JAVA 还提供较为方便的 javafx 方便将我的程序更好的可视化；其三 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。

在上机期间我手动编写了有关迭代方法——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。

本报告展示了不同迭代方法的代码实现、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；牛顿迭代法；可视化软件

目录

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。

本报告的核心任务包含以下三点：一是研究课内外常见的 8 种迭代解法[2]；二为利用 JAVA 打造一个迭代方法可视化软件。该软件允许用户自主输入不同的方程并选择不同迭代策略来执行迭代求解操作，此外我的软件还提供了相关的指标来深入分析各个迭代方法；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的表现。

2　迭代解法的原理与流程

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：

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

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。

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：

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

两边取绝对值得到：

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

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。如果我们能在一个包含根的区间 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 $x_0, x_1, x_2, \ldots$。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：$x_{k+1} = g(x_k), \quad k = 0, 1, 2, \ldots$

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 $g(x) \in [a,b]$。

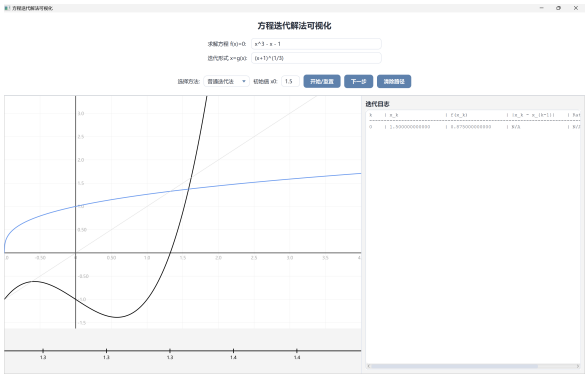
2. 存在一个常数 $0 \leq L < 1$，使得对于任意 $x \in (a,b)$，都有 $|g'(x)| \leq L$。

优缺点

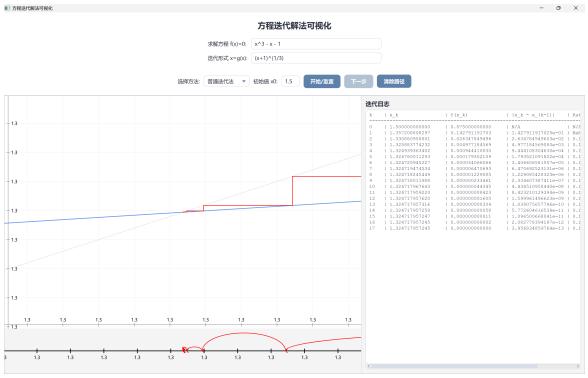
- 优点：算法简单朴素，易于理解。
- 缺点：
  - 迭代函数的构造不唯一，不同的构造方式可能导致收敛性不同，甚至发散。
  - 收敛速度通常较慢，仅为一阶收敛。
  - 对初值的选取比较敏感，需要选取在根附近的初值才能保证收敛。

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）。



(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 1 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：

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

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：

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

在根 x_0 附近，由于 $f(x) \approx 0$，上式简化为：

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

我们令 $g'(x) = 0$，解得：

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

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：

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

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。

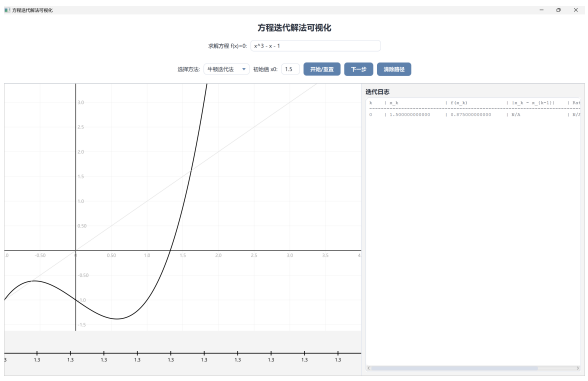
优缺点

- 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。

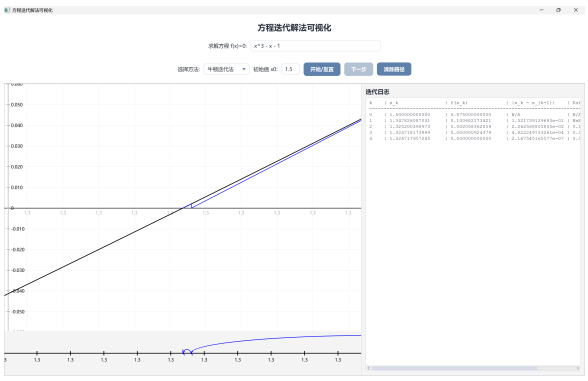
- 需要计算导数，对复杂函数求导计算量大，该方法受限。

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 2 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：

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

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

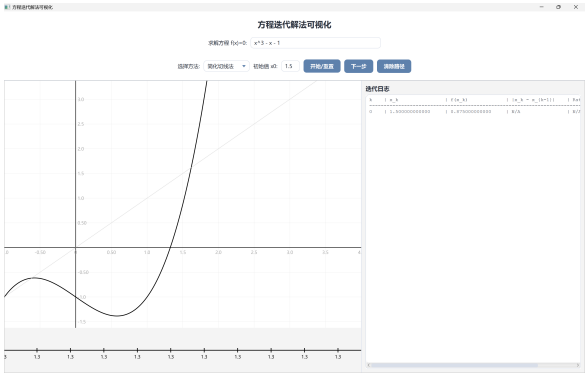
该方法线性收敛，收敛速度慢于牛顿法。当初始值 x_0 离根足够近并且在根的邻域内满足条件 $|1 - f'(x)/f'(x_0)| < 1$，该方法是收敛的。

优缺点

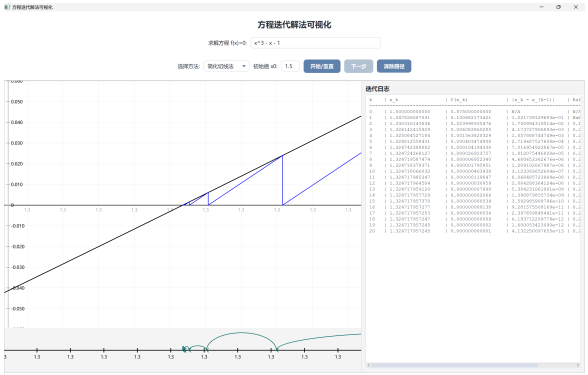
- 优点：极大地减少了计算量（只计算一次导数）。
- 缺点：
  - 收敛速度从牛顿法的二阶下降到一阶，收敛速度变慢。
  - 对初始值的选取较为敏感。

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（详细代码见附录 A）。



(a) 简化切线法软件运行截图 1



(b) 简化切线法软件运行截图 2

图 3 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：

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

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

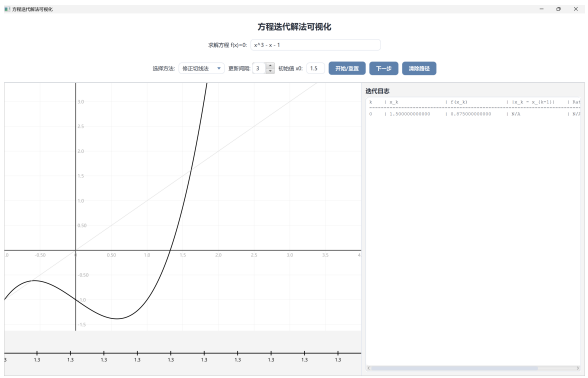
收敛性介于牛顿法和简化切线法之间。更新频率 N 越小，收敛速度越快，但单步计算成本也越高。

优缺点

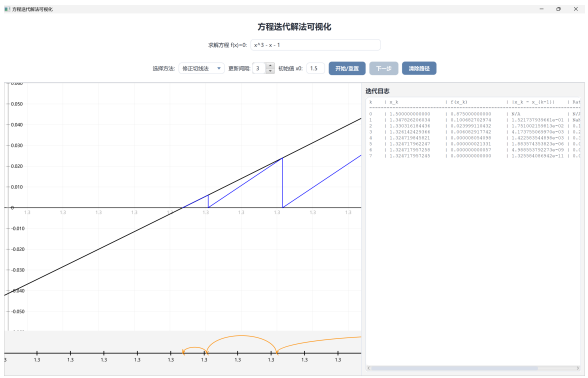
- 优点：计算成本和收敛速度相对平衡。
- 缺点：需要额外设定一个更新频率参数 N，选择不当会影响效率。

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（详细代码见附录 A）。



(a) 修正切线法软件运行截图 1



(b) 修正切线法软件运行截图 2

图 4 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：

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

其中 $\lambda \in (0, 1]$ 是通过搜索选择的，以确保 $|f(x_{k+1})| < |f(x_k)|$ 成立。

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，若不满足则返回第 2 步。

收敛性分析

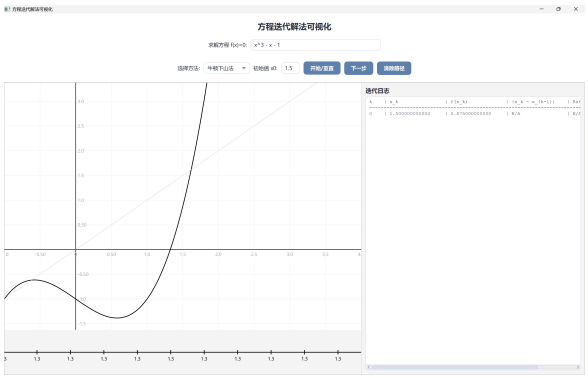
下山法相比牛顿法能够更大范围的收敛。

优缺点

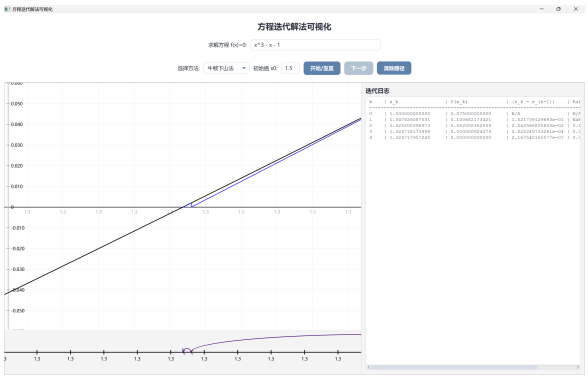
- 优点：相比牛顿法扩大了收敛域，对初值选择不那么敏感。
- 缺点：算法更复杂，增加了单次迭代的计算量。

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（详细代码见附录 A）。



(a) 牛顿下山法软件运行截图 1



(b) 牛顿下山法软件运行截图 2

图 5 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：

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

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 $k = 1, 2, \ldots$，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

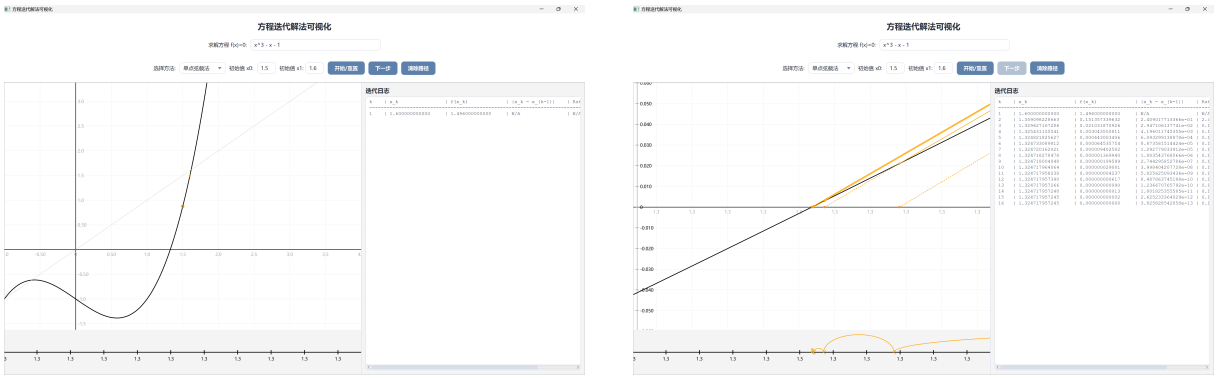
该方法线性收敛，收敛速度通常慢于牛顿法。

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 6 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：

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

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 $k = 1, 2, \ldots$，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

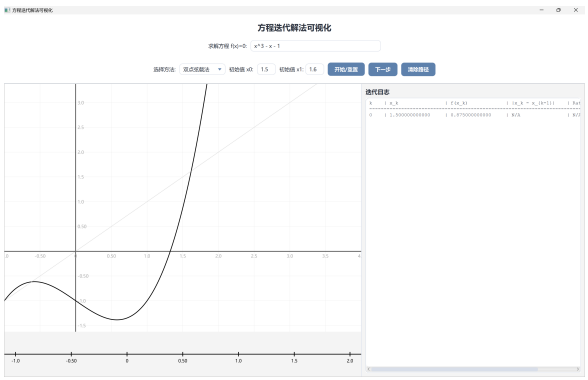
该方法具有超线性收敛性，收敛阶约为 1.618。其收敛速度快于线性收敛方法，但慢于牛顿法的二阶收敛。

优缺点

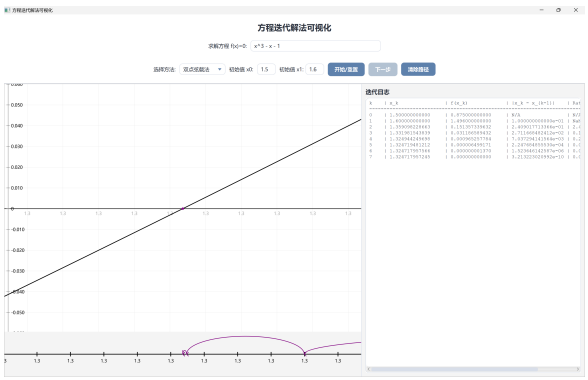
- 优点：无需计算导数，同时保持了较快的收敛速度。
- 缺点：与牛顿法一样是局部收敛的，不能保证对任意初值都收敛。

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 7 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：

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

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

$$f(x_k) = x_k - g(x_k) = x_k - x_{k+1}$$
$$f(x_{k+1}) = x_{k+1} - g(x_{k+1}) = x_{k+1} - x_{k+2}$$

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} 即为加速后的新迭代值 $\hat{x}_k$:

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

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：

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

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：

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

算法步骤

1. 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

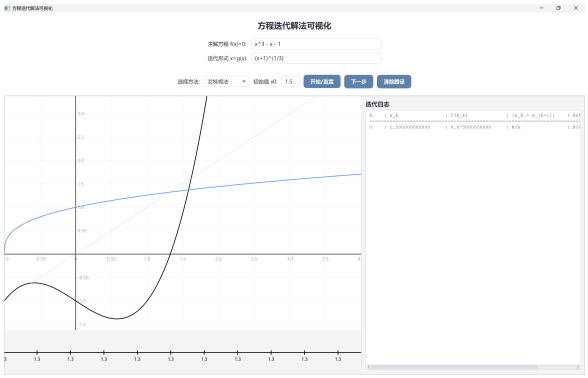
- 优点：显著提高线性收敛迭代法的收敛速度。
- 缺点：
  - 每次迭代的计算复杂性增加。
  - 分母 $x_{k+2} - 2x_{k+1} + x_k$ 接近于零时会出现数值不稳定问题。

优缺点

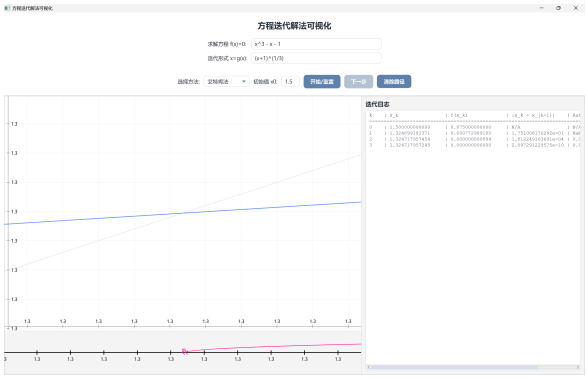
- 优点：算法简单朴素，易于理解。
- 缺点：
  - 迭代函数的构造不唯一，不同的构造方式可能导致收敛性不同，甚至发散。
  - 收敛速度通常较慢，仅为一阶收敛。
  - 对初值的选取比较敏感，需要选取在根附近的初值才能保证收敛。

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）。



(a) 艾特肯法软件运行截图 1



(b) 艾特肯法软件运行截图 2

图 8 艾特肯法软件运行截图

3 java 软件实现

3.1 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。

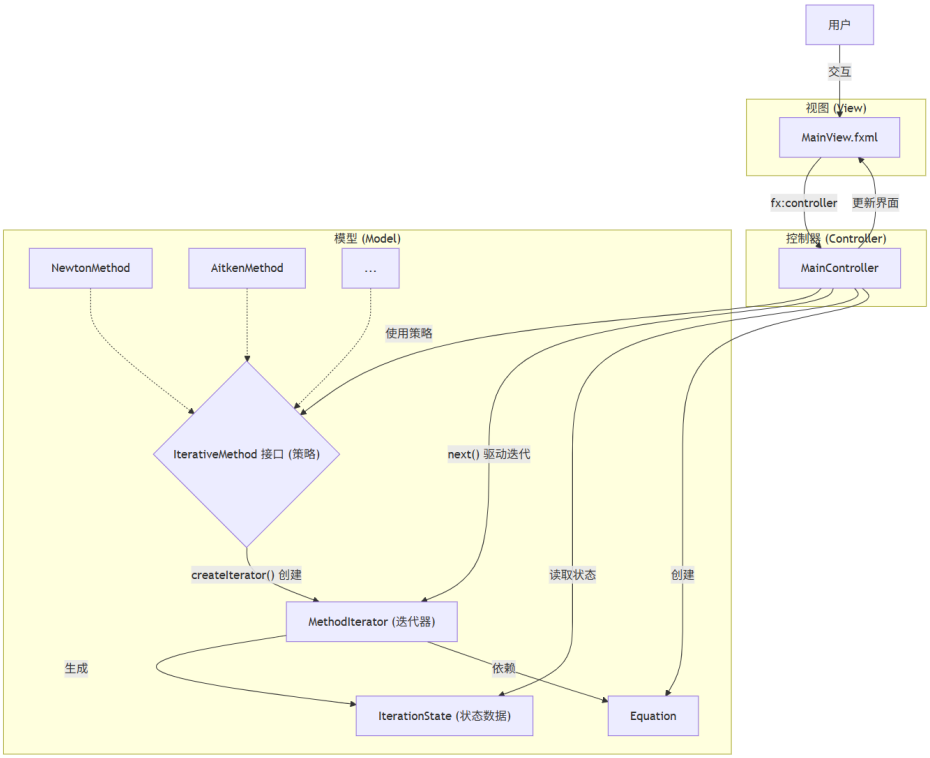


图 9 软件的 MVC 架构图

3.1.1 模型

模型层 (Model) 存储了 8 种不同迭代策略，是运行的核心部分，主要包含下面内容：

- `Equation` 类：负责解析用户输入的函数。
- `IterativeMethod` 接口：定义了迭代方法的规范。
- `MethodIterator` 类：采用迭代器模式，用于管理和执行分步迭代过程。
- `IterationState` 记录：用于封装单次迭代计算后的状态信息。

3.1.2 视图

视图 (View) 主要负责向用户呈现内容，包含以下部分：

- 控制面板: 位于界面顶部，包含用于输入方程、选择迭代方法、设置初始值等控件。

- 可视化面板: 界面中心区域，由两个绘图控件组成。
- 日志面板: 位于界面右侧，实时显示每一次迭代的详细数值结果。

3.1.3 控制器

控制器 (Controller) 连接视图和模型，是管理软件行为的角色。

- 事件处理: 监听视图层控件触发的事件。
- 任务调度: 调用模型层的对象执行核心计算任务。
- 视图更新: 调用视图层的相应方法来更新界面显示。

通过这种方式，MVC 架构确保了各部分的职责单一和明确，使得整个软件结构清晰，易于理解和维护。

3.2 核心功能实现

3.2.1 函数字符串识别

为了让软件具备更好的灵活性，这里我使用了 java 的强大第三方库 exp4j 来解析用户以字符串形式输入的函数方程。它的具体功能被封装到 `Equation` 类中。这个类的核心工作流程如下：

1. 表达式构建: 当用户输入函数字符串时，程序会调用 exp4j 的 `ExpressionBuilder` 将字符串解析成一个内部的 `Expression` 对象，并声明 `"x"` 为其唯一变量。

2. 函数求值: `Equation` 类对外提供 `getF()` 方法。当外部代码需要计算 $f(x)$ 在某一点的值时，只需调用此函数。其内部会执行具体的计算将答案数值赋给变量 `"x"` 并返回计算结果。

3. 自动数值微分: 为了使用牛顿法等需要导数 $f'(x)$ 的算法，这里软件在 `Equation` 类中实现了自动数值微分功能。在 `getDf()` 方法中，程序采用中心差分法来近似计算导数：
$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$
其中 h 是一个极小的步长。

3.2.2 迭代功能实现

软件的迭代功能如总体设计中所述由策略加迭代器构成。接下来以牛顿法为例展示一下迭代的完整流程。

该功能的实现分为两个层次:策略类 `NewtonMethod` 和其内部的迭代器类 `NewtonIterator`。

1. 策略类 **(`NewtonMethod`)**: 该类是策略模式的具体实现，它实现了 `IterativeMethod` 接口，负责接收 `Equation` 对象和初始值 `x0`，然后创建并返回一个专门用于执行牛顿法迭代过程的 `NewtonIterator` 实例。

2. 状态管理类 **(NewtonIterator)**: 这个类实现了 `MethodIterator` 接口，是执行迭代计算的核心。

   - 状态封装: 这个类纪录了迭代过程中所有详细的迭代信息。
   - 核心迭代逻辑 **(`next()` 方法)**: 这是迭代器的核心。
     (a) 获取上一步函数值 $f(x_k)$ 和导数值 $f'(x_k)$。
     (b) 应用牛顿法数学公式 $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ 计算出当前值 `x_curr` 同时进行健壮性检查。
     (c) 将本次迭代产生的所有信息（步数、当前值、函数值、误差等）封装到一个数据对象中。
     (d) 更新迭代器内部的状态将步数加一并将当前值滚动赋给上一步迭代值（x_prev。
     (e) 返回封装好的 `IterationState` 对象给控制器，以供其更新视图。
   - 终止条件判断: (`hasNext()` 方法定义了迭代的终止条件，包括达到预设的最大迭代次数，或计算结果出现无效值等状况，以防止程序陷入无限循环。

通过这种方式，算法的定义与算法的单次执行过程被分离开来，使得主控制器的逻辑保持简洁，只需与标准接口交互即可驱动所有不同类型的迭代方法。

3.3 软件界面介绍

下图展示了软件的用户界面。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。

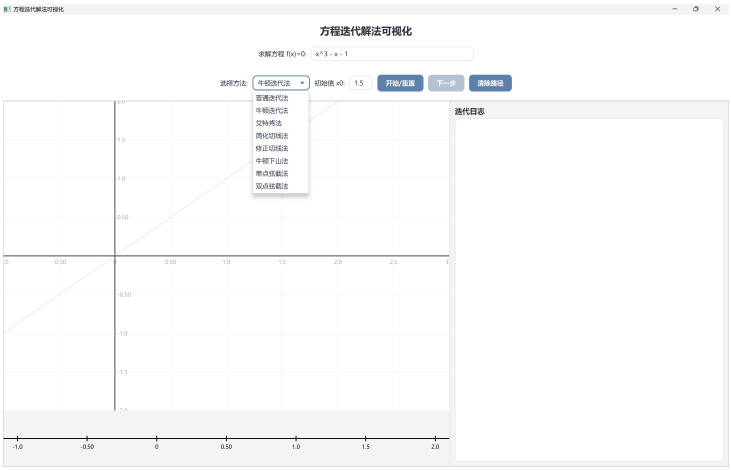


图 10 软件用户界面

4 总结

4.1 工作总结

总的来说本报告较为详细的分析了不同迭代算法各自的数学原理和收敛性分析，就其本质来看，所有相关迭代方法均是由最初的普通迭代法改造而来，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迭代方法在二维坐标系和一维数轴上的表现。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，一定程度上减轻了第二章的学习任务，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/v1.0.1）上下载最新软件。

4.2 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，并针对重根问题提供优化解法。
- 拓展至高维问题: 将软件功能从求解单一方程升级至求解非线性方程组。
- 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.

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 迭代部分完整代码

普通迭代法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }
    // --- 执行一次普通迭代 ---
    x_curr = equation.getG().apply(x_prev);

```

Listing 1 普通迭代法单步迭代核心逻辑

牛顿迭代法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }
    // --- 执行一次牛顿法迭代 ---
    double fx = equation.getF().apply(x_prev);
    double dfx = equation.getDf().apply(x_prev);

    if (Math.abs(dfx) < 1e-12) {
      x_curr = Double.NaN;
    } else {
      x_curr = x_prev - fx / dfx;
    }

```

Listing 2 牛顿迭代法单步迭代核心逻辑

简化切线法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }
    // --- 执行一次简化切线法迭代 ---
    x_curr = x_prev - equation.getF().apply(x_prev) / dfx0;

```

Listing 3 简化切线法单步迭代核心逻辑

修正切线法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }

    double fx = equation.getF().apply(x_prev);

    if (k == 1 || (k - 1) % this.updateInterval == 0) {
      this.derivative_approx = (equation.getF().apply(x_prev + H) - fx) / H
    ;
    }
    // --- 执行一次修正切线法迭代 ---
    if (Math.abs(this.derivative_approx) < 1e-12) {
      x_curr = Double.NaN;
    } else {
      x_curr = x_prev - fx / this.derivative_approx;
    }

```

Listing 4 修正切线法单步迭代核心逻辑

单点弦截法

```
    if (k == 1) {
      double error_abs = Math.abs(x_curr - x_prev);
      IterationState firstState = new IterationState(k, x_curr, x_prev,
    equation.getF().apply(x_curr), error_abs, Double.NaN);
      x_prev_prev = x_prev;
      x_prev = x_curr;
      k++;
      return firstState;
    }
    // --- 执行一次单点割线法迭代 ---
    double fx_prev = equation.getF().apply(x_prev);
    double denominator = fx_prev - fx_fixed;

    if (Math.abs(denominator) < 1e-12) {
      x_curr = Double.NaN;
    } else {
      x_curr = x_prev - fx_prev * (x_prev - x_fixed) / denominator;
    }

```

Listing 5 单点弦截法单步迭代核心逻辑

双点弦截法

```
    if (k == 0) {
      k++;
      return IterationState.initial(x_old, equation.getF().apply(x_old));
    }
    if (k == 1) {
      double error_abs = Math.abs(x_curr - x_old);
      IterationState state = new IterationState(k, x_curr, x_old, equation.
    getF().apply(x_curr), error_abs, Double.NaN);
      x_older = x_old;
      x_old = x_curr;
      k++;
      return state;
    }
    // --- 执行一次双点割线法迭代 ---
    double fx_old = equation.getF().apply(x_old);
    double fx_older = equation.getF().apply(x_older);
    double denominator = fx_old - fx_older;

    if (Math.abs(denominator) < 1e-12) {
```

```
      x_curr = Double.NaN;
    } else {
      x_curr = x_old - fx_old * (x_old - x_older) / denominator;
    }

```

Listing 6 双点弦截法单步迭代核心逻辑

艾特肯迭代法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }

    // --- 执行一次艾特肯加速迭代 ---
    double x0_k = x_prev;
    double x1_k = g.apply(x0_k);
    double x2_k = g.apply(x1_k);

    double denominator = x2_k - 2 * x1_k + x0_k;
    if (Math.abs(denominator) < 1e-12) {
      x_curr = Double.NaN;
    } else {
      x_curr = x0_k - Math.pow(x1_k - x0_k, 2) / denominator;
    }

```

Listing 7 艾特肯迭代法单步迭代核心逻辑

牛顿下山法

```
    if (k == 0) {
      double fx0 = equation.getF().apply(x_curr);
      IterationState initialState = IterationState.initial(x_curr, fx0);
      x_prev = x_curr;
      k++;
      return initialState;
    }

    // --- 执行一次下山法迭代 ---
```

```
    double fx_prev = equation.getF().apply(x_prev);
    double dfx_prev = equation.getDf().apply(x_prev);

    if (Math.abs(dfx_prev) < 1e-12) {
      x_curr = Double.NaN;
    } else {
      double lambda = 1.0;
      int maxTries = 10;
      boolean found = false;
      while (maxTries-- > 0) {
        double x_next_candidate = x_prev - lambda * (fx_prev / dfx_prev);
        if (Math.abs(equation.getF().apply(x_next_candidate)) < Math.abs(
    fx_prev)) {
          x_curr = x_next_candidate;
          found = true;
          break;
        }
        lambda /= 2.0;
      }
      if (!found) {
        x_curr = Double.NaN;
      }
    }

```

Listing 8 牛顿下山法单步迭代核心逻辑

B 函数识别部分完整代码

```
public class Equation {

  private final Expression f, g;
  private static final double H = 1e-7; // 微分小步长

  // 构造函数
  public Equation(String fStr, String gStr) {
    this.f = new ExpressionBuilder(fStr).variable("x").build();
    if (gStr != null && !gStr.isBlank()) {
      this.g = new ExpressionBuilder(gStr).variable("x").build();
    } else {
      this.g = new ExpressionBuilder("0/0").variable("x").build();
    }
  }

  public Function<Double, Double> getF() {
    return (x) -> f.setVariable("x", x).evaluate();
  }

  public Function<Double, Double> getG() {
    return (x) -> g.setVariable("x", x).evaluate();
  }

  // 计算导数
  public Function<Double, Double> getDf() {
    // 使用中心差分公式: (f(x+h) - f(x-h)) / (2h)
    return (x) -> {
      try {
        double f_x_plus_h = f.setVariable("x", x + H).evaluate();
        double f_x_minus_h = f.setVariable("x", x - H).evaluate();
        return (f_x_plus_h - f_x_minus_h) / (2 * H);
      } catch (Exception e) {
        return Double.NaN;
      }
    };
  }
}

```

Listing 9 函数识别核心逻辑