

数值分析上机实验报告

彭浩然 1120242091

2025 年 11 月 30 日

目录

1 实验目标	3
2 实验内容	3
2.1 数值微分方法	3
2.2 一元方程的迭代解法	3
2.2.1 二分法	3
2.2.2 Aitken 加速收敛法	4
2.2.3 Newton 迭代法	4
2.2.4 Newton 下山法	5
2.3 线性方程组的消元法	6
3 实验过程	6
3.1 关于工具的选择	7
3.2 基本结构	7
3.3 求解器与可视化之间的交互设计	7
3.4 可视化器的设计	7
3.5 用户交互设计	8
4 实验结果与分析	9
4.1 二分法	9
4.1.1 输入	9
4.1.2 输出	9
4.2 Aitken 加速收敛法	10
4.2.1 输入	10
4.2.2 输出	10
4.3 Newton 法	11
4.3.1 输入	11
4.3.2 输出	11
4.4 Newton 下山法	13
4.4.1 输入	13
4.4.2 输出	13
4.5 高斯消元法	14
4.5.1 输入	14
4.5.2 输出	14

5 实验总结	15
5.1 一元方程的迭代解法	15
5.2 线性方程组的消元法	16
5.3 其他方面的一些思考	16

1 实验目标

1. 熟悉各种数值计算方法，例如数值微分、方程的迭代解法等。
2. 利用可视化方法加深对数值计算方法的理解（不过如果我都能自己写代码实现了，难道还需要可视化来帮助理解吗……）。
3. 掌握数值计算方法的编程实现，提高编程能力。
4. 体会各种方法的优缺点，了解数值计算方法的适用范围。

2 实验内容

2.1 数值微分方法

数值微分是通过对函数在某一点附近的取值进行计算，来近似求解该函数在该点处的导数值。数值分析本来课堂上是没有讲的，但后面牛顿法要用，所以这里干脆简单实现一下。

三点中心差分公式 这大约是最简便的方法。

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{h^2}{6} f'''(\xi) \quad (\xi \in (x-h, x+h)).$$

其正确性在带 Lagrange 余项的 Taylor 定理视角下是显然的。

从数学上看， h 越小，计算值越接近真实的导数值。然而在数值计算中， h 减小会导致计算除法时分母过小，同时分子上是两个接近的数相减，从而引入较大的舍入误差。因此，我们需要在截断误差与舍入误差之间进行权衡，选出一个合适的 h 值。理论计算和经验都表明，对于双精度浮点数（这也是大多数主流编程语言的默认浮点数类型），选择 $h = \sqrt[3]{\frac{3\varepsilon_{\text{mach}}}{M}} \approx 10^{-5}$ 是使结果精度最高的步长，其中 $\varepsilon_{\text{mach}}$ 是机器精度， M 是函数 f 在区间 $[x-h, x+h]$ 上三阶导数绝对值的上界。

算法步骤

```
1 function get_derivative_of(f, x, h)
2     return x ↦ \frac{f(x+h) - f(x-h)}{2h}
```

2.2 一元方程的迭代解法

2.2.1 二分法

二分查找非常朴素，但又相当高效 ($O(\log \frac{r-l}{\varepsilon})$ 的时间复杂度)、稳定且可靠。二分法对函数要求也很低，只要连续即可。当然，有根区间并不是总能轻易找到。

零点存在定理 若 $f \in C[a, b]$ 且 $f(a)f(b) < 0$ (即在区间端点处函数值异号)，则

$$\exists c \in (a, b) \ (f(c) = 0).$$

即函数 f 在区间 (a, b) 上至少有一个零点。

算法步骤

```

1  function bisection( $f, l, r, \varepsilon$ )
2    while  $r - l > \varepsilon$ 
3       $m \leftarrow \frac{l+r}{2}$ 
4      if  $f(m) = 0$ 
5        return  $m$ 
6      if  $\text{sgn } f(l) = \text{sgn } f(m)$ 
7         $l \leftarrow m$ 
8      else
9         $r \leftarrow m$ 
10     return  $\frac{l+r}{2}$ 

```

2.2.2 Aitken 加速收敛法

对于不动点形式的方程 $x = \varphi(x)$, 引入参数 $\theta \notin \{0, 1\}$ 。将等式 $x = \varphi(x)$ 两边同时减去 θx , 得

$$(1 - \theta)x = \varphi(x) - \theta x,$$

同时除以 $(1 - \theta)$ 得

$$x = \frac{1}{1 - \theta}(\varphi(x) - \theta x) \triangleq \psi(x).$$

于是迭代公式为

$$x^{(n+1)} = \psi(x^{(n)}) = \frac{1}{1 - \theta}(\varphi(x^{(n)}) - \theta x^{(n)}) \quad (n \in \mathbb{N}).$$

在 Aitken 方法中, 我们对 $x^{(n)}$ 迭代两次得到三个相邻得迭代值 $x^{(n)}, y^{(n)} = \varphi(x^{(n)}), z^{(n)} = \varphi(y^{(n)})$ 。取割线斜率作为参数 $\theta^{(n)}$, 即

$$\theta^{(n)} = \frac{z^{(n)} - y^{(n)}}{y^{(n)} - x^{(n)}},$$

则新的迭代公式为

$$x^{(n+1)} = \frac{1}{1 - \theta^{(n)}}(y^{(n)} - \theta^{(n)}x^{(n)}) \quad (n \in \mathbb{N}).$$

算法步骤

```

1  function aitken( $\varphi, x_0, \varepsilon, m$ )
2     $x \leftarrow x_0$ 
3    for  $n \in \mathbb{N} \cap [0, m)$ 
4       $y \leftarrow \varphi(x)$ 
5       $z \leftarrow \varphi(y)$ 
6       $x_{\text{new}} \leftarrow \frac{xz - y^2}{xz - 2y + z} \quad // \text{equivalent, but less rounding error}$ 
7      if  $|x_{\text{new}} - x| < \varepsilon$ 
8        return  $x_{\text{new}}$ 
9       $x \leftarrow x_{\text{new}}$ 
10     return  $x \quad // \text{not converged}$ 

```

2.2.3 Newton 迭代法

对于满足条件的函数, 某一点处的切线与 x 轴的交点往往比该点本身更接近函数的零点。因此, 可以利用切线与 x 轴的交点来逐步逼近零点。

在点 $(x^{(n)}, f(x^{(n)}))$ 处的切线方程为

$$y - f(x^{(n)}) = f'(x^{(n)})(x - x^{(n)}),$$

我们将其与 x 轴的交点取为下一轮的估计值，即

$$x^{(n+1)} = x^{(n)} - \frac{f(x^{(n)})}{f'(x^{(n)})} \quad (n \in \mathbb{N}).$$

利用前面提到的数值微分方法可以很方便地计算 $f'(x^{(n)})$ 的近似值，从而实现 Newton 迭代法。不过，迭代收敛的判断有两个标准：

- 两次迭代结果之差的绝对值小于阈值，即 $|x^{(n+1)} - x^{(n)}| < \varepsilon$ 。
- 函数值的绝对值小于阈值，即 $|f(x^{(n)})| < \varepsilon$ 。

这两个标准各有优缺点，前者更关注结果的准确性，后者更关注结果的有效性。这里我们干脆两个都用上，只要满足其一——要么找到了解（虽然可能不精确），要么收敛在某个点（虽然可能不是零点）——即停止迭代。这么做是合理的，因为如果只有一个条件满足而另一个条件不满足，继续迭代也不会让情况变好。

算法步骤

```

1  function newton( $f, x_0, \varepsilon, m, h$ )
2       $f' \leftarrow get\_derivative\_of(f, h)$ 
3      for  $n \in \mathbb{N} \cap [0, m)$ 
4          if  $|f(x_0)| < \varepsilon$ 
5              return  $x_0$ 
6          if  $|f'(x_0)| < \varepsilon$ 
7              break // to avoid division by 0
8           $d \leftarrow -\frac{f(x_0)}{f'(x_0)}$ 
9           $x \leftarrow x_0 + d$ 
10         if  $|d| < \varepsilon$ 
11             return  $x$ 
12          $x_0 \leftarrow x$ 
13     return  $x_0$  // not converged

```

2.2.4 Newton 下山法

当 Newton 迭代法在某些情况下发散时，可以通过引入步长参数 $\lambda \in (0, 1]$ 来缓解该问题。具体地，我们将迭代公式修改为

$$x^{(n+1)} = x^{(n)} - \lambda \frac{f(x^{(n)})}{f'(x^{(n)})} \quad (n \in \mathbb{N}).$$

通过适当选择步长参数 λ ，可以在一定程度上控制迭代过程，从而提高收敛性。通常情况下，可以通过试探法来选择合适的 λ 值，即从较大的值开始逐渐减小，直到找到一个使得迭代结果收敛的步长。这里我们选择 $\lambda = 2^{-i}$ ($i \in \mathbb{N}$)，即每次减半，直到找到合适的步长，或者 λ 过小以至于无法引起 x 的变化为止。

算法步骤

```
1  function newton_downhill( $f, x_0, \varepsilon, m, h$ )
2       $f' \leftarrow get\_derivative\_of(f, h)$ 
3      for  $n \in \mathbb{N} \cap [0, m)$ 
4          if  $|f(x_0)| < \varepsilon$ 
5              return  $x_0$ 
6          if  $|f'(x_0)| < \varepsilon$ 
7              break // to avoid division by 0
8           $d \leftarrow 1$ 
9          while  $\left| f\left(x_0 - \frac{f(x_0)}{d \cdot f'(x_0)}\right) \right| > |f(x_0)|$  and  $\left| \frac{f(x_0)}{d \cdot f'(x_0)} \right| > \varepsilon$ 
10          $d \leftarrow 2d$ 
11          $x_0 \leftarrow x_0 - \frac{f(x_0)}{d \cdot f'(x_0)}$ 
12     return  $x_0$  // not converged
```

2.3 线性方程组的消元法

这里我们只用 $O(n^3)$ 复杂度的暴力法实现选择列主元的高斯消元法。

算法步骤

```
1  function gaussian_elimination( $a, b$ ) //  $a, b$  are both matrix
2       $(r, c) \leftarrow a.size()$ 
3      for  $i \in \mathbb{N} \cap [0, r)$ 
4           $p \leftarrow \text{argmax}_{j \in [i, r)} |a_{j,i}|$  // column pivoting
5          if  $p \neq i$ 
6               $swap(a_{i,*}, a_{p,*})$ 
7               $swap(b_{i,*}, b_{p,*})$ 
8          for  $j \in \mathbb{N} \cap [i + 1, r)$ 
9               $m \leftarrow \frac{a_{j,i}}{a_{i,i}}$ 
10             for  $k \in \mathbb{N} \cap [i, c)$ 
11                  $a_{j,k} \leftarrow a_{j,k} - m \cdot a_{i,k}$ 
12                 for  $k \in \mathbb{N} \cap [0, b.size()_1)$ 
13                      $b_{j,k} \leftarrow b_{j,k} - m \cdot b_{i,k}$ 
14              $x \leftarrow 0_{c \times b.size()_1}$ 
15             for  $i$  in  $r - 1, \dots, 0$ 
16                 for  $k \in \mathbb{N} \cap [0, b.size()_1)$ 
17                      $sum \leftarrow b_{i,k}$ 
18                     for  $j \in \mathbb{N} \cap [i + 1, c)$ 
19                          $sum \leftarrow sum - a_{i,j} \cdot x_{j,k}$ 
20                      $x_{i,k} \leftarrow \frac{sum}{a_{i,i}}$ 
21     return  $x$ 
```

3 实验过程

其实我不太清楚实验过程要写什么，毕竟无非就是写代码，然后运行，然后修错误。我就大概讲一下我的项目设计罢。

3.1 关于工具的选择

如果说我的自然语言母语是汉语的话，我的编程语言母语应该是 C++。但这也更让我清楚 C++ 的弊端——混乱的包管理。作为一个数值计算项目，我不想浪费时间在配置各种库上，所以我没有选择 C++。

在我看来，我在这个项目中选择蟒蛇（Python）语言，有以下几个优势：

- 极其方便的包管理
- 更贴近数学表达的语法
- 丰富的数值计算（当然，我们要实现算法，就不能调用现成的函数，不过类似 `numpy.ndarray` 这样的工具类可以为我节省时间）与可视化库。

劣势当然也有，例如令我极度反感的动态类型机制。但是没办法，我只能忍受。总的来说，使用蟒蛇是利大于弊的。

可视化也有多种选择，例如传统的 `matplotlib` 库，工业化的 `pyQt` 库，以及可以做非常漂亮的动画的、深受科普工作者喜爱的 `manim` 库等。我最终选择了 `matplotlib`，因为它在保持易用性的同时，提供了足够使用的功能。

3.2 基本结构

分离求解器与可视化模块，使得两者可以独立开发与维护。前者负责实现各种数值计算方法，而后者负责展示计算过程与结果。

在项目中，我将求解器都放 `solvers` 包中，可视化模块放在 `visualizers` 包中。

3.3 求解器与可视化之间的交互设计

每个求解器中都有一个 `SolutionTrace` 对象，它包括以下三方面信息：

- 一个 `Step` 对象的列表，记录每次迭代的信息。
- 最终结果。
- 是否收敛。

将 `Step` 作为保存各次迭代的信息的基类，在各个求解器文件中定义继承自 `Step` 的派生类，保存该求解器特有的迭代信息。采用这种方式，可以统一各个求解器的迭代信息存储格式，提高代码的可维护性和可扩展性。

3.4 可视化器的设计

在开发过程中，我注意到随着迭代次数的增加，估计值越来越接近真实值，图像中的点越来越集中在某个区域，从而导致图像难以观察。为了结果的美观，我让可视化器会绘制左右两个子图，左图中显示全局视图，而右图会放大左图中的焦点部分（用一个虚线矩形框出），以便更清晰地观察迭代过程。

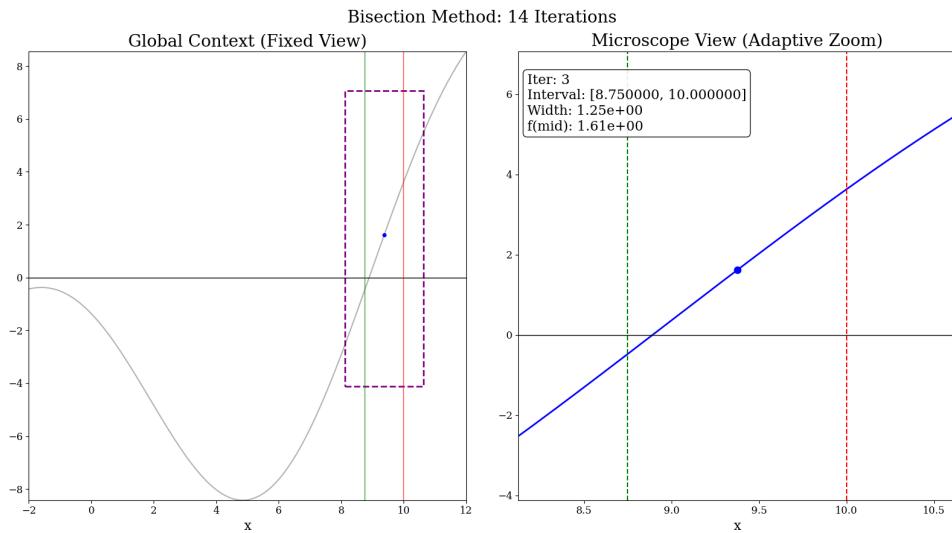


图 1: 可视化界面示意图 (实际上是运行二分法的动画截图)

采用工厂模式 (factory pattern)，使得可视化器的构造函数可以根据不同的求解器类型生成相应的可视化器实例。

3.5 用户交互设计

我最提倡的方式是直接将本项目作为一个包使用。用户可以在蟒蛇脚本或者交互式环境中导入本项目，自行定义求解函数和调用可视化器。

不过为了方便用户使用，我还是实现了简单的命令行交互 (CLI)，通过蟒蛇语言自带的 `eval` 和 `exec` 函数来解析输入的表达式。不过由于这并非课程重点，所以功能相当有限，而且很容易注入攻击。

```

joyce@LAPTOP-R4SOJDAM:~ % + v
-----
Input the parameters: guess=0, tolerance=1e-3
-----
Solution trace:
NewtonDownhillStep(iteration=0, x=0, x_function_value=-8.78401247653964, x_derivative_value=-4.902327156
308672, damping_factor=1)
NewtonDownhillStep(iteration=1, x=-1.7918046259388731, x_function_value=4.529185089082734, x_derivative_
value=-2.360457302064134, damping_factor=2)
NewtonDownhillStep(iteration=2, x=-0.8324174162820565, x_function_value=-1.8852480030965917, x_derivativ
e_value=-9.64223674178033, damping_factor=1)
NewtonDownhillStep(iteration=3, x=-0.0279371959331525, x_function_value=-0.054995359094160534, x_derivat
ive_value=-8.976020633566506, damping_factor=1)
Final result: -1.0340641157943324
Has converged: True
-----
Parameters for visualizer (optional):
-----
Starting animation...
Animation complete.
-----
Select a solver (Ctrl + C to exit):
- BisectionSolver
- NewtonSolver
- NewtonDownhillSolver
^C
User interrupted the execution.
Goodbye!
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ 

```

图 2: 命令行交互示意图

4 实验结果与分析

下面给出各个求解器的示例输入输出。其中最引起我兴趣的是 Newton 法和 Newton 下山法，因为通过可视化可以清晰观察到它们在极值附近“扭动挣扎”，最终逃脱并收敛到真正的零点。

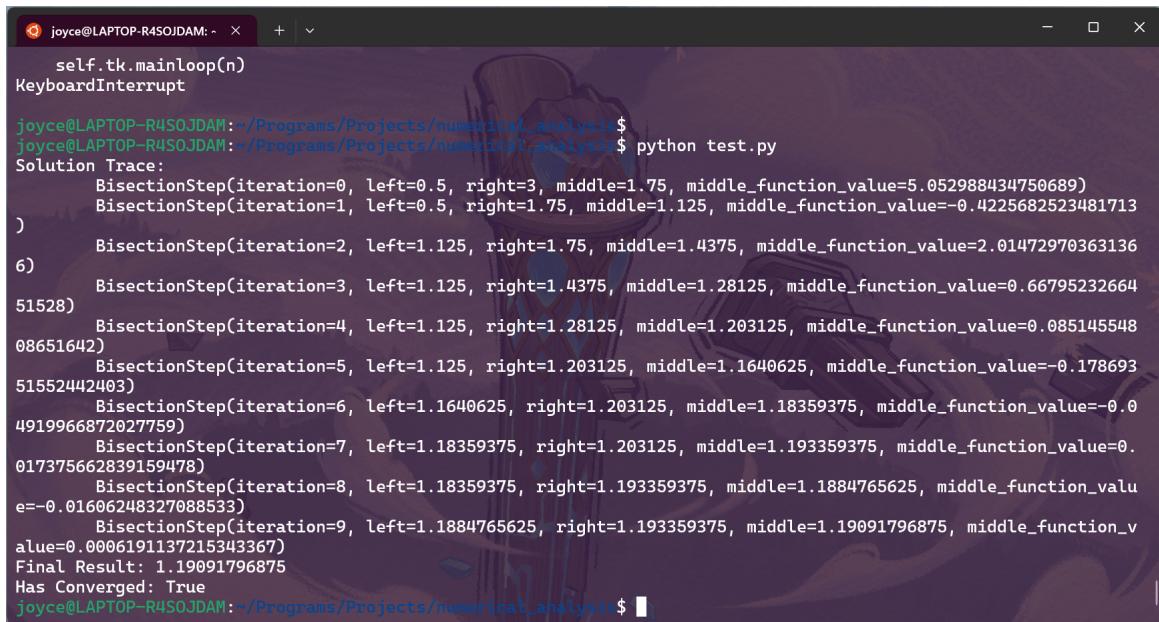
4.1 二分法

4.1.1 输入

我们用二分法求方程 $-0.1 \exp(x) + 4 \sin(2x + 3) + 0.05 * x^3 + 2 * x + 1 = 0$ 在区间 $(0.5, 3)$ 内的一个零点，要求精度达到 10^{-3} 。

4.1.2 输出

可以看到，在 10 次迭代后，算法得到了正确的结果 $x \approx 1.1909$ 。



```
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ python test.py
Solution Trace:
    BisectionStep(iteration=0, left=0.5, right=3, middle=1.75, middle_function_value=5.052988434750689)
    BisectionStep(iteration=1, left=0.5, right=1.75, middle=1.125, middle_function_value=-0.4225682523481713
)
    BisectionStep(iteration=2, left=1.125, right=1.75, middle=1.4375, middle_function_value=2.01472970363136
6)
    BisectionStep(iteration=3, left=1.125, right=1.4375, middle=1.28125, middle_function_value=0.66795232664
51528)
    BisectionStep(iteration=4, left=1.125, right=1.28125, middle=1.203125, middle_function_value=0.085145548
08651642)
    BisectionStep(iteration=5, left=1.125, right=1.203125, middle=1.1640625, middle_function_value=-0.178693
51552442403)
    BisectionStep(iteration=6, left=1.1640625, right=1.203125, middle=1.18359375, middle_function_value=-0.0
4919966872027759)
    BisectionStep(iteration=7, left=1.18359375, right=1.203125, middle=1.193359375, middle_function_value=0.
017375662839159478)
    BisectionStep(iteration=8, left=1.18359375, right=1.193359375, middle=1.1884765625, middle_function_v
alue=-0.01606248327088533)
    BisectionStep(iteration=9, left=1.1884765625, right=1.193359375, middle=1.19091796875, middle_function_v
alue=0.0006191137215343367)
Final Result: 1.19091796875
Has Converged: True
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$
```

图 3: 二分法迭代过程

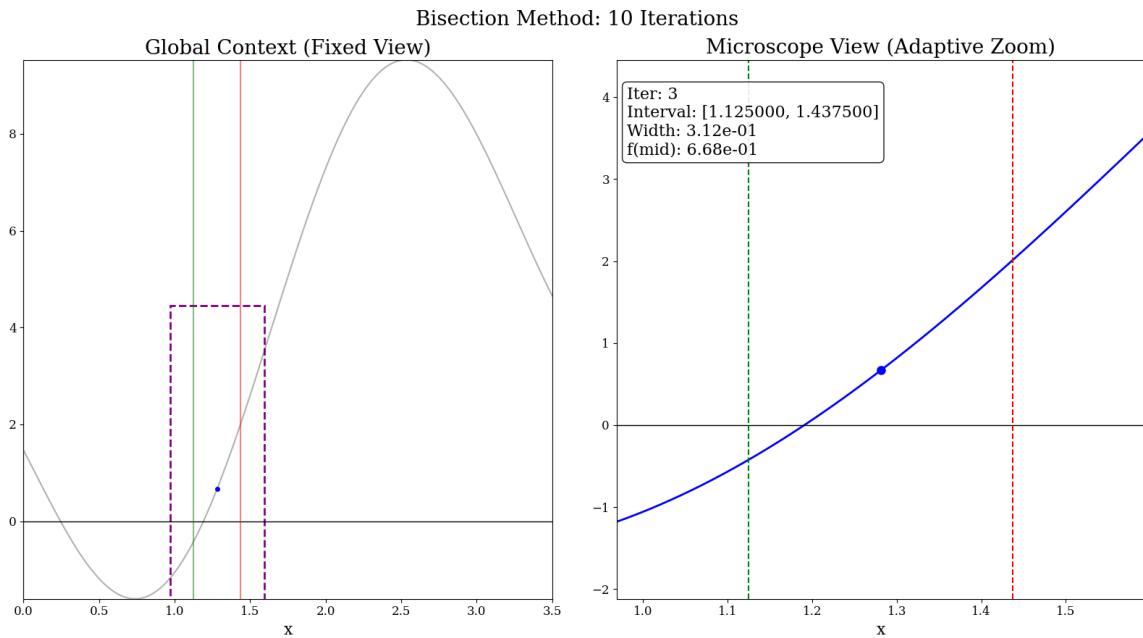


图 4: 二分法可视化动画截图

4.2 Aitken 加速收敛法

4.2.1 输入

不妨选择课本上的一道课后习题：用 Aitken 法求方程 $x^3 - x^2 - 1 = 0$ 在 1.5 附近的根，取精度为 10^{-4} 。

4.2.2 输出

经过 3 次迭代，算法成功收敛到了 $x \approx 1.46557$ 。

```

joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ python test.py
Solution Trace:
    AitkenStep(iteration=0, x=1.5, y=1.625, z=2.275390625, slope=5.203125)
    AitkenStep(iteration=1, x=1.470260223048327, y=1.4868053461709354, z=1.5629324432512028, slope=4.6011804
51552051)
    AitkenStep(iteration=2, x=1.4656658620199963, y=1.465998285986951, z=1.467498956438908, slope=4.51432688
7151999)
Final Result:
1.4655712709424782
Has Converged: True
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ █

```

图 5: Aitken 加速收敛法迭代过程

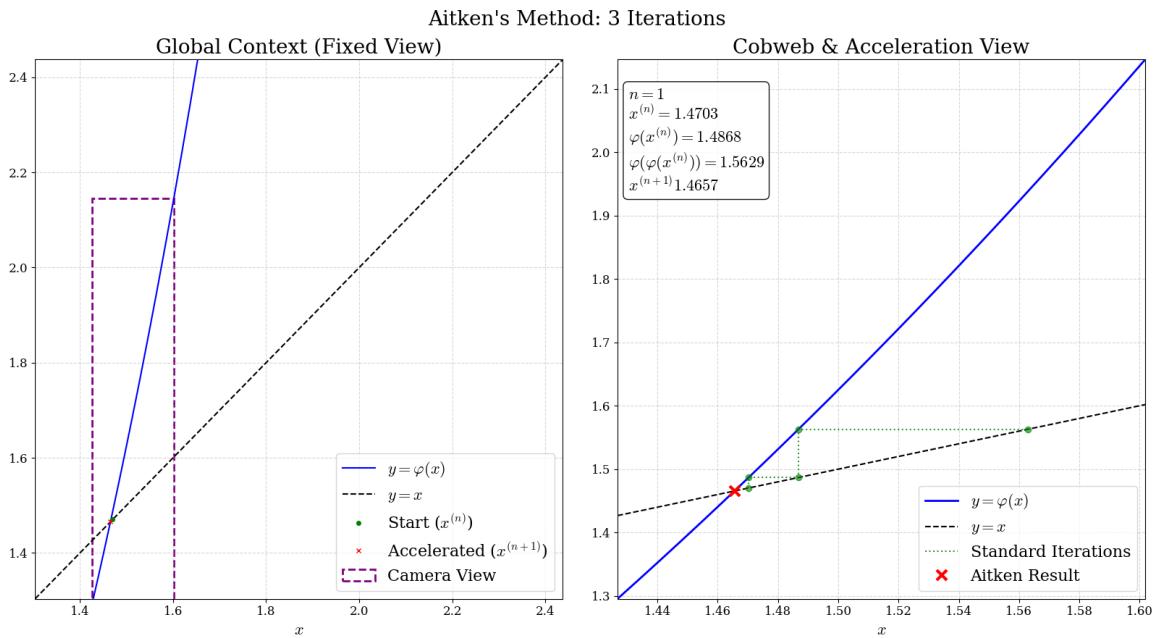


图 6: Aitken 加速收敛法可视化动画截图

4.3 Newton 法

4.3.1 输入

用 Newton 迭代法求方程 $2 \sin(x + 4) + 0.05x^2 - x = 0$ 在 2 附近的一个根。精度同样设置为 10^{-3} 。

4.3.2 输出

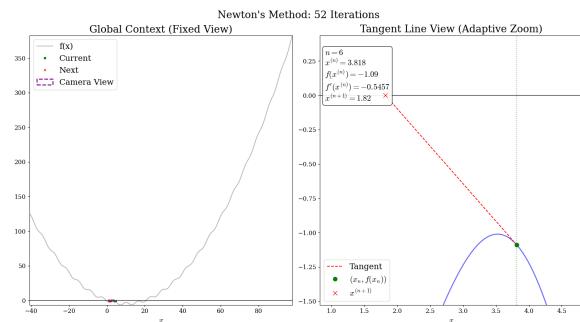
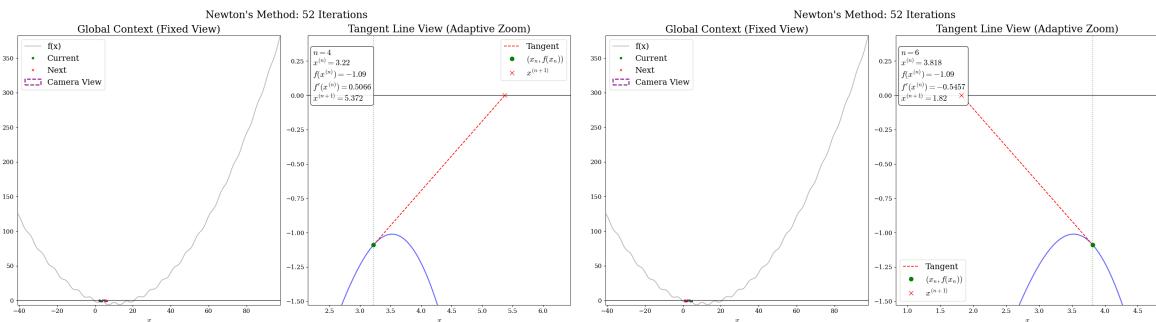
这个函数对 Newton 迭代法其实是很不友好的，它在零点附近比较剧烈地起伏波动。一方面，极值点会困住 x ，它依赖极值点附近的接近 0 的导数值来逃离；另一方面，过大的跳跃又会让 x 跳过零点，在两侧来回反复。不过幸运的是，经过多达 52 次迭代后，我们的算法还是成功收敛到了一个根 $x \approx -0.5642$ 。通过动画，我们可以清晰看到 x 如何被困住又逃脱，逃脱又困住，“一山放出一山拦”；也可以看到 x 是怎样多次错过零点，最终才成功收敛的。

```

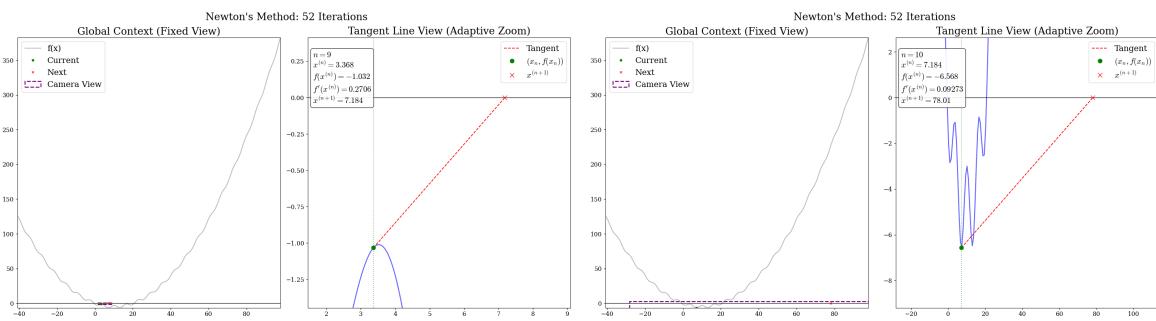
0.6809006796615334)
    NewtonStep(iteration=40, guess=27.922974445269183, function_value=12.032848053391099, derivative_value=3
.540661136369749)
    NewtonStep(iteration=41, guess=24.52449953057333, function_value=5.052925421638715, derivative_value=-0.
48529289990995034)
    NewtonStep(iteration=42, guess=34.93661440506732, function_value=27.981676683725723, derivative_value=3.
1479760536967656)
    NewtonStep(iteration=43, guess=26.047831197562935, function_value=5.917591507339207, derivative_value=2.
007414622440251)
    NewtonStep(iteration=44, guess=23.099964104642353, function_value=5.425346332334442, derivative_value=0.
5377474170842333)
    NewtonStep(iteration=45, guess=13.01094059663244, function_value=-6.475412527087298, derivative_value=-0
.22816357607702284)
    NewtonStep(iteration=46, guess=-15.369630213238397, function_value=29.042612849187435, derivative_value=
-1.8061752259157513)
    NewtonStep(iteration=47, guess=0.7099906521527242, function_value=-2.6847805638699604, derivative_value=
-0.9337975866552383)
    NewtonStep(iteration=48, guess=-2.1651298260298693, function_value=4.330188494101668, derivative_value=
-1.7385436482442171)
    NewtonStep(iteration=49, guess=0.32556892546526495, function_value=-2.1724958886846655, derivative_value
=-1.7219337830498558)
    NewtonStep(iteration=50, guess=-0.9360916041497489, function_value=1.13511727055387, derivative_value=-3
.0875773567799127)
    NewtonStep(iteration=51, guess=-0.5684514961967797, function_value=0.012788514246055538, derivative_valu
e=-2.9733581482171973)
Final Result: -0.5641504622787085
Has Converged: True
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ █

```

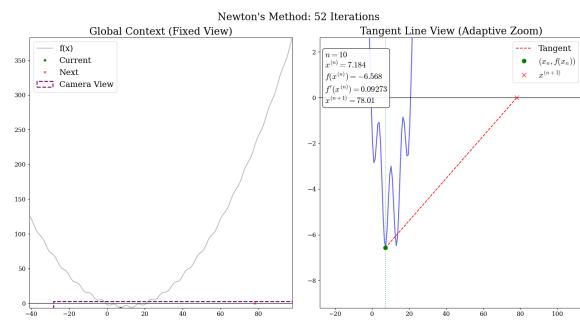
图 7: Newton 法迭代过程



左右腾挪



左右腾挪



步幅增大

成功逃脱

图 8: Newton 法-逃离极值点

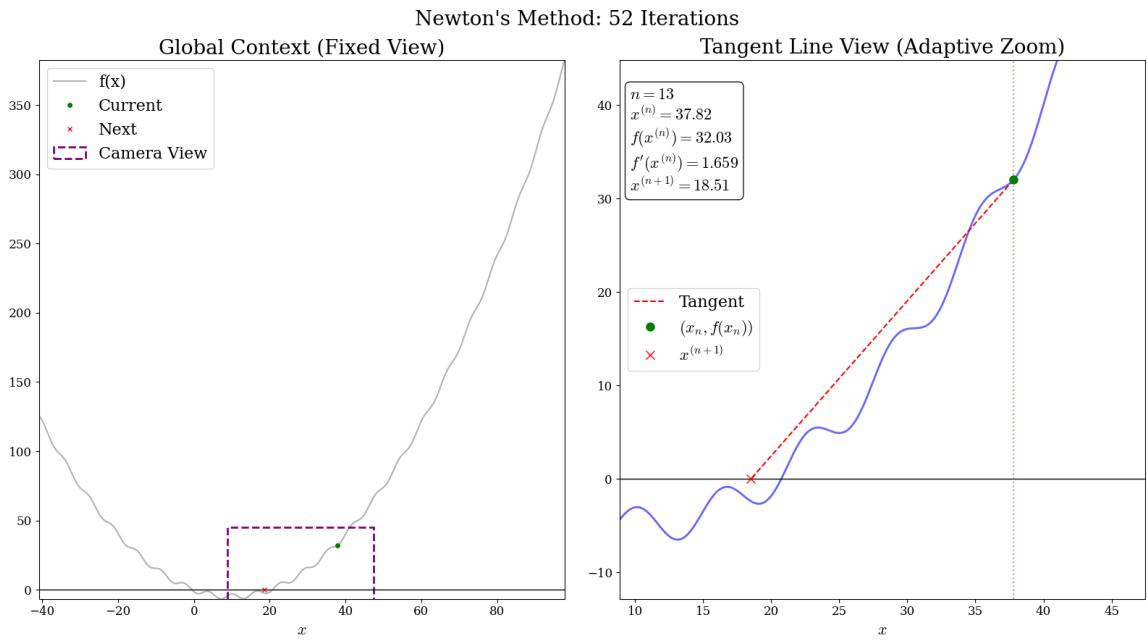


图 9: Newton 法-错过零点

4.4 Newton 下山法

4.4.1 输入

我们用和普通的 Newton 迭代法一样的输入，即 $2 \sin(x + 4) + 0.05x^2 - x = 0$ ，初值设为 2 精度设为 10^{-3} 。

4.4.2 输出

Newton 下山法仅用 9 次迭代就成功收敛到了 $x \approx -0.5643$ ，相比于普通的 Newton 迭代法的 52 次迭代，显著提高了收敛速度。观察动画，我们可以看到， x 仍然会在极值点附近挣扎，而且由于步长调节，其跃出极值点的速度更慢，但避免因为过大的步长而跳过零点，从而更快地收敛。

```

joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$ python test.py
Solution Trace:
    NewtonDownhillStep(iteration=0, x=2, x_function_value=-2.3588309963978515, x_derivative_value=1.12034057
31796678, damping_factor=1)
    NewtonDownhillStep(iteration=1, x=4.105458869264362, x_function_value=-1.3256274613187413, x_derivative_
value=-1.0871241138410426, damping_factor=2)
    NewtonDownhillStep(iteration=2, x=3.4957642426614184, x_function_value=-1.0116992537330454, x_derivative_
value=0.050787097571891586, damping_factor=512)
    NewtonDownhillStep(iteration=3, x=3.5346712721909217, x_function_value=-1.0110719631513856, x_derivative_
value=-0.018709159133045716, damping_factor=4096)
    NewtonDownhillStep(iteration=4, x=3.521477534340472, x_function_value=-1.0109814485716249, x_derivative_
value=0.00496974188468613, damping_factor=65536)
    NewtonDownhillStep(iteration=5, x=3.524581589445907, x_function_value=-1.0109746511283668, x_derivative_
value=-0.000591067395028233, damping_factor=2097152)
    NewtonDownhillStep(iteration=6, x=3.523765996700115, x_function_value=-1.0109747651685352, x_derivative_
value=0.0008706453780149558, damping_factor=2097152)
    NewtonDownhillStep(iteration=7, x=3.524319689795158, x_function_value=-1.010974557800231, x_derivative_v
alue=-0.00012164154128413428, damping_factor=2048)
    NewtonDownhillStep(iteration=8, x=-0.5338327767855953, x_function_value=-0.08972954667362976, x_derivati
ve_value=-2.948956246268119, damping_factor=1)
Final Result:
-0.5642603379904052
Has Converged: True
joyce@LAPTOP-R4SOJDAM:~/Programs/Projects/numerical_analysis$
```

图 10: Newton 下山法迭代过程

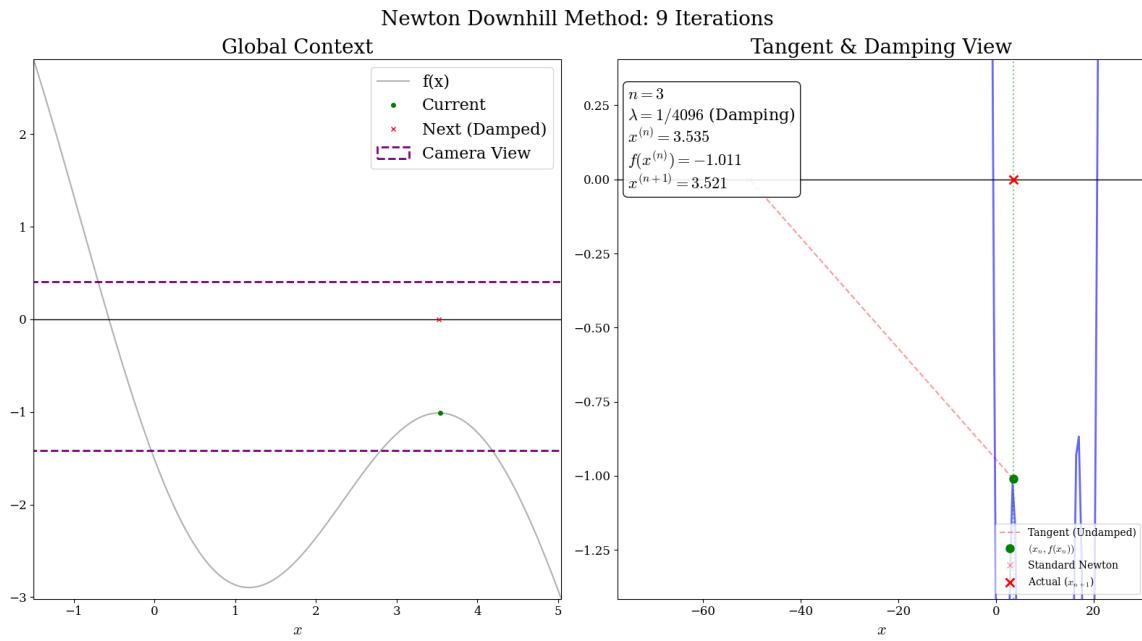


图 11: Newton 下山法-受 λ 调节的逃离极值点

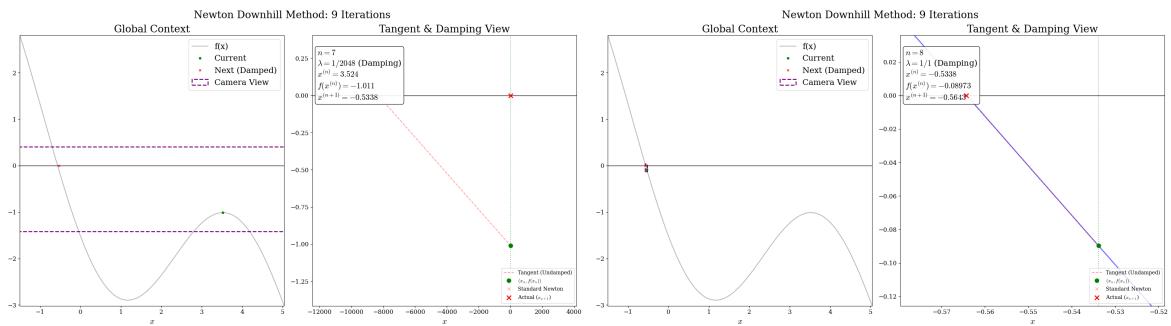


图 12: Newton 下山法-最后两步

4.5 高斯消元法

4.5.1 输入

我们用高斯消元法求解以下线性方程组:

$$\begin{bmatrix} 10 & -19 & -2 \\ -20 & 40 & 1 \\ 1 & 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix} = \begin{bmatrix} 3 & 1 \\ 4 & 2 \\ 5 & 3 \end{bmatrix}$$

4.5.2 输出

高斯消元实在是非常代数的东西，所以我就没有做可视化。下面展示的是求解过程，可以看到六个未知数均获得了正确的解。

```

Solution Trace:
    Step 0: Initial Augmented Matrix
[[ 10. -19. -2. 3. 1.]
 [-20. 40. 1. 4. 2.]
 [ 1. 4. 5. 5. 3.]]]

    Step 1: Pivoting: Swapped Row 0 and Row 1
[[-20. 40. 1. 4. 2.]
 [ 10. -19. -2. 3. 1.]
 [ 1. 4. 5. 5. 3.]]]

    Step 2: Elimination: Cleared column 0 below pivot
[[-20. 40. 1. 4. 2.]
 [ 0. 1. -1.5 5. 2.]
 [ 0. 6. 5.05 5.2 3.1]]]

    Step 3: Pivoting: Swapped Row 1 and Row 2
[[-20. 40. 1. 4. 2.]
 [ 0. 6. 5.05 5.2 3.1]
 [ 0. 1. -1.5 5. 2.]]]

    Step 4: Elimination: Cleared column 1 below pivot
[[-20. 40. 1. 4. 2.]
 [ 0. 6. 5.05 5.2 3.1]
 [ 0. 0. -2.34 4.13 1.48]]]

Final Result:
[[ 4.41637011 1.96797153]
 [ 2.35231317 1.04982206]
 [-1.76512456 -0.63345196]]
Has Converged: True

Process finished with exit code 0

```

图 13: 高斯消元法求解过程

5 实验总结

5.1 一元方程的迭代解法

通过本次实验，我加深了对各种一元方程迭代解法的理解。尤其是通过可视化，我能够直观地观察到各个方法的迭代过程，从而更好地理解它们的收敛性和适用范围。

二分法作为一种简单而有效的方法，在函数连续且易于找到有根区间的情况下表现出色。它对于函数性质要求很低，只是找到有根区间本身也不是容易的事情。

Aitken 加速收敛法对函数的要求较高，我尝试的许多函数都无法收敛。对于能够收敛的函数，它的收敛速度非常快——不过这也可能是因为这些函数本身性质就非常好。

Newton 法和 Newton 下山法在处理复杂函数时表现出色。即便我给出了对它们很不利的函数和初值，他们仍然顽强地挣扎，最终成功收敛到正确的结果，令我赞叹不已。

5.2 线性方程组的消元法

通过实现高斯消元法，我加深了对线性方程组求解方法的理解。虽然高斯消元法的时间复杂度较高，但它非常契合人类的思维方式，而且作为一种基础的求解方法，在小规模和对精度要求高的问题中仍然具有实用价值，毕竟它是没有方法误差的。

5.3 其他方面的一些思考

本次实验也是对我编程能力的一次锻炼。数值计算方法的算法比较清晰，实现起来相对直接；但可视化实在搞得我很烦躁。数值算法基本上是我手搓的，但我必须承认可视化器代码中包含大量经过我修改调整的 AI 生成代码。

编程中也遇到不少阻挠，例如对蟒蛇语言本身性质不熟悉导致各种神秘错误。我讨厌动态类型语言，静态检查能过不代表运行时不报错，运行时不报错的代码可能静态检查不通过，烦人。代码高亮也很差。

此外，我的代码没有考虑超出定义域的情况，所有的求解过程都是假定函数的定义域足够大的条件下进行的。如果输入的函数包含 `log` 之类的表达式，很容易就会报 `MathDomainError` 而爆炸。这使我越发敬佩编写 `numpy`、`scipy` 等库的大佬们了。