

HW8

白博臣、何骐多、夏营

1 Problem1

1.1 二阶

令

$$x_n = x_0 + nh$$

二阶向前差分公式:

$$f''(x_0) = \frac{(f(x_0) - 2f(x_1) + f(x_2)))}{h^2} + O(h)$$

二阶中心差分公式:

$$f''(x_0) = \frac{(f(x_0 - 1)) - 2f(x_0) + f(x_1))}{h^2} + O(h^2)$$

二阶向后差分公式:

$$f''(x_0) = \frac{(f(x_0 - 2)) - 2f(x_0 - 1) + f(x_0))}{h^2} + O(h)$$

1.2 三阶

三阶向前差分公式:

$$f'''(x_0) = \frac{(-f(x_0) + 3f(x_1) - 3f(x_2) + f(x_3)))}{h^3} + O(h)$$

三阶中心差分公式:

$$f'''(x_0) = \frac{(-f(x_0 - 2)) + 2f(x_0 - 1) - 2f(x_1) + f(x_2))}{2h^3} + O(h^2)$$

三阶向后差分公式:

$$f'''(x_0) = \frac{(-f(x_0 - 3)) + 3f(x_0 - 2) - 3f(x_0 - 1) + f(x_0))}{h^3} + O(h)$$

1.3 四阶

四阶向前差分公式：

$$f^{iv}(x_0) = \frac{(f(x_0) - 4f(x_1) + 6f(x_2) - 4f(x_3) + f(x_4))}{h^4} + O(h)$$

四阶中心差分公式：

$$f^{iv}(x_0) = \frac{(f(x_{-2}) - 4f(x_{-1}) + 6f(x_0) - 4f(x_1) + f(x_2))}{h^4} + O(h^2)$$

四阶向后差分公式：

$$f^{iv}(x_0) = \frac{(f(x_{-4}) - 4f(x_{-3}) + 6f(x_{-2}) - 4f(x_{-1}) + f(x_0))}{h^4} + O(h)$$

1.4 五阶

三阶向前差分公式：

$$f^v(x_0) = \frac{(f(x_5) - 5f(x_4) + 10f(x_3) - 10f(x_2) + 5f(x_1) - f(x_0))}{h^5} + O(h)$$

三阶中心差分公式：

$$f^v(x_0) = \frac{(-f(x_{-3}) + 4f(x_{-2}) - 5f(x_{-1}) + 5f(x_1) - 4f(x_2) + f(x_3))}{2h^5} + O(h^2)$$

三阶向后差分公式：

$$f^v(x_0) = \frac{(f(x_{-5}) - 5f(x_{-4}) + 10f(x_{-3}) - 10f(x_{-2}) + 5f(x_{-1}) - f(x_0))}{h^5} + O(h)$$

2 Problem2

利用 $o(h^2)$ 的中心差分公式计算一阶导数，使用 Java 代码实现如下

```

1  import static java.lang.StrictMath.*;
2
3  interface function {
4      double cal(double x);
5  }
6
7  public class Derivative {
8      public static void main(String[] args) {
9          int n = 4;
10         double[] h = new double[n];
11         for (int i = 0; i < n; i++) {
12             h[i] = pow(10, -1 * (i + 1));
13         }
14         function f = (x) -> x * exp(x);
15         function d1f = (x) -> (x + 1) * exp(x);
16         double f01, f02, f11, f12, d1f1;
17         double d1f0 = d1f.cal(2);
18
19         for (double v:h) {
20             f01 = f.cal(2 + v);
21             f02 = f.cal(2 + 2 * v);
22             f11 = f.cal(2 - v);
23             f12 = f.cal(2 - 2 * v);
24             d1f1 = (-f02 + 8 * f01 - 8 * f11 + f12) / 12 / v;
25
26             double err = abs((d1f1 - d1f0) / d1f0);
27
28             System.out.printf("h = %f\n",v);
29             System.out.printf("f'0(2.0)   = %.16f, f'(2.0)   = %.16f, relative
↪ error = %.16f\n",d1f0, d1f1, err);
30         }
31     }
32 }
33

```

运行结果如下

```
h = 0.100000
f'(2.0) = 22.1671682967919500, f'(2.0) = 22.1669956213999000, relative error = 0.0000077896910304
h = 0.010000
f'(2.0) = 22.1671682967919500, f'(2.0) = 22.1671682795501500, relative error = 0.000000007778079
h = 0.001000
f'(2.0) = 22.1671682967919500, f'(2.0) = 22.1671682967887020, relative error = 0.0000000000001465
h = 0.000100
f'(2.0) = 22.1671682967919500, f'(2.0) = 22.1671682968294100, relative error = 0.0000000000016899
```

图 1: Derivative 运行结果

据计算，在 $h = 0.001$ 时，相对误差较小，是该范围内的最佳。

当 h 缩小时，相对误差会先缩小在增大，当 $h < 10^{-4}$ 时，相对误差会随 h 缩小而增大。

3 Problem3

利用 $o(h^2)$ 的中心差分公式计算二阶导数，使用 Java 代码实现如下

```

1  import static java.lang.StrictMath.*;
2
3  interface function {
4      double cal(double x);
5  }
6
7  public class Derivative {
8      public static void main(String[] args) {
9          int n = 4;
10         double[] h = new double[n];
11         for (int i = 0; i < n; i++) {
12             h[i] = pow(10, -1 * (i + 1));
13         }
14         function f = (x) -> x * exp(x * x);
15         function d2f = (x) -> (4 * x * x * x + 6 * x) * exp(x * x);
16         double f00, f01, f02, f11, f12, d2f1;
17         double d2f0 = d2f.cal(2);
18
19         for (double v : h) {
20             f00 = f.cal(2);
21             f01 = f.cal(2 + v);
22             f02 = f.cal(2 + 2 * v);
23             f11 = f.cal(2 - v);
24             f12 = f.cal(2 - 2 * v);
25             d2f1 = (-f02 + 16 * f01 - 30 * f00 + 16 * f11 - f12) / 12 / v / v;
26
27             double err = abs((d2f1 - d2f0) / d2f0);
28
29             System.out.printf("h = %f\n", v);
30             System.out.printf("f'0(2.0) = %.16f, f'(2.0) = %.16f, relative
↪ error = %.16f\n", d2f0, d2f1, err);
31         }
32     }
33 }
34

```

运行结果如下

```
h = 0.100000
f'(2.0) = 2402.3186014583460000, f'(2.0) = 2399.4975364303480000, relative error = 0.0011743092803284
h = 0.010000
f'(2.0) = 2402.3186014583460000, f'(2.0) = 2402.3183294825760000, relative error = 0.0000001132138636
h = 0.001000
f'(2.0) = 2402.3186014583460000, f'(2.0) = 2402.3186015197000000, relative error = 0.000000000255395
h = 0.000100
f'(2.0) = 2402.3186014583460000, f'(2.0) = 2402.3186167454470000, relative error = 0.0000000063634776
```

图 2: Derivative 运行结果

据计算，在 $h = 0.001$ 时，相对误差较小，是该范围内的最佳。

当 h 缩小时，相对误差会先缩小在增大，当 $h < 10^{-4}$ 时，相对误差会随 h 缩小而增大。

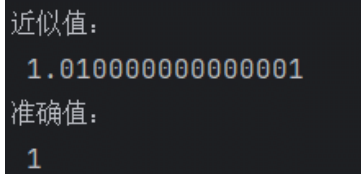
4 Problem4

4.1 a

使用 Python 代码实现如下

```
1 # 设置函数f(x)=x(x-1)
2 def f(x):
3     return x * (x - 1)
4
5
6 # 设置求解导数的公式
7 def derivative_f(x, delta):
8     return (f(x + delta) - f(x)) / delta
9
10
11 # 设置导数函数 f'(x)=2x-1
12 def real_df(x):
13     return 2 * x - 1
14
15
16 # 计算并且输出结果
17 approximation_df = derivative_f(1, 1e-2)
18 exact_df = real_df(1)
19
20 print('近似值: ', '\n', approximation_df)
21 print('准确值: ', '\n', exact_df)
22
```

运行结果如下



```
近似值:
1.0100000000000001
准确值:
1
```

图 3: HW8.4.a 运行结果

结论:

近似值与准确值并不完全相同, 这是因为此时的 δ 的取值 $\delta = 10^{-2}$ 并没有达到 $\lim \delta \rightarrow 0$, 所以此时是用的函数上 $x = 0.5$ 与 $x = 0.5 + \delta$ 两点的割线斜率来代替的导数值, 与实际结果会存在误差。

4.2 b

加入一个 for 循环语句来实现对多个 δ 值的运算, 使用 Python 代码实现如下

```
1 import numpy as np
2
3
4 # 设置函数f(x)=x(x-1)
5 def f(x):
6     return x * (x - 1)
7
8
9 # 设置求解导数的公式
10 def derivative_f(x, delta):
11     return (f(x + delta) - f(x)) / delta
12
13
14 # 设置导数函数 f'(x)=2x-1
15 def real_df(x):
16     return 2 * x - 1
17
18
19 # 设置不同的delta值来计算导数并输出结果
20 log_delta = np.linspace(-2, -18, 9)
21 delta = 10 ** log_delta
22 exact_df = real_df(1)
23 print('准确值为', '\n', exact_df)
24 for i in range(len(delta)):
25     approximation_df = derivative_f(1, delta[i])
26     print('当delta={}时'.format(delta[i]), '\n', approximation_df)
27
```

运行结果如下

结论:

当 δ 值在 $10^{-2}, 10^{-8}$ 中时, 随着 δ 的减小, 近似值逐渐逼近真实值, 而在 $\delta > 10^{-8}$ 后,


```
准确值为
1
当delta=0.01时
1.0100000000000001
当delta=0.0001时
1.0000999999998899
当delta=1e-06时
1.0000009999177333
当delta=1e-08时
1.000000039225287
当delta=1e-10时
1.000000082840371
当delta=1e-12时
1.0000889005833413
当delta=1e-14时
0.9992007221626509
当delta=1e-16时
0.0
当delta=1e-18时
0.0
```

图 4: HW8.4.b 运行结果

随着 δ 的减小, 近似值与真实值的误差逐渐增大, 当 $\delta > 10^{-16}$ 时, 近似值为 0。

现对上述异常现象做出解释。

利用计算机并通过公式 $\frac{df(x)}{dx} = \lim_{\delta \rightarrow 0} \frac{(f(x+\delta) - f(x))}{\delta}$ 估计近似值时, 其同时会产生两种误差, 一类是截断误差 (truncation error), 其与 δ 的值成正相关; 另一类是由于算法问题以及机器精度导致的舍入误差 (round-off error), 其与机器精度成正相关, 与 δ 成负相关。这两类误差的和组成了这一计算公式导致的总误差, 并且由于两类误差与 δ 的相关性不同, 这就导致当 δ 减小时, 截断误差在减小, 而舍入误差在增大。结合 lecture 8 可以写出两类误差具体的表达式。

截断误差:

$$e_{tru} = \delta/2f''(x)$$

舍入误差:

$$e_{rou} = (2cf(x))/\delta$$

其中 c 为机器精度

于是总误差为:

$$e = e_{tru} + e_{rou} = \delta/2f''(x) + (2cf(x))/\delta$$

当 δ 值在 $10^{-2}, 10^{-8}$ 中时, 截断误差的影响占主导地位, 所以此时总误差会随着 δ 的减小而减小, 而当 $\delta > 10^{-8}$ 时, 截断误差趋近于 0, 舍入误差的影响占主导地位了, 所以此时总误差会随着 δ 的减小而增大。而当 $\delta > 10^{-16}$ 时, 此时, δ 接近于甚至小于机器精度 c 值, 于是 $f(x + \delta) - f(x)$ 在计算机的运算过程被直接舍入为 0 了, 所以此时近似值为 0。

4.3 c

首先生成一个在区间 $[0, 20]$ 有 200 项的等差数列 $\log_{10}h$, 而后复用上一问代码, 计算出不同 h 值下的近似值, 而后与真实值做差得到误差。再对误差做一次 \log_{10} 的运算, 再利用题意所给的

$$g(h) = \frac{h}{2} \sin 0.5 + \epsilon'' \frac{2}{h} \sin 0.5$$

画出所需的 $\log_{10} - \log_{10}$ 图像。

使用 Python 代码实现如下

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 # 设置函数f(x)=sin(x)
6 def f(x):
7     return np.sin(x)
8
9
10 # 设置求解导数的公式
11 def derivative_f(x, delta):
12     return (f(x + delta) - f(x)) / delta
13
14
```

```

15 # 设置导数函数 f'(x)=cos(x)
16 def real_df(x):
17     return np.cos(x)
18
19
20 # 设置g(h)=h*sin(0.5)/2+2c*sin(0.5)/h
21 def g(h, x):
22     return h * f(x) / 2 + c * 2 * f(x) / h
23
24
25 # 设置h
26 log_h = np.linspace(0, -20, 200)
27 h = 10 ** log_h
28
29 # 机器精度
30 c = 6 * 10 ** -16
31
32 # 计算误差值
33 error = np.array([])
34 for i in range(len(h)):
35     temp = abs(derivative_f(0.5, h[i]) - real_df(0.5))
36     error = np.append(error, np.log10(temp))
37
38 # 绘制图像
39 plt.figure()
40 plt.plot(log_h, np.log10(g(h, 0.5)))
41 plt.scatter(log_h, error, s=5, color='red')
42 plt.show()
43
44 # 计算最佳的h, 等价于g'(h)=0时的h的值
45 h0=np.sqrt(4*c)
46 print(h0)
47

```

运行结果如下

与示意图相同：重新绘图（Reproduce）（ $\epsilon'' = 6 \times 10^{-16}$ ）最佳 h 值为：

$$h = 4.898979485566356 \times 10^{-8}$$

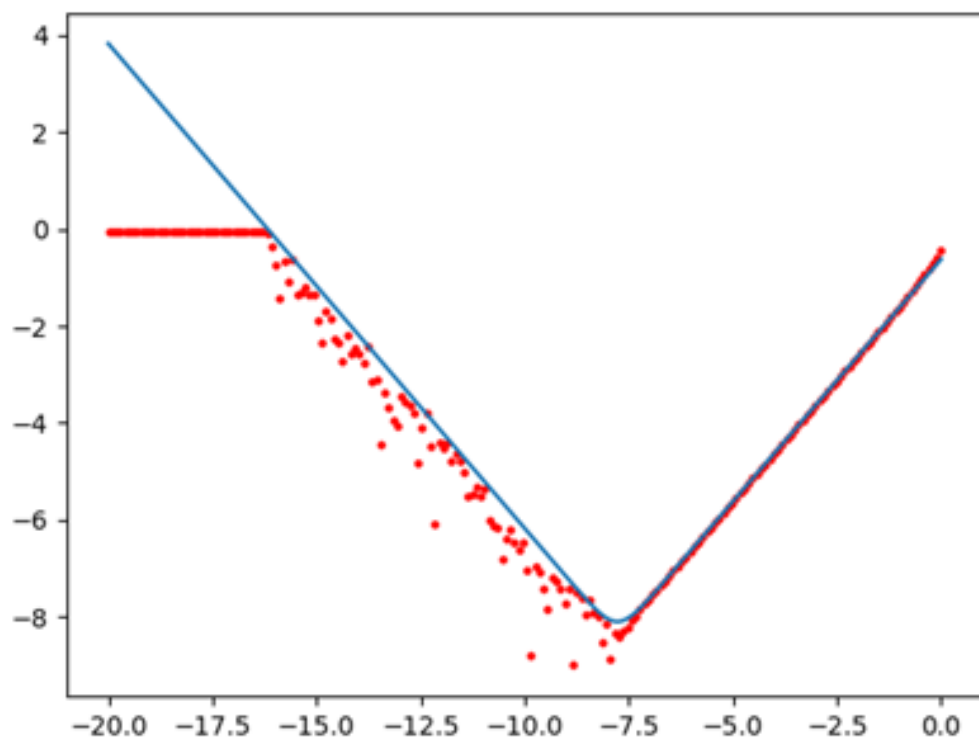


图 5: HW8.4.c 运行结果

4.4 d

如 (b) 问结论部分的叙述, 可以给出两种误差的表达式:

截断误差:

$$e_{tru} = h/2f''(x)$$

舍入误差:

$$e_{rou} = (2cf(x))/h$$

其中 c 为机器精度

进而写出总误差:

$$e = e_{tru} + e_{rou} = h/2f''(x) + (2cf(x))/h$$

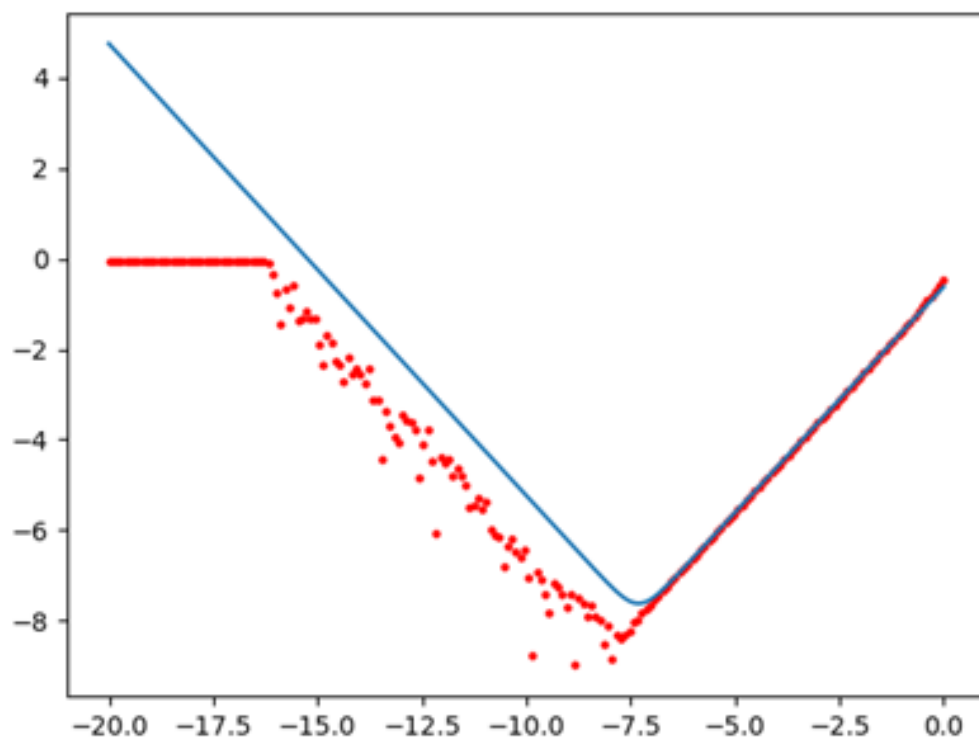


图 6: HW8.4.c 运行结果

可以发现总误差 e 存在一个最小值，而使 e 取到这一最小值的 h 值即为最佳 h 值，也即 h_{best} 。

由均值不等式有：

$$\frac{h}{2}f''(x) + \frac{2cf(x)}{h} \geq 2\sqrt{\frac{cf(x)}{f''(x)}}$$

取等条件为：

$$\frac{h}{2}f''(x) = \frac{2cf(x)}{h}$$

进而求得 h_{best} ：

$$h_{best} = \sqrt{\frac{4cf(x)}{|f''(x)|}}$$

使用 Python 代码实现如下

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4
5 # 设置函数f(x)=sin(x)
6 def f(x):
7     return np.sin(x)
8
9
10 # 设置求解导数的公式
11 def derivative_f(x, delta):
12     return (f(x + delta) - f(x)) / delta
13
14
15 # 设置导数函数 f'(x)=cos(x)
16 def real_df(x):
17     return np.cos(x)
18
19
20 # 设置h
21 log_h = np.linspace(0, -20, 200)
22 h = 10 ** log_h
23
24 # 机器精度
25 c = 6 * 10 ** -16
26
27
28 # 计算截断误差
29 def e_tru(h, x):
30     return h * f(x) / 2
31
32
33 # 计算舍入误差
34 def e_rou(h, x):
35     return c * 2 * f(x) / h
36
37
38 # 计算总误差
39 def g(h, x):
```

```

40     return e_rou(h, x) + e_tru(h, x)
41
42
43 # 计算h的最佳取值
44 def h_best(x):
45     return np.sqrt(4 * c)
46
47
48 # 计算各种误差
49 e_truncation = e_tru(h, 0.5)
50 e_roundoff = e_rou(h, 0.5)
51 e = g(h, 0.5)
52
53 # 绘制图像与输出结果
54 plt.figure()
55 plt.plot(log_h, np.log10(e_truncation), label='truncation error')
56 plt.plot(log_h, np.log10(e_roundoff), label='round-off error')
57 plt.plot(log_h, np.log10(e), label='total error')
58 plt.xlabel('log10(h)')
59 plt.legend()
60 plt.show()
61
62 # 计算并输出h_best
63 print('h的最佳取值为: ', '\n', h_best(0.5))
64

```

以 (c) 给出的机器精度:

$$c = 6 \times 10^{-16}$$

为例。

h 的最佳取值为:

$$4.898979485566356e - 08$$

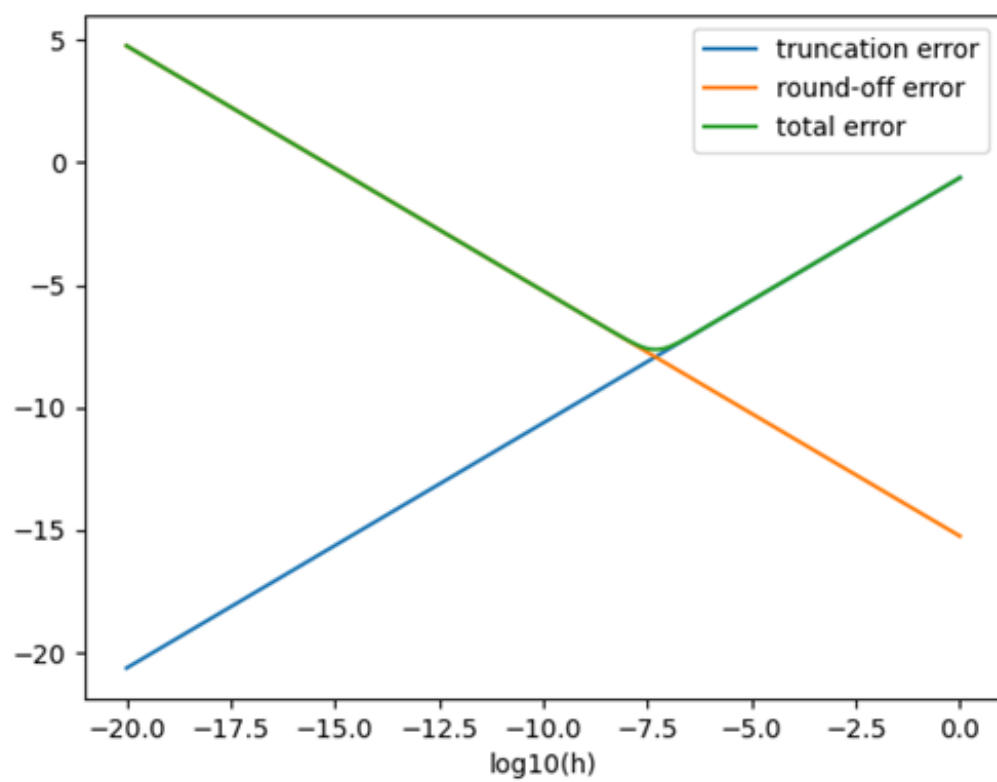


图 7: HW8.4.d 运行结果