

数值分析笔记

Python version

Jiaqi Z.

2023 年 9 月 25 日

目录

1	绪论	5
1.1	误差	5
1.1.1	误差来源与分类	5
1.1.2	误差概念	6
1.1.3	相对误差限和有效数字的关系	9
1.2	数值运算的误差估计	10
1.2.1	四则运算误差估计	10
1.2.2	函数值误差估计	11
1.3	算法数值稳定性	12
1.4	数值计算中应该注意的一些原则	15
1.4.1	避免两相近数相减	15
1.4.2	避免除数绝对值远小于被除数绝对值	15
1.4.3	避免大数“吃”小数	15
1.4.4	简化计算步骤, 避免误差积累	17
2	插值法	21
2.1	引言	21
2.2	Lagrange 插值法	23
2.2.1	线性插值	23
2.2.2	抛物插值	23
2.2.3	Lagrange 插值多项式	24
2.2.4	插值余项	26
2.2.5	Lagrange 插值优缺点	28
2.3	Newton 插值	29

2.3.1	Newton 插值	29
2.3.2	差商	30
2.3.3	Newton 插值余项	32

Chapter 1

绪论

1.1 误差

1.1.1 误差来源与分类

1. (模型误差): 从实际模型中抽象出数学模型;

例如, 一个质量为 m 的小球做自由落体运动, 则位置 s 与时间 t 的关系式满足:

$$m \frac{d^2 s}{dt^2} = mg$$

不难想见, 该式仅在不考虑阻力时成立.

2. (观测误差): 通过测量得到模型中参数的值;
3. (方法误差 (或称截断误差)): 求近似解时所引入的误差;

例 1.1.1. 考虑函数 $f(x)$ 做 *Taylor* 多项式展开所导致的截断误差.

解. 对函数 $f(x)$ 计算 *Taylor* 多项式, 有

$$P_n(x) = f(0) + \frac{f'(0)}{1!}x + \frac{f''(0)}{2!}x^2 + \cdots + \frac{f^{(n)}(0)}{n!}x^n$$

由于有限项, 因此多项式有截断误差

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}x^{n+1}$$

其中, $\xi \in (x, 0)$

□

4. (舍入误差): 机器字长有限所引起的误差

其中, 方法误差和舍入误差是数值分析所重点考虑的误差, 同时, 方法误差是可以避免的.

1.1.2 误差概念

绝对误差与绝对误差限

定义 1.1.1 (绝对误差与绝对误差限). 设 x 是准确值, x^* 是 x 的一个近似值, 则称

$$e(x^*) = x^* - x$$

为 x^* 的绝对误差, 简称误差.

同时, 误差的绝对值的上限 $\varepsilon(x^*)$, 即有

$$|e(x^*)| = |x^* - x| \leq \varepsilon(x^*)$$

$\varepsilon(x^*)$ 称为绝对误差限.

注意: 误差有正有负, 而误差限恒为正值.

习惯上, 我们把精确值和测量值的关系表示为

$$x = x^* \pm \varepsilon$$

相对误差与相对误差限

定义 1.1.2 (相对误差与相对误差限). 设 x 为准确值, x^* 为近似值, 称

$$e_r^* = e_r^*(x^*) = \frac{e(x^*)}{x} = \frac{x^* - x}{x}$$

为近似值 x^* 的相对误差.

同时, 其绝对值的上限 ε_r^* , 即有

$$\left| \frac{x - x^*}{x} \right| \leq \varepsilon_r^*$$

ε_r^* 称为相对误差限.

可以证明, 当 e_r^* 较小时, 有

$$e_r^* \approx \frac{x^* - x}{x^*}$$

同时易得

$$\varepsilon_r^* = \frac{\varepsilon^*}{|x^*|}$$

有效数字

定义 1.1.3 (有效数字, 有效位数, 有效数). 若近似值 x^* 误差满足

$$|x - x^*| \leq \frac{1}{2} \times 10^{-n}$$

则称 x^* 近似表示 x 准确到小数点后第 n 位, 并从第 n 位起一直到最左边非零数字之间的一切数字称为有效数字, 位数为有效位数.

若所有数字均为有效数字, 则称为有效数

例 1.1.2. 考虑圆周率 π , 且有近似值 $\pi_1 = 3.14, \pi_2 = 3.1415, \pi_3 = 3.1416, \pi_4 = 3.14159$. 考虑它们的有效数字, 且判断是否为有效数.

解. 对于 $\pi_1 = 3.14$, 有 $|\pi - \pi_1| \approx 0.00159 \leq 0.5 \times 10^{-2}$, 即 π_1 精确到小数点后 2 位, 有效数字是 3 位, 是有效数.

同理, 有 $|\pi - \pi_2| \approx 0.0000926 \leq 0.5 \times 10^{-3}$, 即 π_2 精确到小数点后 3 位, 有效数字是 4 位, 不是有效数.

$|\pi - \pi_3| \approx 0.0000073 \leq 0.5 \times 10^{-4}$, 即 π_3 精确到小数点后 4 位, 有效数字是 5 位, 是有效数.

$|\pi - \pi_4| \approx 0.0000026 \leq 0.5 \times 10^{-5}$, 即 π_4 精确到小数点后 5 位, 有效数字是 6 位, 是有效数. \square

从上例中不难看出, 有效数通常是采取四舍五入所得到的近似值.

扩展: 我们可以简单给出关于四舍五入的证明.

证明. 设准确值为 x , 其近似值为 x^* , 考虑近似值精确到小数点后 n 位, 即

$$|x - x^*| \leq 5 \times 10^{-(n+1)}$$

若其为有效数, 则 x^* 为小数点后 n 位, 不妨设

$$x^* = a + b \cdot 10^{-n}$$

其中 $b \in [1, 10)$

特别地, 分两种情况讨论.

若 $x > x^*$, 即真实值大于近似值, 此时有

$$x \leq x^* + 5 \times 10^{-(n+1)} = a + b \cdot 10^{-n} + 5 \times 10^{-(n+1)}$$

即当小数点后第 $n+1$ 位小于等于 5 时, 舍去后面的数字可以得到有效数.

若 $x < x^*$, 即真实值小于近似值, 此时有

$$\begin{aligned} x &\geq x^* - 5 \times 10^{-(n+1)} = a + b \cdot 10^{-n} - 5 \times 10^{-(n+1)} \\ &= a + (b - 1) \cdot 10^{-n} + 5 \times 10^{-(n+1)} \end{aligned}$$

即当小数点后第 $n + 1$ 位大于等于 5 时, 进位可以得到有效数. \square

十进制浮点表示法

定义 1.1.4. 设 x^* 为任一十进制数, 则 x^* 可表示为

$$x^* = \pm 0.a_1 a_2 \cdots a_n \cdots \times 10^m$$

其中, a_1 为 1 到 9 之间的一个数字, $a_2 \cdots a_n$ 为 0 到 9 之间的一个数字, m 为整数. 这样表示的 x^* 称为十进制浮点数 (规格化浮点数).

有效数字的等价定义 (基于浮点表示法)

定义 1.1.5. 若近似值 $x^* = \pm 0.a_1 a_2 \cdots a_n a_{n+1} \cdots a_{n+p} \times 10^m (a_1 \neq 0)$ 的误差限是某一位上的半个单位, 即

$$|x - x^*| \leq \frac{1}{2} \times 10^{m-n} \quad (1.1)$$

则称 x^* 有 n 位有效数字.

例 1.1.3. 设 $x_1^* = 0.0051, x_2^* = 5.100$, 两数均为四舍五入得到, 求两个数字的有效位数.

解. 由于有

$$\begin{aligned} \varepsilon(x_1^*) &= 0.5 \times 10^{-4}, x_1^* = 0.51 \times 10^{-2} \\ \varepsilon(x_2^*) &= 0.5 \times 10^{-3}, x_2^* = 0.51 \times 10^1 \end{aligned}$$

可得

$$\begin{aligned} \varepsilon(x_1^*) &= 0.5 \times 10^{-2-2} \\ \varepsilon(x_2^*) &= 0.5 \times 10^{1-4} \end{aligned}$$

即, x_1^* 有两位有效数字, x_2^* 有四位有效数字. \square

例 1.1.4. 设 $x_1^* = 2.180, x_2^* = 10.210$, 均具有四位有效数字, 求绝对误差限和相对误差限.

解. 对 x_1^* , 有

$$x_1^* = 0.2180 \times 10^1$$

即 $m = 1$, 且具有四位有效数字, 即 $n = 4$, 则根据公式 (1.1), 有

$$\varepsilon(x_1^*) = 0.5 \times 10^{1-4} = 0.5 \times 10^{-3}$$

其相对误差限为

$$\varepsilon_r(x_1^*) = \frac{\varepsilon(x_1^*)}{|x_1^*|} = 0.023\%$$

同理可得, 对于 x_2^* , 有

$$\varepsilon(x_2^*) = 0.5 \times 10^{-2}, \varepsilon_r(x_2^*) = 0.049\%$$

□

1.1.3 相对误差限和有效数字的关系

关于有效数字和相对误差限之间的关系, 有如下定理.

定理 1.1.1. 对于用式 (1.1.4) 表示的近似数 x^* , 若 x^* 具有 n 位有效数字, 则其相对误差限为

$$\varepsilon_r^* \leq \frac{1}{2a_1} \times 10^{-n}$$

证明. 由式 1.1.4 可得

$$a_1 \times 10^m \leq |x^*| \leq (a_1 + 1) \times 10^m$$

当 x^* 有 n 位有效数字时, 有

$$|x - x^*| = |x^*| \varepsilon_r^* \leq (a_1 + 1) \times 10^m \times \frac{1}{2(a_1 + 1)} \times 10^{-n} = 0.5 \times 10^{m-n}$$

故 x^* 有 n 位有效数字. □

上述定理表明: 有效位数越多, 相对误差限越小.

例 1.1.5. 令 $\sqrt{20}$ 的近似值相对误差限小于 0.1% , 则需要取多少位有效数字?

解. 由定理 1.1.1 可知

$$\varepsilon_r^* \leq \frac{1}{2a_1} \times 10^{-n}$$

由于 $\sqrt{20} \approx 4.4$, 故 $a_1 = 4$, 只需要取 $n = 4$, 有

$$\varepsilon_r^* \leq 0.125 \times 10^{-3} < 10^{-3} = 0.1\%$$

即只需要对 $\sqrt{20}$ 的近似值取 4 位有效数字, 其相对误差限就可以小于 0.1%, 此时有

$$\sqrt{20} \approx 4.472.$$

□

1.2 数值运算的误差估计

1.2.1 四则运算误差估计

两个近似数分别为 x_1^* 和 x_2^* , 误差限分别为 $\varepsilon(x_1^*), \varepsilon(x_2^*)$, 进行四则运算的误差限分别为:

$$\begin{aligned}\varepsilon(x_1^* \pm x_2^*) &= \varepsilon(x_1^*) + \varepsilon(x_2^*) \\ \varepsilon(x_1^* x_2^*) &\approx |x_1^*| \varepsilon(x_2^*) + |x_2^*| \varepsilon(x_1^*) \\ \varepsilon(x_1^* / x_2^*) &\approx \frac{|x_1^*| \varepsilon(x_2^*) + |x_2^*| \varepsilon(x_1^*)}{|x_2^*|^2}\end{aligned}$$

下面试着给出加减法误差的证明, 对于乘法和除法的证明, 将在后面给出.

证明.

$$\begin{aligned}|e(x_1^* \pm x_2^*)| &= |(x_1^* \pm x_2^*) - (x_1 \pm x_2)| \\ &= |(x_1^* - x_1) \pm (x_2^* - x_2)| \\ &\leq |x_1^* - x_1| + |x_2^* - x_2| \\ &\leq \varepsilon(x_1^*) + \varepsilon(x_2^*)\end{aligned}$$

□

1.2.2 函数值误差估计

一元函数误差估计

设 $f(x)$ 是一元函数, x 的近似值为 x^* , 以 $f(x^*)$ 近似 $f(x)$, 其误差限记作 $\varepsilon(f(x^*))$, 可用 Taylor 展开

$$f(x) - f(x^*) = f'(x^*)(x - x^*) + \frac{f''(\xi)}{2}\varepsilon^2(x^*)$$

其中, ξ 介于 x, x^* 之间, 取绝对值有

$$|f(x) - f(x^*)| \leq |f'(x^*)|\varepsilon(x^*) + \frac{|f''(\xi)|}{2}\varepsilon^2(x^*)$$

假定 $f'(x^*)$ 与 $f''(x^*)$ 的比值不大, 可忽略 $\varepsilon(x^*)$ 的高阶项, 于是可得误差限为

$$\varepsilon(f(x^*)) \approx |f'(x^*)|\varepsilon(x^*)$$

相对误差限为

$$\varepsilon_r(f(x^*)) \approx \frac{|f'(x^*)|\varepsilon(x^*)}{|f(x^*)|} = C_p(f, x^*)\varepsilon_r(x^*)$$

其中,

$$C_p(f, x^*) = \frac{|x^* f'(x^*)|}{|f(x^*)|}$$

称为 $f(x^*)$ 的条件数.

多元函数误差估计

当 f 为多元函数时计算 $A = f(x_1, x_2, \dots, x_n)$, 如果 x_1, x_2, \dots, x_n 的近似值为 $x_1^*, x_2^*, \dots, x_n^*$, 则 A 的近似值为 $A^* = f(x_1^*, x_2^*, \dots, x_n^*)$, 于是函数值 A^* 的误差 $e(A^*)$ 由 Taylor 展开, 得

$$\begin{aligned} e(A^*) &= A^* - A = f(x_1^*, x_2^*, \dots, x_n^*) - f(x_1, x_2, \dots, x_n) \\ &\approx \sum_{k=1}^n \left(\frac{\partial f(x_1^*, x_2^*, \dots, x_n^*)}{\partial x_k} \right) (x_k^* - x_k) = \sum_{k=1}^n \left(\frac{\partial f}{\partial x_k} \right)^* e_k^* \end{aligned}$$

于是误差限为

$$\varepsilon(A^*) \approx \sum_{k=1}^n \left| \left(\frac{\partial f}{\partial x_k} \right)^* \right| \varepsilon(x_k^*) \quad (1.2)$$

而 A^* 的相对误差限为

$$\varepsilon_r^* = \varepsilon_r(A^*) = \frac{\varepsilon(A^*)}{|A^*|} \approx \sum_{k=1}^n \left| \left(\frac{\partial f}{\partial x_k} \right)^* \right| \frac{\varepsilon(x_k^*)}{|A^*|}$$

例 1.2.1. 已测得某场地长 l 的值为 $l^* = 110 \text{ m}$, 宽 d 的值为 $d^* = 80 \text{ m}$, 已知 $|l - l^*| \leq 0.2 \text{ m}$, $|d - d^*| \leq 0.1 \text{ m}$, 试求面积 $S = ld$ 的绝对误差限与相对误差限.

解. 因为 $S = ld$, $\frac{\partial S}{\partial l} = d$, $\frac{\partial S}{\partial d} = l$, 由式 1.2 可知

$$\varepsilon(S^*) \approx \left| \left(\frac{\partial S}{\partial l} \right)^* \right| \varepsilon(l^*) + \left| \left(\frac{\partial S}{\partial d} \right)^* \right| \varepsilon(d^*)$$

其中,

$$\left(\frac{\partial S}{\partial l} \right)^* = d^* = 80 \text{ m}, \left(\frac{\partial S}{\partial d} \right)^* = l^* = 110 \text{ m}$$

而

$$\varepsilon(l^*) = 0.2 \text{ m}, \varepsilon(d^*) = 0.1 \text{ m}$$

于是绝对误差限为

$$\varepsilon(S^*) \approx (80 \times 0.2 + 110 \times 0.1) \text{ m}^2 = 27 \text{ m}^2$$

相对误差限为

$$\varepsilon_r(S^*) = \frac{\varepsilon(S^*)}{|S^*|} = \frac{\varepsilon(S^*)}{l^* d^*} \approx \frac{27}{8800} = 0.31\%$$

□

注意: 绝对误差限有量纲, 而相对误差限没有量纲.

1.3 算法数值稳定性

定义 1.3.1 (数值稳定). 一个算法如果初始数值有微小扰动 (即有误差), 而计算过程中舍入误差不增长, 使得结果产生微小误差. 则称该算法为数值稳定的. 反之称为数值不稳定.

例 1.3.1. 计算定积分

$$I_n = \int_0^1 \frac{x^n}{n+5} dx, n = 0, 1, 2, \dots, 8$$

解. 对被积函数变形, 得

$$\begin{aligned} I_n &= \int_0^1 \frac{(x+5)-5}{x+5} x^{n-1} dx \\ &= \int_0^1 x^{n-1} dx - 5 \int_0^1 \frac{x^{n-1}}{x+5} dx \\ &= \frac{1}{n} - 5I_{n-1} \end{aligned}$$

其中, $n = 1, 2, \dots, 8$.

易知, $I_0 = \ln 6 - \ln 5 = \ln 1.2$, 由于机器只能计算小数, 取三位有效数字, 即 $\ln 1.2 \approx 0.182$.

分析上述积分, 可知, $0 < I_n < 0.2$, 且随着 n 增大, I_n 逐渐减小, 当 $n \rightarrow \infty$ 时, $I_n \rightarrow 0$.

迭代计算上述积分, 可得结果为:

$$I_0 = 0.182, I_1 = 0.09, I_2 = 0.05, I_3 = 0.083, I_4 = -0.17$$

$$I_5 = 1.03, I_6 = -5.0, I_7 = 25.14, I_8 = -125.59$$

可以发现, 该算法数值不稳定.

若对上述积分递推公式进行变形, 可得

$$I_{n-1} = \frac{1}{5n} - \frac{1}{5}I_n, n = 9, 8, \dots, 1$$

由于当 $n \rightarrow \infty$ 时, $I_n \rightarrow 0$, 因此当 n 充分大时, 可近似认为 $I_n = I_{n+1}$, 故有 $I_9 \approx I_0$, 将其代入并求解方程, 可得 $I_9 \approx 0.017$.

迭代计算, 可得结果为

$$I_0 = 0.182, I_1 = 0.088, I_2 = 0.058, I_3 = 0.043, I_4 = 0.034$$

$$I_5 = 0.028, I_6 = 0.024, I_7 = 0.021, I_8 = 0.019$$

该算法为数值稳定的.

分析二者的误差, 可得对于第一个算法, 其误差为

$$e_n = |I_n - I_n^*| = 5|e_{n-1}| = 5^n|e_0|$$

而对于第二个算法, 其误差为

$$|e_{n-1}| = |I_{n-1} - I_{n-1}^*| = \frac{1}{5}|e_n| = \left(\frac{1}{5}\right)^n |e_9|$$

□

通过上述例子, 可以看到对于同一个问题, 使用不同算法, 得到的误差结果可能有很大不同.

扩展: 考虑到数值分析需要结合计算机使用, 故在笔记的适当地方, 将给出代码以供参考 (注: 代码不唯一. 且考虑到算法的设计原则, 如无必要, 不会引入相应的库函数).

本例的运行代码如下所示:

```
1  # 验证数值稳定性(例题) Exercise1-1.py
2  # 方法1(数值不稳定)
3  def I1(n):
4      if n==0:
5          return 0.182
6      else:
7          return 1/n-5*I1(n-1)
8  # 方法2(数值稳定)
9  def I2(n):
10     if n==9:
11         return 0.017
12     else:
13         return 1/(5*(n+1))-(1/5)*I2(n+1)
14
15  for n in range(0,9):
16     print(f"I1_{n}_={I1(n)}")
17
18  for n in range(0,9):
19     print(f"I2_{n}_={I2(n)}")
```

定义 1.3.2 (良态与病态). 对于一个数学问题, 若初始数据有微小扰动 (即误差), 导致计算结果产生较小误差, 则称此问题是良态的, 否则称其为病态的.

注意: 良态和病态是针对数学问题本身的, 与算法无关.

1.4 数值计算中应该注意的一些原则

1.4.1 避免两相近数相减

使用两相近数相减, 将会导致有效数字损失. 下面的例子将有效说明这一点:

例 1.4.1. 计算函数 $y = \sqrt{x+1} - \sqrt{x}$ 在 $x = 1000$ 处的取值.

已知 y 的四位有效数字为 0.01580

解. 若选择直接相减, 则有 $y = \sqrt{1001} - \sqrt{1000} \approx 31.64 - 31.62 = 0.02$, 只有两位有效数字.

若选择对其进行变形, 令

$$y = \frac{1}{\sqrt{x+1} + \sqrt{x}}$$

则可得

$$y = \frac{1}{\sqrt{1001} + \sqrt{1000}} \approx \frac{1}{31.64 + 31.62} = 0.01581$$

有三位有效数字. □

注意: 在本例中, 使用第二种方法得到的只有三位有效数字, 这是因为第四位有效数字是 0 而不是 1.

1.4.2 避免除数绝对值远小于被除数绝对值

例如, $\frac{42}{0.01}$ 和 $\frac{42}{0.011}$ 的结果分别是 4200 和 3818.18, 明显可以发现, 除数只变化了 0.001, 结果变化了 381.82

1.4.3 避免大数”吃”小数

由于计算机字长是有限的, 在计算过程中需要考虑到对阶, 例如, 计算下面的式子:

$$10^9 + 1$$

在计算前首先需要将其规格化, 即将上式化为

$$0.1 \times 10^{10} + 0.1 \times 10^1$$

在计算机计算过程中, 需要进行对阶, 即将指数部分化为相同的. 在这里, 计算机将会做如下处理:

$$0.1 \times 10^{10} + 0.0000000001 \times 10^{10} = 0.1000000001 \times 10^{10}$$

同时, 考虑到计算机内部的小数存储是有长度限制的, 假设以 8 位为例, 则上式中的小数最后一位的 1 将被舍去, 从而得到结果为 0.1×10^{10} , 显然与结果不符.

下面的例子将详细说明这一点:

例 1.4.2. 用单精度 (浮点数保留 8 位小数) 计算

$$10^9 + 40 + 39 + \cdots + 1$$

解. 假设从左到右计算, 由于

$$10^9 + 40 = 0.1 \times 10^{10} + 0.4 \times 10^2 \approx 0.1 \times 10^{10}$$

会出现大数”吃”小数的情况.

假设从右到左计算, 则首先计算 $1 + 2 + \cdots + 40$, 不难得结果为 820, 即有

$$\begin{aligned} \text{原式} &= 820 + 10^9 = 0.82 \times 10^3 + 0.1 \times 10^{10} \\ &= 0.000000082 \times 10^{10} + 0.1 \times 10^{10} \\ &= 0.100000082 \times 10^{10} \approx 0.10000008 \times 10^{10} = 1000000800 \end{aligned}$$

显然误差较小. □

下面的代码将演示这一点

注意: 由于计算机内部的存储方式和实际计算有些许误差 (计算机采用二进制存储), 因此运行结果可能与理论分析不一样.

```

1 # 演示大数"吃"小数 Exercise1-2.py
2 import numpy as np
3 # 使用从左到右的计算方式, 会有很大误差
4 def example1():
5     result = np.float32(0)
6     result = result + np.float32(1e9)

```



```

7     for i in range(1,41):
8         result = result + np.float32(i)
9         print(f"从左到右计算结果为{result}")
10    # 使用从右到左的计算方式, 误差较小
11    def example2():
12        result = np.float32(0)
13        for i in range(1,41):
14            result = result + np.float32(i)
15            result = result + np.float32(1e9)
16        print(f"从右到左计算结果为{result}")
17    # 运行结果
18    example1()
19    example2()

```

1.4.4 简化计算步骤, 避免误差积累

例 1.4.3. 多项式求值: 给定 x , 求下列 n 次多项式的值:

$$P(x) = a_0 + a_1x + \cdots + a_nx^n$$

解. 若采用直接求和的方法, 则有

$$P_n(x) = a_0 + a_1x + a_2x \cdot x + \cdots + a_n \underbrace{x \cdot x \cdot x \cdots x}_{n \uparrow x}$$

一共需要 $\frac{n(n+1)}{2}$ 次乘法, n 次加法

若使用逐项求和, 即令

$$x^2 = x \cdot x, x^3 = x^2 \cdot x, \cdots x^n = x^{n-1} \cdot x$$

一共需要 $(2n-1)$ 次乘法, n 次加法

若采用秦九韶算法 (Horner 算法), 则可以将上式整理为

$$P_n(x) = a_0 + x(a_1 + x(a_2 + x(\cdots + x(a_{n-1} + a_nx) \cdots)))$$

一共需要 n 次乘法, n 次加法

□

可以明显发现, 使用秦九韶算法的效率明显优于其他两个算法.
对于秦九韶算法, 可以使用如下递推公式:

$$\begin{cases} S_n = a_n \\ S_k = xS_{k+1} + a_k, k = n-1, n-2, \dots, 0 \\ P_n(x) = S_0 \end{cases}$$

实际上机运行可以参考如下代码:

```

1  # 演示100000次多项式不同算法时间差异(假设每一项系数a_n都是2)
   Exercise1-3.py
2  import time
3  POWER = 100000
4  # 直接求和
5  def method1():
6      result = 0
7      x = 2
8      a = []
9      for i in range(0, POWER+1):
10         a.append(2)
11         start_time = time.time()
12         for i in range(0, POWER+1):
13             result = result + a[i]*x**i
14         end_time = time.time()
15         # print(result)
16         print(end_time-start_time)
17  # 使用逐项求和
18  def method2():
19      result = 0
20      x = [1, 2]
21      for i in range(2, POWER+1):
22         x.append(x[i-1]*2)
23      a = []
24      for i in range(0, POWER+1):
25         a.append(2)

```

```
26     start_time = time.time()
27     for i in range(0,POWER+1):
28         result = result + a[i]*x[i]
29     end_time = time.time()
30     # print(result)
31     print(end_time-start_time)
32 # 秦九韶算法
33 def method3():
34     s = 2
35     x = 2
36     start_time = time.time()
37     for i in range(1,POWER+1):
38         s = s*x+2
39         # s.append(s[i-1]*x+2)
40     end_time = time.time()
41     # print(s)
42     print(end_time-start_time)
43
44 method1()
45 method2()
46 method3()
```

注意： 在本地测试时，得到运行结果分别为 10.107, 0.264, 0.249(单位“秒”)。这个时间可能在不同计算机上会有所差距，但一般情况下，随着多项式次数的增加，时间差距会逐渐拉大。同时，三种算法的时间增长速率也会有明显差距。

Chapter 2

插值法

2.1 引言

定义 2.1.1 (插值法). 设函数 $y = f(x)$ 在区间 $[a, b]$ 上有定义, 且已知在点 $a \leq x_0 \leq x_1 < \cdots < x_n \leq b$ 上的值 y_0, y_1, \cdots, y_n , 若存在一简单函数 $P(x)$, 使

$$P(x_i) = y_i, i = 0, 1, \cdots, n \quad (2.1)$$

成立, 则称 $P(x)$ 为函数 $f(x)$ 的插值函数, 点 x_0, x_1, \cdots, x_n 为插值节点, 包括插值节点的区间 $[a, b]$ 称为插值区间, 求插值函数 $P(x)$ 的方法称为插值法.

定义 2.1.2 (多项式插值). 若 $P(x)$ 是次数不超过 n 的代数多项式, 即

$$P(x) = a_0 + a_1x + \cdots + a_nx^n \quad (2.2)$$

其中 a_i 为实数, 则称 $P(x)$ 为插值多项式, 相应的插值法称为多项式插值.

本章所讨论的主要内容是多项式插值.

在寻找插值多项式之前, 需要对其存在性与唯一性进行讨论¹. 给出如下定理:

定理 2.1.1. 对于给定互异节点 x_0, x_1, \cdots, x_n , 满足插值条件式 (2.1) 的 n 阶插值多项式 (2.2) 存在且唯一.

¹ 存在性表明插值多项式存在, 唯一性表明无论采用哪种插值方法, 得到的结果是唯一的.

证明. 设所要构造的插值多项式为

$$P_n(x) = a_0 + a_1x + \cdots + a_nx^n$$

由插值条件

$$P_n(x_i) = y_i, i = 0, 1, \cdots, n$$

得如下线性方程组

$$\begin{cases} 1 \cdot a_0 + x_0a_1 + \cdots + x_0^na_n = y_0 \\ 1 \cdot a_0 + x_1a_1 + \cdots + x_1^na_n = y_1 \\ \vdots \\ 1 \cdot a_0 + x_na_1 + \cdots + x_n^na_n = y_n \end{cases}$$

求解 a_0, a_1, \cdots, a_n , 计算系数行列式

$$D = \begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{vmatrix}$$

该行列式为 Vandermonde 行列式, 其值为

$$D = \prod_{0 \leq j < i \leq n} (x_i - x_j)$$

当 $x_i \neq x_j$ 时, $D \neq 0$, 即 $P_n(x)$ 由 a_0, a_1, \cdots, a_n 唯一确定 □

在实际计算过程中, 直接求解方程组往往计算量较大, 且方程组可能具有病态性. 例如, 对于 x_1, x_2, x_3, x_4 , 若值分别为 0.1, 0.2, 0.3, 0.4, 则行列式 $D = 1.2 \times 10^{-6} \approx 0$.

因此, 通常的做法是在 n 次多项式空间中寻找一组基函数

$$\varphi_0(x), \varphi_1(x), \cdots, \varphi_n(x)$$

使得

$$P_n(x) = a_0\varphi_0(x) + a_1\varphi_1(x) + \cdots + a_n\varphi_n(x)$$

不同的基函数对应不同的插值法. 本章重点讨论 Lagrange 插值法与 Newton 插值法.

2.2 Lagrange 插值法

2.2.1 线性插值

例 2.2.1. 对于节点 $(x_0, y_0), (x_1, y_1)$, 求一次多项式

解. 利用直线的两点式, 不难得到其插值多项式为

$$\begin{aligned} P_1 &= \left(\frac{x - x_1}{x_0 - x_1} \right) y_0 + \left(\frac{x - x_0}{x_1 - x_0} \right) y_1 \\ &= l_0(x)y_0 + l_1(x)y_1 = \sum_{i=0}^1 l_i(x)y_i \end{aligned}$$

□

在这里, 称

$$l_0(x) = \frac{x - x_1}{x_0 - x_1}, l_1(x) = \frac{x - x_0}{x_1 - x_0}$$

为一次 Lagrange 插值基函数.

不难验证, 对于一次 Lagrange 插值基函数而言, 存在如下性质

- $l_0(x), l_1(x)$ 均为一次多项式
- $l_0(x_0) = 1, l_1(x_0) = 0, l_0(x_1) = 0, l_1(x_1) = 1$

2.2.2 抛物插值

与线性插值类似, 对于抛物插值, 设有三个插值点 $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, 可得其插值多项式为

$$P_2(x) = y_0 l_0(x) + y_1 l_1(x) + y_2 l_2(x)$$

其中 $l_0(x), l_1(x), l_2(x)$ 均为二次多项式, 且有

$$l_0(x_0) = 1, l_1(x_0) = 0, l_2(x_0) = 0$$

$$l_0(x_1) = 0, l_1(x_1) = 1, l_2(x_1) = 0$$

$$l_0(x_2) = 0, l_1(x_2) = 0, l_2(x_2) = 1$$

2.2.3 Lagrange 插值多项式

将上述结论推广至 n 阶情况. 即假设有 $n+1$ 个节点 x_0, x_1, \dots, x_n 的 n 阶插值多项式 $L_n(x)$, 且满足条件

$$L_n(x_i) = y_i, i = 1, 2, \dots, n \quad (2.3)$$

类似于线性插值和抛物插值, 我们首先需要定义出基函数.

定义 2.2.1. 若 n 次多项式 $l_j(x), j = 0, 1, \dots, n$ 在 $n+1$ 个节点 $x_0 < x_1 < \dots < x_n$ 上满足条件

$$l_j(x_k) = \begin{cases} 1, & k = j \\ 0, & k \neq j \end{cases}, j, k = 0, 1, \dots, n$$

则称这 $n+1$ 个 n 次多项式 $l_0(x), l_1(x), \dots, l_n(x)$ 为节点 x_0, x_1, \dots, x_n 上的 n 次插值基函数.

利用其性质, 可以得到基函数形式为

$$l_k(x) = \frac{(x-x_0) \cdots (x-x_{k-1})(x-x_{k+1}) \cdots (x-x_n)}{(x_k-x_0) \cdots (x_k-x_{k-1})(x_k-x_{k+1}) \cdots (x_k-x_n)}, k = 0, 1, \dots, n$$

扩展: 下面将说明如何计算基函数的形式.

利用性质, 可知对于 $l_k(x), k = 0, 1, \dots, n$, 当 $x \neq x_k$ 时, 其函数值为 0. 则可以将其分解为若干因式 $(x-x_j), j = 0, 1, \dots, n$ 且 $j \neq k$, 即

$$l_k(x) = \lambda(x-x_0)(x-x_1) \cdots (x-x_{k-1})(x-x_{k+1}) \cdots (x-x_n), k = 0, 1, \dots, n$$

同时, 由于当 $x = x_k$ 时, $l_k(x_k) = 1$, 可得待定系数 λ 为

$$\lambda = \frac{1}{(x_k-x_0)(x_k-x_1) \cdots (x_k-x_{k-1})(x_k-x_{k+1}) \cdots (x_k-x_n)}, k = 0, 1, \dots, n$$

代入并整理, 可得基函数的具体形式为

$$l_k(x) = \frac{(x-x_0) \cdots (x-x_{k-1})(x-x_{k+1}) \cdots (x-x_n)}{(x_k-x_0) \cdots (x_k-x_{k-1})(x_k-x_{k+1}) \cdots (x_k-x_n)}, k = 0, 1, \dots, n$$

上式因此得证.

下面将试着给出基于 Lagrange 多项式插值的一个程序代码, 仅供参考.


```
1  # 使用拉格朗日多项式插值法的实例 Exercise2-1.py
2  # 假设四个插值点分别为 (1,2), (2,3), (3,6), (4,7)
3  # 实际运行时这些数据可以自行修改, 从而观察插值的实际作用.
4
5  import numpy as np
6  import matplotlib.pyplot as plt
7
8  def lagrange_interpolation(x, points):
9      n = len(points)
10     result = 0.0
11     for i in range(n):
12         xi, yi = points[i]
13         term = yi
14         for j in range(n):
15             if i != j:
16                 xj, yj = points[j]
17                 term *= (x - xj) / (xi - xj)
18         result += term
19     return result
20
21 x = [1,2,3,4]
22 y = [2,3,6,7]
23 plt.scatter(x,y,color="red")
24 points = list(zip(x,y))
25 x = np.arange(1,5,0.01)
26 result = lagrange_interpolation(x, points)
27 plt.plot(x,result)
28 plt.show()
```

使用这段代码运行的结果如图 2.1所示.

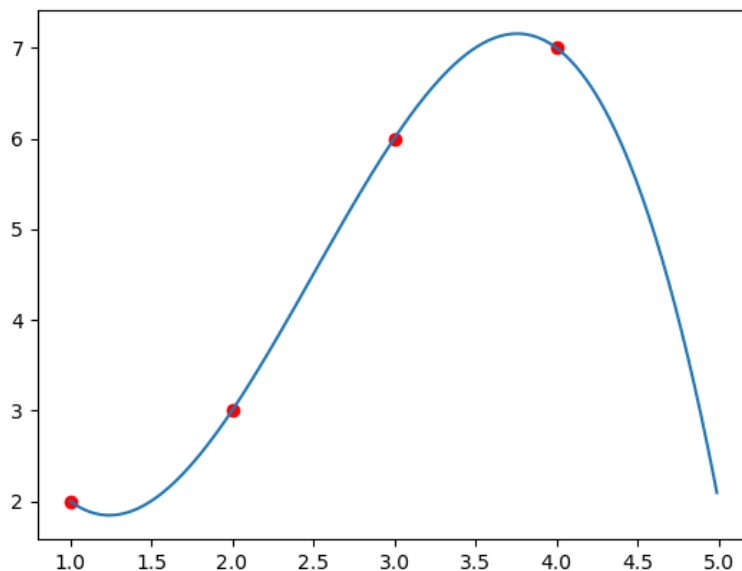


图 2.1: Lagrange 多项式插值 (使用上述代码生成)

2.2.4 插值余项

使用 $L_n(x)$ 近似表示 $f(x)$, 则会引起截断误差. 其截断误差为 $R_n(x) = f(x) - L_n(x)$. 通常会称 $R_n(x)$ 为插值多项式的余项或简称为插值余项.

为估计插值余项, 有如下定理:

定理 2.2.1. 设 $f^{(n)}(x)$ 在 $[a, b]$ 上连续, $f^{(n+1)}(x)$ 在 (a, b) 存在, 节点 $a \leq x_0 < x_1 < \cdots < x_n \leq b$, $L_n(x)$ 是满足条件式 (2.3) 的插值多项式, 则对于任何 $x \in [a, b]$, 插值余项

$$R_n(x) = f(x) - L_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^n (x - x_i) \quad (2.4)$$

其中, $\xi \in (a, b)$ 且依赖于 x

证明. 由插值条件可知, $R_n(x)$ 在节点 $x_k, k = 0, 1, \dots, n$ 上为 0, 即可

以做因式分解

$$R_n(x) = K(x)(x - x_0)(x - x_1) \cdots (x - x_n) = K(x) \prod_{i=0}^n (x - x_i)$$

其中, $K(x)$ 是与 x 有关的待定系数.

把 x 视作区间 $[a, b]$ 上的一个固定点, 作函数

$$\varphi(t) = f(t) - L_n(t) - K(t)(t - x_0)(t - x_1) \cdots (t - x_n)$$

因此, $\varphi(t)$ 在 x_0, x_1, \dots, x_n 和 x 处均为 0, 即在区间 $[a, b]$ 上有 $n+2$ 个零点. 根据 Roll 定理可知, $\varphi'(t)$ 在两个零点间至少有一个零点, 即在区间 $[a, b]$ 上, $\varphi'(t)$ 至少有 $n+1$ 个零点. 以此类推, 不难得出 $\varphi^{(n+1)}$ 在区间 (a, b) 内至少有一个零点, 将其记为 $\xi \in (a, b)$, 使得

$$\varphi^{(n+1)}(\xi) = f^{(n+1)}(\xi) - (n+1)!K(x) = 0$$

整理可得

$$K(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}, \xi \in (a, b)$$

将其代入上式, 可得余项表达式 (2.4)

□

通常, ξ 无法具体确定, 从而实际操作中, 经常估计误差上限. 由

$$|f^{(n+1)}(x)| \leq M_{n+1}$$

对于任意的 $x \in (a, b)$, 将

$$\frac{M_{n+1}}{(n+1)!} \prod_{i=0}^n |x - x_i|$$

作为误差估计上限. 通常取

$$M_{n+1} = \max_{a \leq x \leq b} |f^{(n+1)}(x)|$$

特别的, 若 $f(x)$ 为任一次数小于等于 n 的多项式, 即 $f(x) \in H_n \in \text{span}\{1, x, x^2, \dots, x^n\}$, 此时 $f^{(n+1)} \equiv 0$, 即 $R_n(x) \equiv 0$. 因此, 插值多项式对于次数小于等于 n 的多项式总是精确的.

例 2.2.2. 设有 $x_0 \neq x_1 \neq x_2 \neq x_3 \neq x_4$, 且 $l_i(x), i = 0, 1, 2, 3, 4$ 为 Lagrange 插值基函数. 求

$$\sum_{i=0}^4 (3x_i^4 + 4x_i^2 + 2x_i + 1) l_i(x)$$

解. 设函数 $f(x) = 3x^4 + 4x^2 + 2x + 1$, 则

$$\text{原式} = \sum_{i=0}^4 f(x_i)l_i(x) = f(x)$$

□

例 2.2.3. 已知 $\sin \frac{\pi}{6} = \frac{1}{2}$, $\sin \frac{\pi}{4} = \frac{\sqrt{2}}{2}$, $\sin \frac{\pi}{3} = \frac{\sqrt{3}}{2}$, 利用 $\sin x$ 的一次, 二次 Lagrange 插值, 计算 $\sin 50^\circ$ 并估计误差

解. 当 $n = 1$ 时, 利用 x_0, x_1 , 有

$$\begin{aligned}\Rightarrow L_1(x) &= \frac{x - \frac{\pi}{4}}{\frac{\pi}{6} - \frac{\pi}{4}} \cdot \frac{1}{2} + \frac{x - \frac{\pi}{6}}{\frac{\pi}{4} - \frac{\pi}{6}} \cdot \frac{\sqrt{2}}{2} \\ \Rightarrow L_1\left(\frac{5}{18}\pi\right) &\approx 0.77614\end{aligned}$$

其误差

$$R_1(x) = \frac{f''(\xi)}{2!} \left| x - \frac{\pi}{6} \right| \left| x - \frac{\pi}{4} \right|$$

其中 $|\sin \xi| \leq \frac{\sqrt{2}}{2}$, 因此有

$$\left| R_1\left(\frac{5}{18}\pi\right) \right| < 0.01319$$

类似地, 利用 x_1, x_2 , 可得 $L_1(x) \approx 0.76008$, 估计误差, 由于 $|\sin \xi| \leq \frac{\sqrt{3}}{2}$

$$\left| R_1\left(\frac{5}{18}\pi\right) \right| < 0.00660$$

当 $n = 2$ 时, 有

□

2.2.5 Lagrange 插值优缺点

优点: 具有严格的规律性, 便于记忆与理论推导;

缺点: 计算量大, 每次添加 (或删除) 节点都需要重新计算基函数, 不具有承袭性.

为解决上述缺点, 将引出新的插值法, 即 Newton 插值.

2.3 Newton 插值

2.3.1 Newton 插值

与 Lagrange 插值类似, 首先考虑 $n = 1$

例 2.3.1. 已知两个节点 x_0, x_1 , 其函数值分别为 y_0, y_1 , 试求一次多项式 $P_1(x) = A_0 + A_1x$, 使得 $P_1(x_0) = y_0, P_1(x_1) = y_1$

解. 利用直线方程点斜式, 可知

$$P_1(x) = y_0 + \frac{y_1 - y_0}{x_1 - x_0}(x - x_0)$$

此时, 插值节点为 x_0, x_1 , 基函数设为 $1, (x - x_0)$, 组合系数为 $A_0 = y_0, A_1 = \frac{y_1 - y_0}{x_1 - x_0}$ □

考虑 $n = 2$ 的情况, 要求具有承袭性. 设 $g(x) = P_2(x) - P_1(x)$, 则 $g(x)$ 为次数不超过 2 的多项式, 且有

$$g(x_i) = P_2(x_i) - P_1(x_i) = y_i - y_i = 0, i = 0, 1$$

因此可对其进行因式分解, 有

$$g(x) = A_2(x - x_0)(x - x_1)$$

$$\Rightarrow P_2(x) = P_1(x) + A_2(x - x_0)(x - x_1)$$

故, 对于 $n = 2$ 而言, 插值节点为 x_0, x_1, x_2 , 基函数为 $1, (x - x_0), (x - x_0)(x - x_1)$, 组合系数为 A_0, A_1, A_2 . 插值多项式为

$$P_2(x) = A_0 + A_1(x - x_0) + A_2(x - x_0)(x - x_1)$$

类似地, 给定插值条件

$$N_n(x_i) = f(x_i), i = 0, 1, \dots, n$$

, 插值节点为 $x_i, i = 0, 1, \dots, n$, 基函数为 $1, (x - x_0), (x - x_0)(x - x_1), \dots, (x - x_0)(x - x_1) \cdots (x - x_{n-1})$, 组合系数为 $A_i, i = 0, 1, \dots, n$

下面需要讨论如何求解组合系数.

设 $A_n(x) = A_0 + A_1(x - x_0) + A_2(x - x_0)(x - x_1) + \cdots + A_n(x - x_0)(x - x_1) \cdots (x - x_n)$, 利用插值条件

$$N_n(x_i) = f(x_i), i = 0, 1, \dots, n$$

代入, 得线性方程组

$$\begin{pmatrix} 1 & 0 & \cdots & 0 \\ 1 & x_1 - x_0 & \cdots & 0 \\ \vdots & \vdots & & \vdots \\ 1 & x_n - x_0 & \cdots & \prod_{i=0}^{n-1} (x_n - x_i) \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_n) \end{pmatrix}$$

求解方程组, 可得

$$\begin{aligned} A_0 &= f(x_0) \\ A_1 &= \frac{f(x_1) - f(x_0)}{x_1 - x_0} \\ A_2 &= \frac{\frac{f(x_2) - f(x_0)}{x_2 - x_0} - \frac{f(x_1) - f(x_0)}{x_1 - x_0}}{x_2 - x_1} \\ &\vdots \end{aligned}$$

为简化计算, 总结上述规律, 给出下面关于差商的定义

2.3.2 差商

差商的定义

定义 2.3.1 (差商). 称

$$f[x_0, x_k] = \frac{f(x_k) - f(x_0)}{x_k - x_0}$$

为函数 $f(x)$ 关于点 x_0, x_k 的一阶差商, 称

$$f[x_0, x_1, x_k] = \frac{f[x_0, x_k] - f[x_0, x_1]}{x_k - x_1}$$

为 $f(x)$ 关于点 x_0, x_1, x_k 的二阶差商. 一般地, 称

$$f[x_0, x_1, \dots, x_k] = \frac{f[x_0, x_1, \dots, x_{k-2}, x_k] - f[x_0, x_1, \dots, x_{k-1}]}{x_k - x_{k-1}}$$

为 $f(x)$ 的 k 阶差商.

由差商定义可知: 高阶差商是两个低一阶差商的差商.

利用差商定义, 可知组合系数为:

$$\begin{aligned} A_0 &= f(x_0) = f[x_0] \\ A_1 &= f[x_0, x_1] \\ &\vdots \\ A_n &= f[x_0, x_1, \dots, x_n] \end{aligned}$$

$$\Rightarrow N_n(x) = f(x_0) + f[x_0, x_1](x-x_0) + f[x_0, x_1, x_2](x-x_0)(x-x_1) + \dots + f[x_0, x_1, \dots, x_n](x-x_0)(x-x_1) \dots (x-x_{n-1})$$

差商性质

差商与函数值有如下关系:

定理 2.3.1. 记 $\omega(x) = (x-x_0)(x-x_1) \dots (x-x_n)$, 则

$$f[x_0, x_1, \dots, x_n] = \sum_{i=0}^n \frac{f(x_i)}{\omega'(x_i)}$$

证明. 对于 $f(x)$, 使用 Lagrange 插值法, 可得:

$$L_n(x) = \sum_{i=0}^n f(x_i) l_i(x)$$

使用 Newton 插值法, 可得:

$$N_n(x) = f(x_0) + f[x_0, x_1](x-x_0) + \dots + f[x_0, x_1, \dots, x_n](x-x_0)(x-x_1) \dots (x-x_{n-1})$$

由于插值多项式具有唯一性, 因此两种插值方法得到的结果相同, 即同次系数相等. 整理可得

$$\sum_{i=0}^n f(x_i) = f[x_0, x_1, \dots, x_n](x-x_0)(x-x_1) \dots (x-x_{n-1})$$

对 $\omega(x)$ 求导后, 原式得证. □

差商的值与节点的排列顺序无关, 即差商具有对称性. 用公式表示为:

$$f[x_0, \dots, x_i, \dots, x_j, \dots, x_n] = f[x_0, \dots, x_j, \dots, x_i, \dots, x_n]$$

设 $f(x)$ 在 $[a, b]$ 有 n 阶导数, 且 $x_0, x_1, \dots, x_n \in [a, b]$, 则存在 $\xi \in (a, b)$, 使得

$$f[x_0, x_1, \dots, x_n] = \frac{f^{(n)}(\xi)}{n!}$$

例 2.3.2. 若 $f(x) = 3x^4 + 4x^2 + 2x + 1$, 计算

$$f[2^0, 2^1, 2^2, 2^3, 2^4], f[2^0, 2^1, 2^2, 2^3, 2^4, 2^5]$$

解.

$$\begin{aligned} f[2^0, 2^1, 2^2, 2^3, 2^4] &= \frac{f^{(4)}(\xi)}{4!} = 3 \\ f[2^0, 2^1, 2^2, 2^3, 2^4, 2^5] &= \frac{f^{(5)}(\xi)}{5!} = 0 \end{aligned}$$

□

由前面的性质, 不难得到, 对于差商而言, 有

$$f[x_0, x_1, \dots, x_n] = \frac{f[x_1, x_2, \dots, x_n] - f[x_0, x_1, \dots, x_{n-1}]}{x_n - x_0}$$

该性质表明在实际计算差商过程中, 可以使用列表法计算.

2.3.3 Newton 插值余项

定理 2.3.2. *Newton* 插值多项式的余项为:

$$R_n(x) = f[x_0, x_1, \dots, x_n] \omega_{n+1}(x)$$

其中,

$$\omega_{n+1}(x) = (x - x_0)(x - x_1) \cdots (x - x_n) = \prod_{i=0}^n (x - x_i)$$