科学计算引论笔记

Chapter1 误差

一. 误差控制

- 避免大数除小数

  $\frac{x}{y}, y \to 0, x$的微小波动会引起很大误差

- 避免相近数相减

  相近数相减会损失有效数字

- 避免大数吃小数

  $Eg: A = 5331, B = 0.001, A + B$中 B就会被忽略(Solution:对数化)

- 减少运算次数

  多次迭代会导致误差被放大

二. 有效数字（重要）

定义：若x^*的近似值$x = \pm x_1 x_2 \dots x_n \times 10^m$，其中$x_1 \neq 0$，诸$x_i \in \{0, 1, 2, \dots, 9\}$，$m \in \mathbb{Z}$，且

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

则称近似值x有p位有效数字或称x精确到10^{m-p}位

$Example$:若以$\frac{355}{113}$作为圆周率π的逼近值，问此逼近值具有多少位有效数字

解：$x = \frac{325}{113} = 0.314159204 \dots \times 10^1$

$|x - \pi| = 2.6676 \dots \times 10^{-7} < 0.5 \times 10^{-6} = 0.5 \times 10^{1-7}$

∴有7位有效数字

Chapter2 迭代法

一. 二分法

1. 二分法的定义

用分点 $a_0 = a, x_0 = \frac{1}{2}(a + b), b_0 = b$ 将区间 $[a, b]$ 二等分

并计算函数值 $f(x_0)$. 若 $f(x_0) = 0$, 则求得实根 $x^* = x_0$; 否则$f(x_0)$与 $f(a)$ 或 $f(b)$ 异号.若 $f(a)f(x_0) < 0$, 则根在区间 $[a, x_0]$内,取 $a_1 = a, b_1 = x_0$; 若 $f(x_0)f(b) < 0$, 则根在区间 $[x_0, b]$ 内,取

$$a_1 = x_0, b_1 = b.$$

不断重复上述二分步骤,则可得系列隔离区间

$$[a_0, b_0] \supset [a_1, b_1] \supset [a_2, b_2] \supset \cdots \supset [a_k, b_k] \supset \cdots$$

其中二分k次后的隔离区间 $[a_k, b_k]$ 的长度为

$$b_k - a_k = \frac{1}{2^k}(b-a).$$

若记隔离区间 $[a_k, b_k]$ 的中点 $x_k = \frac{1}{2}(a_k + b_k)$, 则有:

$$|x^* - x_k| \leqslant \frac{1}{2}(b_k - a_k) = \frac{1}{2^{k+1}}(b-a), \quad k = 0, 1, 2, \cdots.$$

由此可知,随着二分次数 k 的增加,序列 $\{x_k\}$ 愈来愈逼近精确解 x^*. 因此,我们可取序列 $\{x_k\}$ 作为精确解x^*的逼近解序列若预定精度要求为 $\varepsilon > 0$: $|x^* - x_k| < \varepsilon$, ,我们只需

$$k > \frac{\ln(b-a) - \ln(2\varepsilon)}{\ln 2}.$$

以上即为二分法的过程

2. 二分法的优缺点

- 计算简单
- 对函数要求很低，仅需连续
- 收敛速度慢
- 未充分利用函数值信息

二. 简单迭代

1. Picard迭代法

定理2：设 Picard迭代格式（2.12)中的迭代函数 $\varphi(x)$ 满足下列条件：

$$(1)\forall x \in [a,b], a \leqslant \varphi(x) \leqslant b;$$
$$(2)\exists q \in (0,1), s.t. \forall x \in [a,b], |\varphi'(x)| \leqslant q < 1,$$

则该迭代格式自任意初值$x_0 \in [a,b]$ 出发均收敛于方程$x = \varphi(x)$的根 x^*,且有如下误差估计:

$$(a)|x^* - x_k| \leq \frac{1}{1-q}|x_{k+1} - x_k|$$
$$(b)|x^* - x_k| \leq \frac{q^k}{1-q}|x_1 - x_0|$$

证明：

事实上：我们有
$|x_{k+1} - x_k| = |\varphi'(\zeta)||x_k - x_{k-1}| \leq q|x_k - x_{k-1}| \leq q^2|x_{k-1} - x_{k-2}| \leq \ldots \leq q^k|x_1 - x_0|$

$$\begin{aligned}
|x_{n+p} - x_k| &= |x_{n+p} - x_{n+p-1} + x_{n+p-1} - x_{n+p-2} + \ldots + x_{k+1} - x_k| \\
&\leq |x_{n+p} - x_{n+p-1}| + |x_{n+p-1} - x_{n+p-2}| + \ldots + |x_{k+1} - x_k| \\
&\leq q^{p-1}|x_{k+1} - x_k| + q^{p-2}|x_{k+1} - x_k| + \ldots + |x_{k+1} - x_k| \\
&= (q^{p-1} + q^{p-2} + \ldots + 1)|x_{k+1} - x_k| \\
&= \frac{1 - q^{p-1}}{1-q}|x_{k+1} - x_k|
\end{aligned}$$

当$p \to \infty$时， $|x^* - x_k| \leq \frac{1}{1-q}|x_{k+1} - x_k|$

由上式易得：$|x^* - x_k| \leq \frac{q^k}{1-q}|x_1 - x_0|$

局部收敛定理：

$$(a)\exists\delta>0,\varphi(x)\in CN(x^*,\delta)$$
$$(b)|\varphi'(x)|\leq 1$$

满足以上条件也可保证$x=\varphi(x)$局部收敛

关于改造迭代函数的方法1：

给定方程$f(x)=0$求其根，最容易想到的方法就是在方程两边同时加上x，变为$x=f(x)+x=\varphi(x)$，但是迭代函数并不一定能满足收敛条件，下面介绍一种改造迭代函数的方法：

- 首先确定$f'(x)\in[m,M],0\leq m\leq M$
- 修改方程为：$x=x-\lambda f(x)\Rightarrow\varphi(x)=x-\lambda f(x)$
- 确定λ的取值

$$\begin{aligned}&|\varphi'(x)|=|1-\lambda f'(x)|<1\\\Leftrightarrow\quad&-1<1-\lambda f'(x)<1\\\Leftrightarrow\quad&0<\lambda f'(x)<2\\\Leftrightarrow\quad&0<\lambda<\frac{2}{M}\end{aligned}$$

关于改造迭代函数的方法2：

给定迭代方程$x=\varphi(x)$，改造迭代函数$\phi(x)=\varphi(x)+\lambda[\varphi(x)-x]s.t.\phi(x^*)\approx 0$，此时近似有：

$$\lambda=\frac{\varphi'(x^*)}{1-\varphi'(x^*)}\approx\frac{\varphi'(x_k)}{1-\varphi'(x_k)}$$

这种方法实际上调节了每次迭代的步长（类似与神经网络中的学习率控制参数更新速度），Python代码实现：

```
def h(x):
    return x**2 - 2

def Picard(x, epsilon):
    x_new = 0
    count = 0
    while(True):
        x_new = h(x) + x
        if np.abs(x_new - x) < epsilon or count > 100:
            break
        print(f"Epoch {count+1}: {x_new, x}")
        x = x_new
        count += 1
    return x_new

def Picard_new(x, omega, epsilon):# 改进的Picard算法
    x_new = 0
    count = 0
    while(True):
        x_new = x - omega * h(x)
        if np.abs(x_new - x) < epsilon or count > 100:
            break
        print(f"Epoch {count+1}: {x_new, x}")
        x = x_new
        count += 1
    return x_new
```

```
res = Picard_new(0.24, 0.5, 10e-8)
res
```

```
Epoch 1: (1.2111999999999998, 0.24)
Epoch 2: (1.4776972800000001, 1.2111999999999998)
Epoch 3: (1.3859026543403008, 1.4776972800000001)
Epoch 4: (1.425539570686555, 1.3859026543403008)
Epoch 5: (1.4094580368899512, 1.425539570686555)
Epoch 6: (1.4161720580131136, 1.4094580368899512)
Epoch 7: (1.4134004090645649, 1.4161720580131136)
Epoch 8: (1.4145500508926252, 1.4134004090645649)
Epoch 9: (1.4140741276524609, 1.4145500508926252)
Epoch 10: (1.4142713084044267, 1.4140741276524609)
Epoch 11: (1.414189641516442, 1.4142713084044267)
Epoch 12: (1.4142234704302405, 1.414189641516442)
Epoch 13: (1.4142094582723639, 1.4142234704302405)
Epoch 14: (1.4142152623388573, 1.4142094582723639)
Epoch 15: (1.4142128582227758, 1.4142152623388573)
Epoch 16: (1.4142138540414595, 1.4142128582227758)
Epoch 17: (1.4142134415600602, 1.4142138540414595)
Epoch 18: (1.4142136124154852, 1.4142134415600602)

1.4142135416448571
```

2. Aitken迭代法

关于Picard迭代法的简单改进:

Picard迭代法计算格式简单，但其收敛速度一般较慢，为提高其收敛速度，本节考虑改进 Picard迭代法. 设 x_k 为第k次迭代逼近值，迭代函数 $\varphi(x)$ 在方程$x=\varphi(x)$的精确解 x^* 的某邻域内连续可微，且其导数值变化不大,记其近似值为l，则由Taylor展开式有

$$\begin{aligned} x^*-\varphi(x_k) &= \varphi(x^*)-\varphi(x_k) \\ &= \int_0^1 \varphi'(x_k+\theta(x^*-x_k))(x^*-x_k)\mathrm{d}\theta \\ &\approx l(x^*-x_k). \end{aligned}$$

由此得

$$x^* \approx (1-l)^{-1}[\varphi(x_k)-lx_k].$$

故得加速迭代格式

$$x_{k+1}=(1-l)^{-1}[\varphi(x_k)-lx_k], \quad k=0,1,\cdots.$$

Aitken迭代法:

上述迭代格式可加快 Picard迭代法的收敛速度，但该方法的缺陷是需要确定参数 l， 而这对于某些方程而言是十分困难的.为克服该困难,我们引入如下 Aitken (艾特肯）加速迭代法.记

$$\overline{x}_{k+1}=\varphi(x_k), \quad \overline{\overline{x}}_{k+1}=\varphi(\overline{x}_{k+1}),$$

则由 Taylor展开定理近似地有

$$x^*-\bar{x}_{k+1}\approx l(x^*-x_k), \quad x^*-\overline{\overline{x}}_{k+1}\approx l(x^*-\overline{x}_{k+1}).$$

由上两式消去未知参数l得

$$x^* \approx \overline{\overline{x}}_{k+1} - \frac{(\overline{\overline{x}}_{k+1} - \overline{x}_{k+1})^2}{\overline{\overline{x}}_{k+1} - 2\overline{x}_{k+1} + x_k}.$$

故得Aitken 加速迭代格式

$$x_{k+1} = \overline{\overline{x}}_{k+1} - \frac{(\overline{\overline{x}}_{k+1} - \overline{x}_{k+1})^2}{\overline{\overline{x}}_{k+1} - 2\overline{x}_{k+1} + x_k}.$$

Aitken迭代无需计算导数值，这是其最大的优势之一，下面给出Aitken迭代法的Python实现：

```
def g(x):
    return x**3 - 1

def Aitken(x, epsilon):
    b = 0.0
    a = 0.0
    i = 0
    while(True):
        a = g(x) + x
        b = g(a) + a
        x_new = b - (b - a)**2/(b - 2*a + x)
        if np.abs(x_new - x) < epsilon or i > 100:
            break
        print(f'Epoch {i+1}: {a, b, x}')
        x = x_new
        i += 1

res = Aitken(1.5, 10e-8)
res
```

输出如下：

```
Epoch 1: (3.875, 61.060546875, 1.5)
Epoch 2: (3.12400578317771, 32.612465728108404, 1.3970886932972206)
Epoch 3: (2.4346830558289363, 15.866708964702676, 1.2896651739743845)
Epoch 4: (1.8383965988421707, 7.0516293611820595, 1.1829617399989463)
Epoch 5: (1.3791320274008025, 3.0022482446715903, 1.0887068249538423)
Epoch 6: (1.1036193207997305, 1.447800731008053, 1.0254162367543656)
Epoch 7: (1.0097093292285497, 1.0391210454452988, 1.0024229258239874)
Epoch 8: (1.0000933457969126, 1.000373409328777, 1.0000233360407969)
1.0
```

三. Newton迭代法（重要）

Newton 迭代法是一种求解非线性方程的高效方法，其通过在隔离区间 $[a,b]$ 上不断作曲线 $y=f(x)$ 的切线而获得解的逼近序列.构造该方法的具体步骤如下：在方程$f(x)=0$的解的隔离区间 $[a,b]$ 上选取适当迭代初值x_0，过曲线 $y=f(x)$ 的点 $(x_0,f(x_0))$引切线

$$l_1: \quad y=f(x_0)+f'(x_0)(x-x_0),$$

其与x轴相交于点

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

进一步,过曲线 $y = f(x)$ 的点 $(x_1, f(x_1))$ 引切线

$$l_2: \quad y = f(x_1) + f'(x_1)(x - x_1)$$

其与x轴相交于点

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$$

如此循环往复，可得一列逼近方程$y = f(x)$ 精确解x^*的点
$x_0, x_1, \ldots, x_k, \ldots$，其一般表达式为

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})}, \quad k = 1, 2, \cdots.$$

该公式所表述的求解方法称为 **Newton 迭代法**或**切线法**

$Example$：应用 Newton 迭代法求方程 $x^3 - x - 1 = 0$ 在 $x = 1$附近的数值解x_k，并使其满足 $|x_k - x_{k-1}| < 10^{-8}$.

代码实现如下

```
def f_(x):
    return 3*(x**2) - 1

def f(x):
    return x**3 -1 - x


a = 1.3
while(True):
    temp = a
    print(a)
    a = a - f(a)/f_(a)
    if math.fabs(temp - a) < 10e-8:
        break
```

输出:

```
1.3
1.3253071253071254
1.324718280461173
1.3247179572448433
```

Newton法收敛的充分条件:

设 $f \in \mathbb{C}^2[a, b]$, 若满足以下条件:

$$(1) f(a)f(b) < 0$$
$$(2) \forall x \in [a, b], f''(x) > 0 (< 0) f'(x) \neq 0;$$
$$(3) Choose \quad x_0 \in [a, b] s.t. f(x_0)f''(x_0) > 0;$$

则$\{x_k\}$收敛到$f(x)$在$[a, b]$的唯一根。

牛顿法主要有**两个缺点**：**局部收敛，计算量大**

牛顿法的简化版本：

- 简易Newton法

$$x_{k+1}=x_k-\frac{f(x_k)}{M}\qquad(k=0,1,2,\dots)$$

- 割线法

$$x_{k+1}=x_k-\frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})}f(x_k)\quad(k=0,1,2,\cdots)$$

- 牛顿下山法

$$x_{k+1}=x_k-\boldsymbol{\omega}\frac{f(x_k)}{f'(x_k)}\quad(k=0,1,2,\cdots)$$

  可引入一个下山因子$\omega(0<\omega\leq1)$使每一步有$|f(x_{k+1})|<|f(x_k)|$（同样为步长控制，ω为学习率)

牛顿法如何解：$\begin{cases}f_1(x_1,x_2)=0\\f_2(x_1,x_2)=0\end{cases}$？

引入广义$F(\boldsymbol{x})=0=F(\boldsymbol{x}_k)+\frac{\partial F}{\partial\boldsymbol{x}_k}(\boldsymbol{x}-\boldsymbol{x}_k)$

$J(\boldsymbol{x}_k)\boldsymbol{x}_{k+1}=J(\boldsymbol{x}_k)\boldsymbol{x}_k-F(\boldsymbol{x}_k)$

Chapter3 线性方程组的数值解法

一. 直接法

1. Cramer法则

计算量太大，舍去

2. Gauss消元法

Gauss法的主要思想是首先将方程组$A\boldsymbol{X}=\boldsymbol{b}$，化为一个系数矩阵为下三角形矩阵（或上三角形矩阵）的方程组，然后采用前推（或回代)方法求得其线性方程组的解。 为利用计算机实现 Gauss顺序消元，我们将计算过程中出现的矩阵及其元素进行编号，其计算步骤如下

记方程组（3.1)为 $A^{(1)}\boldsymbol{X}=\boldsymbol{b}^{(1)}$, 其中

$$A^{(1)}=\begin{pmatrix}a_{11}^{(1)}&a_{12}^{(1)}&\cdots&a_{1n}^{(1)}\\a_{21}^{(1)}&a_{22}^{(1)}&\cdots&a_{2n}^{(1)}\\\vdots&\vdots&&\vdots\\a_{n1}^{(1)}&a_{n2}^{(1)}&\cdots&a_{nn}^{(1)}\end{pmatrix}=A,\quad b^{(1)}=\begin{pmatrix}b_1^{(1)}\\b_2^{(1)}\\\vdots\\b_n^{(1)}\end{pmatrix}=b.$$

假设 $a_{11}^{(1)}\neq0$, 取 $m_{i1}=a_{i1}^{(1)}/a_{11}^{(1)}(i=2,3,\cdots,n)$, 用该数的负值乘方程组$A\boldsymbol{X}=\boldsymbol{b}$的第1个方程, 然后将其加到第1个方程$(i=2,3,\cdots,n)$上,则依次可消去自第2个方程到第n个方程中的变量 x_1, 由此得以下等价方程组

$$A^{(2)}X=b^{(2)}$$

其中：

$$A^{(2)}=\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & \cdots & a_{nn}^{(2)} \end{pmatrix}, \quad b^{(2)}=\begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{pmatrix},$$

$$a_{ij}^{(2)}=a_{ij}^{(1)}-m_{i1}a_{1j}^{(1)}, \quad b_i^{(2)}=b_i^{(1)}-m_{i1}b_1^{(1)}, \quad i,j=2,3,\cdots,n.$$

假设 $a_{22}^{(2)}\neq 0$, 取 $m_{i2}=a_{i2}^{(2)}/a_{22}^{(2)}(i=3,4,\cdots,n)$，再用 m_{i2}的负值乘方程$A^{(2)}X=b^{(2)}$的第2个方程,然后将其加到第1个方程$(i=3,4,\cdots,n)$上，则依次可消去自第3个方程到第n个方程中的变量 x_2，得以下等价方程组

$$A^{(3)}X=b^{(3)},$$

其中:

$$A^{(3)}=\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \cdots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \cdots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(3)} & \cdots & a_{3n}^{(3)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{n3}^{(3)} & \cdots & a_{nn}^{(3)} \end{pmatrix}, \quad b^{(3)}=\begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ b_3^{(3)} \\ \vdots \\ b_n^{(3)} \end{pmatrix},$$

$$a_{ij}^{(3)}=a_{ij}^{(2)}-m_{i2}a_{2j}^{(2)}, \quad b_i^{(3)}=b_i^{(2)}-m_{i2}b_2^{(2)}, \quad i,j=3,4,\cdots,n.$$

重复上述步骤，经 $n-1$ 次消元后得以下系数矩阵为上三角形矩阵的方程组

$$A^{(n)}X=b^{(n)},$$

其中:

$$A^{(n)}=\begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \cdots & a_{1n}^{(1)} \\ & a_{22}^{(2)} & \cdots & a_{2n}^{(2)} \\ & & \ddots & \vdots \\ & & & a_{nn}^{(n)} \end{pmatrix}, \quad \boldsymbol{b}^{(n)}=\begin{pmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(n)} \end{pmatrix},$$

$$a_{nn}^{(n)}=a_{nn}^{(n-1)}-m_{n,n-1}a_{n-1,n}^{(n-1)}, b_n^{(n)}=b_n^{(n-1)}-m_{n,n-1}b_{n-1}^{(n-1)},$$

$$m_{n,n-1}=\frac{a_{n,n-1}^{(n-1)}}{a_{n-1,n-1}^{(n-1)}}.$$

上述过程称为前推过程

在前推过程完成后,我们从方程组 $A^{(n)}X=b^{(n)}$ 的第n个方程开始，自下而上依次解出 $x_n,x_{n-1},\cdots,x_1$， 该过程称为回代过程,其计算公式如下:

$$x_n=\frac{b_n^{(n)}}{a_{nn}}, \quad x_i=\frac{b_i^{(i)}-\sum_{j=i+1}^{n}a_{ij}^{(i)}x_j}{a_{ii}^{(i)}}, \quad i=n-1,n-2,\cdots,1]$$

上述方法即为 Gauss 顺序消元法。这种消元法计算可行性的前提条件是其消元过程的所有主元素 $a_{kk}^{(k)}\neq 0$，否则将导致计算过程无法进行或计算结果严重失真。

时间复杂度：$O(n^3)$

3. Gauss主元素消去法

4. 矩阵的LU分解

将矩阵分解为一个上三角矩阵和下三角矩阵的乘积

5. 对称正定矩阵的Cholesky分解

$LU_1 = \boldsymbol{A} = \boldsymbol{A}^T = U_1^T L^T$

$U_1^T = U^T D \Rightarrow U_1 = DU$

$$LDU = \boldsymbol{A} = \boldsymbol{A}^T = (DU)^T L^T = U^T D L^T$$
$$\Rightarrow L = U^T$$
$$\Rightarrow A = LDL^T = \tilde{L}\tilde{L}^T, D > 0$$

Cholesky分解的求法:

- 计算 L 的第 1 列元素

$$l_{11} = \sqrt{a_{11}}, \quad l_{i1} = a_{i1}/l_{11}, \quad i = 2, 3, \cdots, n.$$

- 若 L 的前 $k-1$ 列元素已计算, 则进一步计算 L的第 k 列元素

$$l_{kk} = \sqrt{a_{kk} - \sum_{p=1}^{k-1} l_{kp}^2}, \quad l_{ik} = \left(a_{ik} - \sum_{p=1}^{k-1} l_{ip} l_{kp}\right)/l_{kk},$$
$$i = k+1, k+2, \cdots, n.$$

- 在完成 Cholesky 分解后, 我们可分别求解以下系数矩阵为下三角形矩阵和上三角形矩阵的方程组

$$LY = b, \quad L^{\mathrm{T}} X = Y$$

  从而获得原方程组 的解 X

我们注意到上述线性方程组的解法含有开方运算,其在计算过程中占用大量的运行时间.为避免开方运算, 在下面我们介绍一个改进的 Cholesky 分解法,或称为改进的平方根法. 由定理 3.3 的证明过程可知, 对称正定矩阵 A 也有如下分解:

$$A = LDL^{\mathrm{T}},$$

其中 $L = (l_{ij})$ 为单位下三角形矩阵, $D = \mathrm{diag}(d_1, d_2, \cdots, d_n) > 0$.由(3.17)有

$$a_{ij} = \sum_{k=1}^{j-1} l_{ik} d_k l_{jk} + l_{ij} d_j, \quad 1 \leqslant j \leqslant i \leqslant n.$$

由该式可得 L, D 的如下计算步骤:

- Step 1. 计算 d_1, L 的第 1 列元素

$$d_1 = a_{11}, \quad l_{j1} = a_{j1}/d_1, \quad j = 2, 3, \cdots, n.$$

- Step 2.若 D, L 的前 $j-1$ 列元素已计算,则计算 D,L 的第 j 列元素

$$d_j = a_{jj} - \sum_{k=1}^{j-1} l_{jk} v_{jk}, \quad v_{jk} = l_{jk} d_k,$$

$$l_{ij} = \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} v_{jk}\right)/d_j, \quad i = j+1, j+2, \cdots, n.$$

- 在完成分解后,我们可分别求解下列系数矩阵为下三角形矩阵和上三角形矩阵的方程组,从而获得原方程组的解 X

$$LY = b, \quad DL^{\mathrm{T}}X = Y.$$

6. 三对角矩阵的追赶法

形如以下形式的三对角矩阵如下:

$$\begin{pmatrix} b_1 & c_1 & & & & & \\ a_2 & b_2 & c_2 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & a_i & b_i & c_i & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_i \\ \vdots \\ \vdots \\ x_{n-1} \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_i \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}.$$

LU分解:

$$\begin{bmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & b_n & a_n \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ p_2 & 1 & & & \\ & p_2 & 1 & & \\ & & \ddots & \ddots & \\ & & & p_n & 1 \end{bmatrix} \begin{bmatrix} q_1 & c_1 & & & \\ & q_2 & c_2 & & \\ & & \ddots & \ddots & \\ & & & q_{n-1} & c_{n-1} \\ & & & & q_n \end{bmatrix}$$

$$= \begin{bmatrix} q_1 & c_1 & & & & \\ p_2q_1 & p_2c_1+q_2 & c_2 & & & \\ & p_3q_2 & p_3c_2+q_3 & c_3 & & \\ & & & & c_{n-1} & \\ & & & & & \\ & & & P_nq_{n-1} & P_nc_{n-1}+q_n \end{bmatrix} \begin{cases} q_1 = a_1 \\ p_i = \frac{b_i}{q_{i-1}} \\ p_i = a_i - p_ic_{i-1} \end{cases}$$

追赶法:

求$Ax = f$等价于求$\begin{cases} Ly = f \\ Ux = y \end{cases}$其中$f = (b_1, f_2, \ldots, f_n)^T$,故有:

$$\begin{bmatrix} 1 & & & & \\ p_2 & 1 & & & \\ & p_3 & 1 & & \\ & & \ddots & \ddots & \\ & & & p_n & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_n \end{bmatrix}$$

解得$\begin{cases} y_1 = f_1 \\ y_i = f_i - p_iy_{i-1} \end{cases} (i = 2, \ldots, n)$

再由

$$\begin{bmatrix} q_1 & c_1 & & & \\ & q_2 & c_2 & & \\ & & \ddots & \ddots & \\ & & & q_{n-1} & c_{n-1} \\ & & & & q_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}$$

解得 $\begin{cases} x_n = \dfrac{y_n}{q_n} \\ x_i = \dfrac{y_i - c_i x_{i+1}}{q_i} \quad (i = n-1, \ldots, 1) \end{cases}$

以上称为解三对 角方程组的追赶法

二. 迭代法

1. 几个主要问题

- 迭代格式

  $X^{(k+1)} = BX^{(k)} + d$
- 迭代矩阵B满足的条件

2. Jacobi迭代法

Jacobi 迭代公式

$$X^{(k+1)} = -D^{-1}(L+U)X^{(k)} + D^{-1}b, \quad k = 0, 1, \cdots,$$

其中$X^{(k)} = (x_1^{(k)}, x_2^{(k)}, \cdots, x_n^{(k)})^{\mathrm{T}} \in \mathbb{R}^n$.该迭代公式也可写成如下分量形式:

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

中止条件$||X^{k+1} - X^k||_2 < \varepsilon$

只保留对角线元素

P范数：$||\boldsymbol{x}||_p = (|x_1|^p + |x_2|^p + \ldots + |x_n|^p)^{\frac{1}{p}}$

3. Gauss-Seidel迭代法

Gauss-Seidel迭代公式

$$\begin{cases} x_1^{(k+1)} = \frac{1}{a_1} \left[b_1 - (a_{12} x_2^{(k)} + a_{13} x_3^{(k)} + a_{14} x_4^{(k)} + \ldots + a_{1n} x_n^{(k)}) \right] \\ x_2^{(k+1)} = \frac{1}{a_{22}} \left[b_2 - (a_{21} x_1^{(k+1)} + a_{23} x_3^{(k)} + a_{21} x_4^{(k)} + \ldots + a_{2n} x_n^{(k)}) \right] \\ x_3^{(k+1)} = \frac{1}{a_{33}} \left[b_3 - (a_{31} x_1^{(k+1)} + a_{32} x_2^{(k+1)} + a_{34} x_4^{(k)} + \ldots + a_{3n} x_n^{(k)}) \right] \\ \cdots \\ x_n^{(k+1)} = \frac{1}{a_m} \left[b_n - (a_{n1} x_1^{(k+1)} + a_{n2} x_2^{(k+1)} + a_{n3} x_3^{(k+1)} + \ldots + a_{n(n-1)} x_{n-1}^{(k+1)}) \right] \end{cases}$$

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

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

$$\boldsymbol{x}^{(k+1)} = -(L+D)^{-1} U \boldsymbol{x}^{(k)} + (L+D)^{-1}$$

Gauss-Seidel迭代法一般比Jacobi迭代法收敛更快

但是在大规模计算上不见得，因为Jacobi迭代法并行度高，但Gauss-Seidel基本上串行计算，并行度差

4. 加速法

$$Dx^{(k+1)} = (1-\omega)Dx^{(k)} + \omega\left(Ix^{(k+1)} + Ux^{(k)} + b\right)$$
$$(D-\omega L)x^{(k+1)} = ((1-\omega)D + \omega U)x^{(k)} + \omega b$$
$$B_\omega = (D-\omega L)^{-1}((1-\omega)D + \omega U)$$
$$f_\omega = \omega(D-\omega L)^{-1}b$$
$$x^{(k+1)} = B_\omega x^{(k)} + f_\omega$$

上式为逐次超松弛法(SOR迭代法)的矩阵形式

当$\omega = 1$时,SOR法化为

$$x^{(k+1)} = (D-L)^{-1}Ux^{(k)} + (D-L)^{-1}b \quad \textbf{G-S}\text{送代法}$$

G-S法为SOR法的特例，SOR法为G-S法的加速

对角占优的方程组更容易收敛

Chapter4 插值方法

在实际计算中常遇到这样的情况: 函数的解析表达式$f(x)$未知,而仅仅知道它在若干个不同点处的函数值;或者函数的解析表达式非常复杂,仅仅给出若干个点处的函数值。设

$$f(x_i) = y_i\ (i = 0, 1, 2, \cdots, n)$$

对任意点处$x \neq x_i$,如何计算$f(x)$的近似值? 从一个简单函数类中求$p(x)$,使得

$p(x_i) = y_i, \quad i = 0, 1, \cdots, n$
而在其他$x \neq x_i$的点,$p(x) \approx f(x)$。这类问题称为插值问题。$x_0, x_1, \cdots, x_n$称为插值节点,所在区间$[a, b]$称为插值区间, $p(x)$称为插值函数。

1. 直接插值法

插值函数为多项式函数：

$$P_n(x) = \sum_{k=0}^{n} a_k x^k$$

待定系数：$a_1, a_2, \ldots, a_n$

$$\begin{cases} a_0 + a_1x_0 + a_2x_0^2 + \cdots + a_nx_0^n = y_0 \\ a_0 + a_1x_1 + a_2x_1^2 + \cdots + a_nx_1^n = y_1 \\ \vdots \\ a_0 + a_1x_n + a_2x_n^2 + \cdots + a_nx_n^n = y_n \end{cases}$$

系数矩阵：范德蒙(Vandermonde)行列式

$$| D |=\begin{vmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^n \\ 1 & x_1 & x_1^2 & \cdots & x_1^n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^n \end{vmatrix}=\prod_{0\le i<j\le n}(x_j-x_i)$$

解出待定系数即可

2. Lagrange插值法

Lagrange多项式：

$$l_j(x)=\prod_{i=0,i\neq j}^{n}\frac{x-x_i}{x_j-x_i}$$

满足：

$$l_j(x_i)=\delta_{ij}\triangleq\begin{cases}1, & i=j,\\ 0, & i\neq j,\end{cases}\quad i,j=0,1,2,\cdots,n.$$

Lagrange插值多项式：

$$L_n(x)=\sum_{j=0}^{n}y_jl_j(x),n>0,$$

插值误差：

设$f(x)\in C^n[a,b],f^{(n+1)}$在开区间$(a,b)$存在，则Lagrange插值多项式的余项有以下估计：

$$R_n(x)=f(x)-L_n(x)=\frac{f^{(n+1)}(\xi)}{(n+1)!}\omega(x),\quad \xi\in(a,b)s.t.x$$

插值点内部误差小，外部误差大

3. Newton插值法

差商的定义 设 $x_0,x_1,\cdots,x_n$ 为区间 $[a,b]$ 上的互异节点, 则称 $f(x_i)$ 为 $f(x)$ 在 x_i 处的零阶差商; 称

$$\frac{f(x_i)-f(x_j)}{x_i-x_j}$$

为函数 $f(x)$ 在 x_i,x_j 处的一阶差商, 记为 $f[x_i,x_j]$; 一般称

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

为 $f(x)$ 在 $x_0,x_1,\cdots,x_n$ 处的 n 阶差商. 利用差商的定义及数学归纳法可以直接获得n 阶差商的表达式.

定理 4.5 n 阶差商

$$f[x_0,x_1,\cdots,x_n]=\sum_{j=0}^{n}\frac{f(x_j)}{\omega'_{n+1}(x_j)},\omega_{n+1}(x)=\prod_{i=0}^{n}(x-x_i)$$

上式表明差商的值与节点$\{x_i\}_{i=0}^n$ 的排序无关,即具有对称性.此外,由差商定义有

$$
\begin{aligned}
&f(x) = f(x_0) + f[x_0, x](x - x_0),\\
&f[x_0, x] = f[x_0, x_1] + f[x_0, x_1, x](x - x_1),\\
&f[x_0, x_1, x] = f[x_0, x_1, x_2] + f[x_0, x_1, x_2, x](x - x_2),\\
&\cdots\cdots\cdots\\
&f[x_0, x_1, \cdots, x_{n-1}, x] = f[x_0, x_1, \cdots, x_n] + f[x_0, x_1, \cdots, x_n, x](x - x_n).
\end{aligned}
$$

从上面最后一个式子逐次往上代人,最终得

$$
f(x) = N_n(x) + R_n(x),
$$

其中

$$
\begin{aligned}
&N_n(x) = f(x_0) + f[x_0, x_1](x - x_0) + \cdots + f[x_0, x_1, \cdots, x_n]\omega_n(x)\\
&R_n(x) = f[x_0, x_1, \cdots, x_n, x]\omega_{n+1}(x),\\
&\omega_n(x) = \prod_{i=0}^{n-1}(x - x_i),
\end{aligned}
$$

由于

$$
f(x_i) - N_n(x_i) = R_n(x_i) = 0, \quad i = 0, 1, 2, \cdots, n,
$$

且 $N_n(x)$ 为 n 次多项式, 则由插值解唯一可知

$$
N_n(x) \equiv P_n(x), \quad \forall x \in [a, b].
$$

因此, $N_n(x)$ 可作为 $f(x)$ 的 n 次插值多项式, 称之为 n 次 **Newton** 插值公式., 其截断误差为

$$
R_n(x) = f(x) - N_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\omega_{n+1}(x), \quad x \in [a, b],
$$

其中ξ在诸x_i 与x之间.

核心：计算n阶差商（使用差商表）

1. 差商表-1

x_k	$f(x_k)$	$f[x_{k-1}, x_k]$	$f[x_{k-2}, x_{k-1}, x_k]$	...	$f[x_1, x_2, .., x_n]$
x_0	$f(x_0)$				
x_1	$f(x_1)$	$f[x_0, x_1]$			
x_2	$f(x_2)$	$f[x_1, x_2]$	$f[x_0, x_1, x_2]$		
x_3	$f(x_3)$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$	...	
...	...	...	...	...	
x_n	$f(x_n)$	$f[x_{n-1}, x_n]$	$f[x_{n-2}, x_{n-1}, x_n]$	...	$f[x_1, x_2, .., x_n]$

2. 差商表-2

x_k	$f(x_k)$	$f[x_0, x_k]$	$f[x_0, x_1, x_k]$	...	$f[x_1, x_2, .., x_n]$
x_0	$f(x_0)$				
x_1	$f(x_1)$	$f[x_0, x_1]$			
x_2	$f(x_2)$	$f[x_0, x_2]$	$f[x_0, x_1, x_2]$		
x_3	$f(x_3)$	$f[x_0, x_3]$	$f[x_0, x_1, x_3]$	...	
...	...	...	...	...	
x_n	$f(x_n)$	$f[x_0, x_n]$	$f[x_0, x_1, x_n]$	...	$f[x_1, x_2, .., x_n]$

Example：

等距节点Newton插值公式：

实际应用中，常是等距节点：$x_i = a + ih$，这时候Newton插值公式还能得到简化，我们首先定义差商的概念：

- 称$\Delta f_i = f(x_{i+1}) - f(x_i)(i = 0, 1, 2, \cdots, n)$为函数 $f(x)$在点$\{x_i\}_0^n$上的一阶向前差分(简称差分);又称$\Delta^k f_i = \Delta^{k-1} f(x_{i+1}) - \Delta^{k-1} f(x_i)(k=1, 2, \ldots, n; i = 0, 1, \ldots, n-k)$为函数$f(x)$在点$\{x_i\}_0^n$上的 k 阶向前差分,这里约定$\{x_i\}_0^n$
- 称$\nabla f_i = f(x_i) - f(x_{i-1}) \quad (i = n, n-1, \ldots, 1)$为函数 $f(x)$在点上的$\{x_i\}_0^n$后差分; 又称$\nabla^k f_i = \nabla^{k-1} f(x_i) - \nabla^{k-1} f(x_{i-1}) \quad (k = 1, 2, \ldots, n; \quad i = n-k+1, \ldots, 2, 1)$为函数$f(x)$在点$\{x_i\}_0^n$上的$k$阶向后差分, 同样约定$\nabla^0 f_i = f_i$

差商和差分的关系：在等距节点条件下，

$$f[x_0, x_1] = \frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{1}{h}\Delta f_0$$

$$f[x_0, x_1, x_2] = \frac{f[x_1, x_2]) - f[x_0, x_1])}{x_2 - x_0} = \frac{\frac{1}{h}\Delta f_1 - \frac{1}{h}\Delta f_0}{2h} = \frac{1}{2!h^2}\Delta^2 f_0$$

$$Common \quad conculsion: f[x_0, x_1, \ldots, x_n] = \frac{1}{n!h^n}\Delta^n f_0$$

现在给出等距Newton插值公式：令$x = x_0 + th$，则Newton插值公式和余项可以表示为(n阶向前差分)：

$$N_n(x) = N_n(x_0 + th) = f_0 + \frac{t}{1}\Delta f_0 + \frac{t(t-1)}{2!}\Delta^2 f_0 + \ldots + \frac{t(t-1)\ldots(t-n+1)}{n!}\Delta^n f_0$$

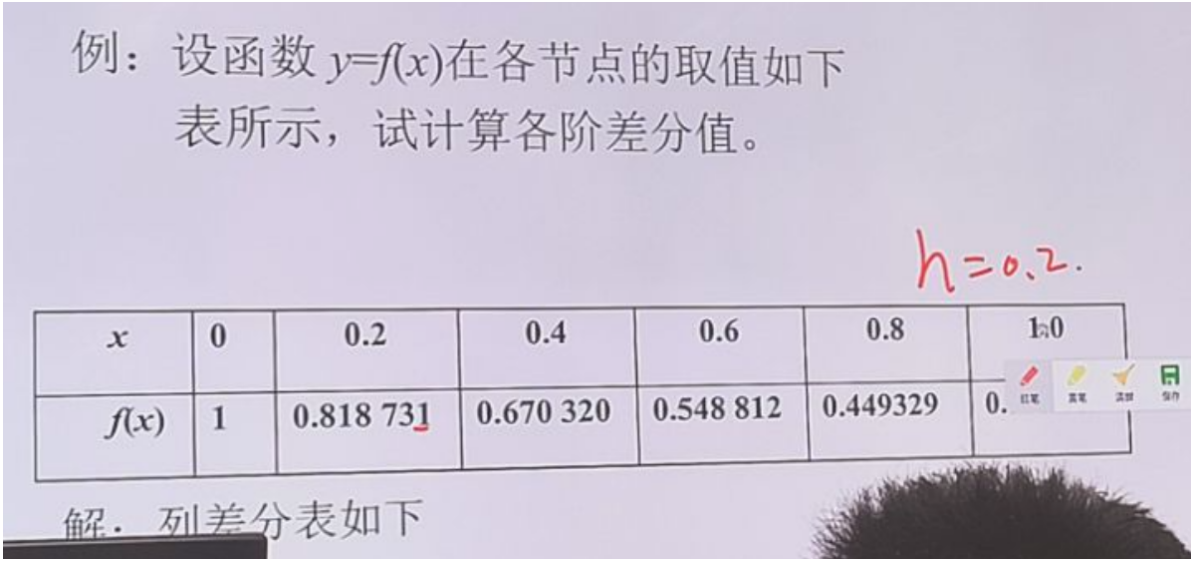
$$R_n(x) = \frac{h^{(n+1)}}{(n+1)!}t(t-1)\ldots(t-n)f^{(n+1)}(\xi), \xi \in (x_0, x_0 + nh)$$

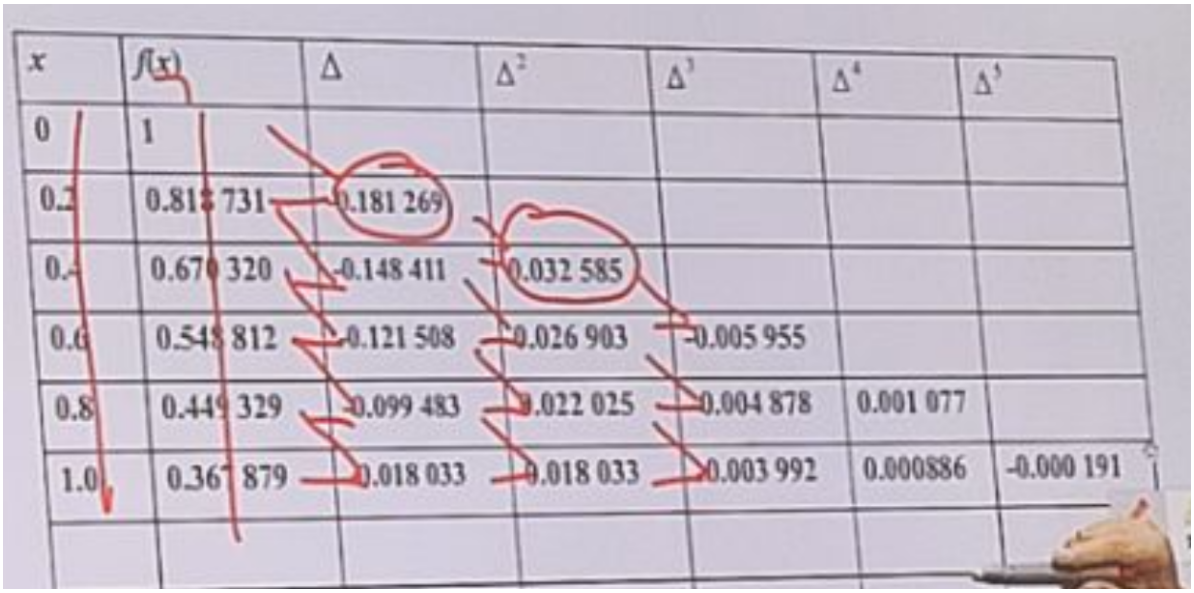
令$x = x_0 + sh(s < 0)$，则Newton插值公式和余项可以表示为(n阶向前差分)：

$$N_n(x) = N_n(x_0 + sh) = f_n + \frac{s}{1}\nabla f_0 + \frac{s(s+1)}{2!}\nabla^2 f_n + \ldots + \frac{s(s+1)\ldots(s+n-1)}{n!}\nabla^n f_n$$

$$R_n(x) = \frac{h^{(n+1)}}{(n+1)!}s(s+1)\ldots(s+n-1)f^{(n+1)}(\xi), \xi \in (x_0 - nh, x_n)$$

同样，可以给出差分表:





x	$f(x)$	Δ	Δ^2	Δ^3	Δ^4	Δ^5
0	1					
0.2	0.818 731	-0.181 269				
0.4	0.670 320	-0.148 411	0.032 585			
0.6	0.548 812	-0.121 508	0.026 903	-0.005 955		
0.8	0.449 329	-0.099 483	0.022 025	-0.004 878	0.001 077	
1.0	0.367 879	-0.018 033	0.018 033	-0.003 992	0.000886	-0.000 191

4. Hermite插值多项式

为在不增加插值节点的前提下提高插值精度,一种可行的方法是: 不但要求插值函数 $H(x)$ 与被插值函数 $f(x)$ 在节点$x_0, x_1, \cdots, x_n$ 处的值相等,而且要求两者的导数值也在各节点

$$H^{(j)}(x_i) = f^{(j)}(x_i), \quad j = 0, 1, \cdots, m_i \quad i = 0, 1, \cdots, n$$

这种方法称为 Hermite 插值法. 若其诸节点 x_i 互异,且插值函数 $H(x)$ 取为代数多项式 $\Phi(x)$,则确定多项式 $\Phi(x)$ 共需$n + \sum_{i=0}^{n} m_i + 1$ 个插值条件.为简单起见,下面我们将仅讨论 $2n+1$次 Hermite 代数插值问题

$$H(x_i) = f(x_i), \quad H'(x_i) = f'(x_i), \quad i = 0, 1, \cdots, n.$$

$$H_{2n+1}(x) = \sum_{j=0}^{n} [y_j \alpha_j(x) + y'_j \beta_j(x)],$$

记 $y_j = f(x_j), y'_j = f'(x_j)$, 则下列结果给出了$2n+1$次 $Hermite$代数插值问题的解.

定理：$2n+1$次 Hermite 插值问题 的解 $H_{2n+1}(x)$ 存在且唯一,其表达式为

$$H_{2n+1}(x) = \sum_{j=0}^{n} [y_j + (x - x_j)(y'_j - 2y_j l'_j(x_j))] l_j^2(x), \quad l_j(x) = \prod_{i=0, i \neq j}^{n} \frac{x - x_i}{x_j - x_i}.$$

误差如下：

$$R_{2n+1}(x) \triangleq f(x) - H_{2n+1}(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!} \omega_{n+1}^2(x), \quad x \in [a, b]$$

其中 ξ 在诸x_i 与x之间，$\omega_{n+1}(x) = \prod_{i=0}^{n} (x - x_i)$.