AI1807 Numerical Analysis

Xiyuan Yang
2025.10.09

Lecture Notes and Code for AI 1807, Numerical Analysis

Contents

§1. Introduction

§1.1. Definition

- 数值计算方法、理论和计算实现
- 作为计算数学的一部分
- 精读和误差分析在计算机领域至关重要。

§1.2. Error

§1.2.1. Definition

误差来源：

- 模型误差（建模时产生）
- 观测误差
- 截断误差 & 方法误差 (Truncation Error)
  - 求近似解

- 舍入误差 (RoundOff Error)
  - 机器字长有限

§1.2.2. Several Examples for Error

Example (Error in Polynomial Computation).
- 直接计算会导致多次昂贵且无意义的乘法操作
- 使用秦九韶算法可以减少乘法操作的次数
- 更优的算法优化：因式分解

Example (Solving Matrix).
求解 $Ax = b$, we need:
- 使用克莱姆法则，则求解 n 个未知数需要 n+1 次矩阵行列式运算。
- 基于代数余子式的计算行列式的方法达到了 $O(n!)$ 的时间复杂度
- 行列式计算优化：$O(n^3)$

更少的运算次数往往意味着更少的误差！

Example (Error for integrate).
在实际计算的过程中，往往需要考虑更多和理论计算有差异的部分，例如：

$$I_n = \frac{1}{e}\int_0^1 x^n \mathrm{d}x$$

We have:

$$I_n = 1 - nI_{n-1}$$

- 在实际计算中，如果从 I_0 开始计算，因为多次乘法操作的实现，会导致在 n 非常大的时候，浮点数误差很大，精度低。
- 精度更高的方法：估值

$$I_{n-1} = \frac{1}{n}(1 - I_n)$$

- 首先误差分析确定上下限：$\frac{1}{e(n+1)} < I_n < \frac{1}{n+1}$
- 取中间值进行估计，再倒回去计算到 I_0
- 虽然进行了很多次乘法操作，但是在这个操作中误差逐渐减小。

Recordings (Explanation).
- 解释：在正向递推式中，浮点数乘法带来的误差 ε 会随着 n 的增大而不断的被放大
- 但是在逆向递推式中，一开始的插值误差因为 $\frac{1}{n}$ 的缩减效应导致其被减小。

§1.2.3. Absolute Error

$$e^* = x^* - x$$

误差限：误差的绝对值的上界

$$|x - x^*| \leq \varepsilon^*$$

§1.2.4. Relative Error

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

实际计算中通常取 x^* 的实际值作为分母。

相对误差限：

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

§1.2.5. Significant Figures

§1.2.5.1. Quick Judgement

- 对于四舍五入的有效数字评判（这也是一般情况），可以按照数数位的方式进行判断有效数字的位数 n
- 有效数字的设计和科学计数法无关，可以实现科学计数法的归一化

$$x^* = \pm 10^m (a_1.a_2a_3a_4...a_n)$$

§1.2.5.2. Strict Definition

Given the original number x and the truncated number x^*:

$$x^* = \pm 10^m (a_1.a_2a_3a_4...a_n), a_i \in \{0, 1, 2, ..., 9\}$$

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

- 相对误差限：$\frac{1}{2} \times 10^{m-n+1}$ 为有效数字定义的相对误差限。
- 有效数字：n

Recordings (Significant figures).
- 找有效数字 n 的方法和高中一样
- 找移位 m 的方法就是转变成科学计数法
- 找相对误差限 $m + n - 1$ 看小数点后有几位数字

Example (An Example for significant figure).
考虑 $x = 3.14159265357$ and different x^*:

- $x^* = 3.1416$:
  - $x^* = 10^0 \times 3.1416$, $m = 0$
  - $|x - x^*| \approx 0.0000073 \leq 0.00005 = \frac{1}{2} \times 10^{-4}$, $m - n + 1 = -4$
  - $n = 5$
- $x^* = 3.1415$:
  - $x^* = 10^0 \times 3.1415$, $m = 0$
  - $|x - x^*| \approx 0.0000927 \leq 0.0005 = \frac{1}{2} \times 10^{-3}$, $m - n + 1 = -3$
  - $n = 4$

§1.3. Python and Numerical Analysis

```python
def demo_2():
    print("0.1 + 0.2 == 0.3? ", (0.1 + 0.2 == 0.3))
    print(0.1 + 0.2)
```

The answer is:

```
0.1 + 0.2 == 0.3?  False
0.30000000000000004
```

问题在于计算机中浮点数的存储方式，或者说，二进制的根源问题。

Recordings (About Binary and floating number).
In binary (or base-2), the only prime factor is 2, so you can only cleanly express fractions whose denominator has only 2 as a prime factor. In binary, 1/2, 1/4, 1/8 would all be expressed cleanly as decimals, while 1/5 or 1/10 would be repeating decimals. So 0.1 and 0.2 (1/10 and 1/5), while clean decimals in a base-10 system, are repeating decimals in the base-2 system the computer uses. When you perform math on these repeating decimals, you end up with leftovers which carry over when you convert the computer's base-2 (binary) number into a more human-readable base-10 representation.

Example (使用 Python 求解相对有效数字).

```python
def get_significant_figure(ref: str, est: str) -> int:
    """计算实数估计值 est 相对于实数参考值 ref 的有效数字位数

    Args:
    ref (str): 实数参考值的字符串形式
    est (str): 实数估计值的字符串形式

    Returns:
    n (int): 有效数字位数
    """
    try:
        ref_val = float(ref)
        est_val = float(est)
    except ValueError:
        raise ValueError("输入必须是有效的实数字符串。")

    if ref_val == est_val:
        return 15
    if ref_val == 0:
        return 0

    error = abs(ref_val - est_val)

    if ref_val != 0:
        ref_magnitude = math.floor(math.log10(abs(ref_val)))
    else:
        return 0

    if error != 0:
```

```
        error_magnitude = math.floor(math.log10(error))
    else:
        return 15

    sig_fig = int(ref_magnitude - error_magnitude)
    last_sig_fig_magnitude = 10**error_magnitude
    if error < 0.5 * last_sig_fig_magnitude:
        sig_fig += 1

    return max(0, sig_fig)
```

§2. 插值法

§2.1. Definition

Definition 2.1.1 (插值).
- 原函数 f 在区间 $[a,b]$ 存在定义
- $a \le x_0 < x_1 < \ldots < x_n \le b, P(x_i) = y_i$

- 多项式插值
- 分段插值
- 三角插值

§2.2. Lagrange 插值

考虑插值多项式 $P(x) \in H_n$，可证明插值多项式的存在唯一性，即集合 H_n 中有且仅有一个多项式满足插值多项式的定义。

Proof.
- 使用范德蒙行列式
- 转化为一个线性方程组问题
- 范德蒙行列式不等于 0，说明该线性方程组有唯一解

□

§2.2.1. 线性插值

- 直接求解上述的线性方程组来找到合适的解是困难且昂贵的。
- 这也是原始的解方程的办法。
- 并且因为唯一性，这样得到的解往往即为复杂，在实际情况中用处不大

因此，我们可以牺牲一些精度和准确性，采用更简单的线性插值

从基本的点斜式变形：

$$L_1(x) = \frac{x_{k+1} - x}{x_{k+1} - x_k} y_k + \frac{x - x_k}{x_{k+1} - x_k} y_{k+1}$$

这可以看做是两个一次插值基函数的线性组合。

$$l_1(x) = \frac{x_{k+1} - x}{x_{k+1} - x_k}$$

$$l_2(x) = \frac{x - x_k}{x_{k+1} - x_k}$$

§2.2.2. 抛物插值

考虑 $n = 2$ 的二次函数插值拟合。

此时需要确定三个基函数。这些基函数都是二次函数，满足：

- 两个插值点的函数值为 0
- 剩下一个插值点的函数值为 1
- 这些插值函数因为确定了零点，很容易通过零点式求解唯一的缩放参数。
  - 规定剩下一个插值点的函数值为 1 的目的也就是作为基要标准化

$$l_{k-1}(x_{k-1}) = 1, l_{k-1}(x_j) = 0, (j = k, k+1)$$

$$l_k(x_k) = 1, l_k(x_j) = 0, (j = k-1, k+1)$$

$$l_{k+1}(x_{k+1}) = 1, l_{k+1}(x_j) = 0, (j = k-1, k)$$

最终，我们可以得到插值的形式：

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

这个函数的形式非常的简洁，也非常的直观，几乎是直接构造出来而不需要任何的运算技巧！

于是最终，我们就可以得到抛物插值的基本公式：

$$L_2(x) = y_{k-1} l_{k-1}(x) + y_k l_k(x) + y_{k+1} l_{k+1}(x)$$

§2.2.3. 一般化的插值多项式

考虑有 $n+1$ 个插值点的 n 次插值多项式 $L_{n(x)}$ ：

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

$$L_{n(x)} = \sum_{i=0}^{n} y_i l_i(x)$$

考虑 $\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i)$

$$\omega'_{n+1}(x_k) = (x_k - x_0)...(x_k - x_{k-1})(x_k - x_{k+1})...(x_k - x_n)$$

$$L_{n(x)} = \sum_{i=0}^{n} \frac{y_i w_{n+1}(x)}{(x - x_i)\omega'_{n+1}(x_i)}$$

§2.2.4. 插值余项

Definition 2.2.4.1 (插值余项).

$$R_{n(x)} = f(x) - L_{n(x)}$$

Proposition 2.2.4.1 (插值余项).

$$R_{n(x)} = f(x) - L_{n(x)} = \frac{f^{n+1}(\xi)}{(n+1)!} w_{n+1}(x)$$

§2.3. 逐次线性插值法

Recordings (拉格朗日插值的问题).

- 拉格朗日插值的精度达到了理论最优，实际上，他也给出了一种可行的求解唯一的插值函数的算法
- 但是其最大的问题在于拉格朗日插值如果需要增加一个插值节点，这插值函数需要完全重新计算
  - 这在实际应用中会带来大量的计算资源的浪费
- 我们希望插值的精度是不断提升的，

$I_{i_1,i_2,i_3,\ldots,i_n}(x)$ 为函数 $f(x)$ 关于节点 $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$ 的 $n-1$ 次插值多项式。

现令 $I_{i_1,i_2,\cdots,i_n}(x)$ 表示函数 $f(x)$ 关于节点 $x_{i_1}, x_{i_2}, \cdots, x_{i_n}$ 的 $n-1$ 次插值多项式，$I_{i_k}(x)$ 是零次多项式，记 $I_{i_k}(x)=f(x_{i_k})$，$i_1, i_2, \cdots, i_n$ 均为非负整数. 一般情况，两个 k 次插值多项式可通过线性插值得到 $k+1$ 次插值多项式

$$I_{0,1,\cdots,k,l}(x) = I_{0,1,\cdots,k}(x) + \frac{I_{0,1,\cdots,k-1,l}(x) - I_{0,1,\cdots,k}(x)}{x_l - x_k}(x - x_k). \quad (2.3.1)$$

这是关于节点 $x_0, \cdots, x_k, x_l$ 的插值多项式. 显然

$$I_{0,1,\cdots,k,l}(x_i) = I_{0,1,\cdots,k}(x_i) = f(x_i)$$

对于 $i=0,1,\cdots,k-1$ 成立. 当 $x=x_k$ 时，有

$$I_{0,1,\cdots,k,l}(x_k) = I_{0,1,\cdots,k}(x_k) = f(x_k),$$

当 $x=x_l$ 时，有

$$I_{0,1,\cdots,k,l}(x_l) = I_{0,1,\cdots,k}(x_l) + \frac{f(x_l) - I_{0,1,\cdots,k}(x_l)}{x_l - x_k}(x_l - x_k) = f(x_l).$$

这就证明了式(2.3.1)的插值多项式满足插值条件，称式(2.3.1)为 Aitken **逐次线性插值公式**. 当 $k=0$ 时为线性插值. 当 $k=1$ 时插值节点为 x_0, x_1, x_l，插值多项式为

$$I_{0,1,l}(x) = I_{0,1}(x) + \frac{I_{0,l}(x) - I_{0,1}(x)}{x_l - x_1}(x - x_1).$$

Figure 1: Aitken 逐次线性插值公式

§2.3.1. Aitken Interpolation

目标是找到 n 阶插值多项式 $P_n(x)$

$P_{i,k}(x)$ 定义为通过点 $(x_i, y_i), (x_{i+1}, y_{i+1}), \ldots, (x_k, y_k)$ 的 $k-i$ 阶插值多项式。

$P_{i,i}(x) = y_i$

Recordings (Like DP?).
- 怎么一股动态规划状态转移方程的味道

从零阶多项式出发，作为初始条件不断的归纳到更高阶的多项式。

$$P_{i,k}(x) = \frac{1}{x_k - x_i} \begin{vmatrix} P_{i,k-1}(x) & x_i - x \\ P_{i+1,k}(x) & x_k - x \end{vmatrix}$$

$$= \frac{(x_k - x)P_{i,k-1}(x) - (x_i - x)P_{i+1,k}(x)}{x_k - x_i}$$

§2.4. 差商与 Newton 插值公式

§2.4.1. 差商

Definition 2.4.1.1 (差商).

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

为函数的一阶差商。

高阶差商的定义是递归定义而来的：

$$f[x_0, x_1, x_2, ..., x_k] = \frac{f[x_0, x_1, x_2, ..., x_{k-2}, x_k] - f[x_0, x_1, x_2, ..., x_{k-1}]}{x_k - x_{k-1}}$$

Recordings (Newton 插值).
- 对于 N 阶的可导函数来说，其形式和泰勒展开极为类似。

§3. Conclusion