AI1807 Numerical Analysis

Xiyuan Yang
2025.11.05

Lecture Notes and Code for AI 1807, Numerical Analysis

目录

§ 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. Floating Number

§ 2.1. 浮点数的十进制和二进制表示

Recordings Binary.

$$\sum_{-\infty}^{+\infty} b_i 2^i$$

The number $...b_2b_1b_0.b_{-1}b_{-2}...$ forms the binary digit for the original number.

- 具体的计算过程中，可以不断的乘除 2 并从小数点向左向右记录余项 0 或者 1
- 在很多情况下，小数部分的计算会形成循环，即无限循环小数
  - 这也是为什么有些在十进制下的有限小数在计算机存储中成为了二进制下的无限不循环小数，带来的精度误差
  - 因此在具体的转换过程中可以使用方程求解

Example For the digit?.
Suppose $x = 0.3$:
- $0.3 * 2 = 0.6$: mod 0
- $0.6 * 2 = 1.2$: mod 1
- $0.2 * 2 = 0.4$: mod 0

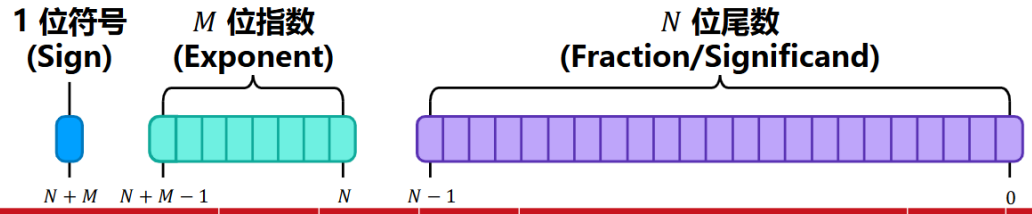
§ 2.2. 实数在计算中的浮点表示

在 Python 中，默认为双精度 fp64 的表示方式。

§ 2.2.1. IEEE 754 二进制浮点数算数标准

实数在计算机中的浮点表示（Floating-point）

- IEEE 754 二进位浮点数算术标准
  - ❑ 正规化的浮点数表示：$\pm 1.b_1 b_2 b_3 b_4 \cdots b_N \times 2^p$
  - ❑ 其中尾数 $b_2, b_3, b_4 \cdots b_N \in \{0,1\}$；指数项 p 是有符号整数



精度	符号	指数	尾数	可表示的最大/最小数值	机器精度	有效数字
半精度 (16 bit, FP16)	1 bit	5 bit	10 bit	$\pm(2-2^{-10})\cdot 2^{15} = \pm 65504$	2^{-10}	约3~4
单精度 (32 bit, FP32)	1 bit	8 bit	23 bit	$\pm(2-2^{-23})\cdot 2^{127}$	2^{-23}	约6~9
双精度 (64 bit, FP64)	1 bit	11 bit	52 bit	$\pm(2-2^{-52})\cdot 2^{1023}$	2^{-52}	约15~17

为了方便比较大小，我们希望将不同的浮点数直接看做是有符号整数来比较大小。在未经过移码之前，指数部分的取值范围是：

$$-2^{E-1}+2 \le e \le 2^{E-1}-1$$

$$\begin{aligned} \text{Bias} &= 2^{E-1}-1 \\ E' &= e + \text{Bias} \\ 1 &\le E' \le 2^E - 2 \end{aligned}$$

在加上这一部分后，指数部分的数值范围变成 $[1, 2^M - 2]$.

补码表示法中，负数的最高位是 1，正数的最高位是 0。因此，负数的补码值在无符号比较时会大于正数。这会打乱数值的大小关系。而加上移码之后，就更方便比较。

Recordings +0/-0.

- 在浮点数中，将指数和尾数部分设置为全 0 就可以得到 0 值
- 但是符号位可以选择正或者负：
  - $+0$: Sign=1
  - -0: Sign=0
- 更方便计算正无穷和负无穷

在计算中，通常受到字长的限制，需要将尾数部分截断。

§ 2.2.2. 浮点数的算术运算误差

在实际运算过程中，浮点数的加法和乘法均会带来舍入误差 Round-of Error.

§ 2.2.2.1. 浮点数加法

Definition 2.2.2.1.1 Plus.

- 补齐阶码
  - 严重的舍入误差出现在这一步！
- 尾数求和
- 正规化

例

计算 $1+3\times2^{-53}$：

$$
\begin{aligned}
& (1+3\times2^{-53})_{10} \\
=\ & (1.00\cdots0)_2\times2^0+(11.0\cdots0)_2\times2^{-53} \\
=\ & 1.\boxed{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000}\times2^0 \quad \text{① 对阶} \\
+\ & 0.\boxed{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001}\ 1\times2^0 \\
=\ & 1.\boxed{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001}\ 1\times2^0 \quad \text{② 尾数求和} \\
\to\ & 1.\boxed{0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010}\times2^0 \quad \text{③ 正规化} \\
=\ & (1+2^{-51})_{10}
\end{aligned}
$$

§ 2.2.2.2. 浮点数乘法

Definition 2.2.2.2.1 浮点数乘法.

$$
\begin{aligned}
g_3 &= (1+f_1)\times(1+f_2) \\
p_3 &= p_1+p_2 \\
s_3 &= s_1\oplus s_2
\end{aligned}
$$

可以看到，浮点数乘法是相当昂贵的操作。

Recordings Error Analysis.

- 大数吃小数的过程中会导致在阶码对齐的过程中损失相当多的精度

- $x \gg y$: $\frac{x}{y}$ is not recommended!
  - Overflow
- $x \approx y$: $x - y$ is not recommended!
- 运算顺序要注意，避免大数吃小数
- 使用相关数学运算技巧，例如分母分子有理化等。不过先估值估计误差很重要。

§ 3. 插值法

§ 3.1. Definition

Definition 3.1.1 插值.
- 原函数 f 在区间 $[a, b]$ 存在定义
- $a \leq x_0 < x_1 < ... < x_n \leq b, P(x_i) = y_i$

- 多项式插值
  - 线性插值
  - 抛物线插值
  - 分段插值
  - 样条插值
- 三角插值
- 有理插值

§ 3.2. 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

□

Recordings Using Vandermonde Matrix.
- 这个是一般的解法。
- 只要保证每一个采样点的 x 值不同，就可以保证左侧的范德蒙行列式不为零，对应的范德蒙矩阵可逆
- 因此可以唯一计算具体的多项式的参数向量 $[a_0, a_1, ..., a_{n-1}]$

§ 3.2.1. 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。

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

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

§ 3.2.2. 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：

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

§ 3.2.3. 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 $L_{n(x)}$：

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

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

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

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

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

Recordings Definition for Lagrange.

- 从原理上看，本质上只是找到对应多项式空间的一组基，在范德蒙行列式中，因为 coefficient 被直接求出，相当于使用了 $\{x^i\}$ 的一组基本基。
- 对于拉格朗日求解的方法，本质上就是求解一组基向量，在给定系数的情况下

$$l_{j(x)} = \begin{cases} 1 \text{ if } k = j \\ 0 \text{ if } k \neq j \end{cases}$$

§ 3.2.4. 插值余项

Definition 3.2.4.1 插值余项.

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

Proposition 3.2.4.1 插值余项.

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

§ 3.3. 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

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

图 3 Aitken 逐次线性插值公式

§ 3.3.1. 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 $(x_i,y_i),(x_{i+1},y_{i+1}),...,(x_k,y_k)$ 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。

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

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

§ 3.4. 差商与 Newton 插值公式

§ 3.4.1. 差商

Definition 3.4.1.1 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。

$$f(x) = f(x_0) + f[x, x_0](x - x_0)$$

Until··· (This is the definition of that!)

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

从 $f(x) = f(x_0) + f[x, x_0](x - x_0)$，将差商不断展开到高价差商，直到写成如下的形式：

$$f(x) = f(x_0) + \sum_{i=1}^{n} f[x_0, x_1, ..., x_i] \prod_{j=0}^{i-1}(x - x_j) + f[x, x_0, ..., x_n] \prod_{j=0}^{n}(x - x_j)$$

Recordings 差商的几何意义.

- 对于二阶差商，对于连续可导函数来说，二阶差商的极限值就是函数在该点处的导数

$$\lim_{h \to 0} f[x_0, x_0 + h] = f'(x_0)$$

- 对于更高阶的差商

$$\lim_{x_0, ..., x_n \to x} f[x_0, ..., x_n] = \frac{f^{(n)}(x)}{n!}$$

Recordings 差商的更清晰的代数性质.

- 差商具有轮换对称性

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

- 差商可以看成函数值的线性组合

$$f[x_0, x_1, x_2, ..., x_k] = \sum_{j=0}^{k} \frac{f(x_j)}{\prod_{i=0}^{k}(x_j - x_i)(i \neq j)}$$

- 差商和 k 阶导数存在相等的关系（只要高阶导数存在）

$$f[x_0, x_1, ..., x_k] = \frac{f^{(k)}(\varphi)}{k!}$$

- 同时，差商也可以保证线性性

§ 3.4.2. 差商表的计算

差商（Difference Quotient)

- 如何计算差商?
  - ❑ 差商表

差商的等价定义

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

x_i	$f(x_i)$	一阶差商	二阶差商	$\cdots$	k 阶差商
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	$f(x_3)$	$f[x_2, x_3]$	$f[x_1, x_2, x_3]$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	
x_k	$f(x_k)$	$f[x_{k-1}, x_k]$	$f[x_{k-2}, x_{k-1}, x_k]$	$\cdots$	$f[x_0, x_1, \cdots, x_k]$

Recordings 计算差商表的注意事项.

- 差商表类似于动态规划是逐步计算完成的
- 使用的是差商的等价定义
- 注意差商计算时的分子两个值作差需要依靠上游的两个差商的计算值，但是分母是 $x_k - x_0$

§ 3.4.3. Newton 插值

$$N_n(x) = f(x_0) + \sum_{i=1}^{n} f[x_0, x_1, ..., x_i] w_i(x)$$

$$w_i(x) = \prod_{j=0}^{i-1} (x - x_j)$$

$$R_n(x) = f(x) - N_n(x) = f[x_0, x_1, ..., x_n, x] w_{n+1}(x)$$

Recordings Newton 插值.

- 遵循就近原则，优先选取距离 x 插值位置更近的节点

- 插值节点无需按大小排列
- 最终计算插值多项式只需要使用差商表中对角线部分的值
- 增加插值点的时候，新增加的插值点必须在原先插值点的后面
  - 和拉格朗日插值方法相比，这样的插值方法可以实现高效复用！

§ 3.4.4. Time complexity

§ 3.4.4.1. Lagrange

- 计算基函数也需要 $O(n^2)$ 的时间复杂度
- 通过分治计算的优化可以降低时间复杂度为 $O(n\log^2 n)$

§ 3.4.4.2. Newton

对于牛顿插值法，很显然最关键的复杂度在于计算差商表：$O(n^2)$

§ 3.5. 等距节点插值公式

Definition 3.5.1 等距节点的插值.

$$x_k = x_0 + kh$$

Definition 3.5.2 差分运算符.

$$\Delta f_k \triangleq= f_{k+1} - f_k$$

$$\Delta^m f_k \triangleq \Delta^{m-1} f_{k+1} - \Delta^{m-1} f_k$$

$$\nabla f_k \triangleq f_k - f_{k-1}$$

$$\nabla^m f_k \triangleq \nabla^{m-1} f_k - \nabla^{m-1} f_{k-1}$$

Definition 3.5.3 算子.
不变算子 I:

$$If_k = f_k$$

$$Ef_k = f_{k+1}$$

$$\Delta = E - I$$

$$\nabla = I - E^{-1}$$

在等距节点的情况下，可以实现对前面的插值节点的定义进行简化：

$$f[x_0, x_1, ..., x_k] = \frac{\Delta^k f_0}{k!h^k}$$

$$f[x_k, x_{k+1}, ..., x_{k+m}] = \frac{\Delta^m f_k}{m!h^m}$$

§ 3.6. Hermite Interpolation

Definition 3.6.1 更高要求的插值函数.

$$f(x) \approx g(x)$$

$$p^{(j)}(x_i) = f^{(j)}(x_i)$$

只考虑一阶导数，并且函数值和导数值的个数相等的情况。一共有 $2n+2$ 个条件，因此可以唯一确定一个次数不超过 $2n+1$ 的多项式。

$$f(x_j) = g(x_j),\ f'(x_k) = g'(x_j)\ (0 \le j \le n-1)$$

Recordings Selecting the base function.

$$H(x) = H_{2n+1}(x)$$

对于上面的函数，直接确定系数矩阵 coefficient 是非常困难的（相当于选择标准基），原则上，你只需要选择一组正交基即可。因此确定基函数非常重要！

$$H(x_i) = y_i$$
$$H'(x_i) = m_i$$

We will do the same thing like lagrange:

$$H_{2n+1}(x) = \sum_{i=1}^{n} y_i \alpha_i(x) + m_i \beta_{i(x)}$$

基函数需要满足：

$$\alpha_i(x_k) = \begin{cases} 1 \text{ if } k = i \\ 0 \text{ if } k \neq i \end{cases}, \alpha'(x_k) = 0$$

$$\beta_i'(x_k) = \begin{cases} 1 \text{ if } k = i \\ 0 \text{ if } k \neq i \end{cases}, \beta_{x_k} = 0$$

可以看到，相比于拉格朗日的条件，$\alpha(x)$ 还多了一个导数为 0 的额外限制，因此可以尝试用 n 次拉格朗日基函数 $l_i(x)$ 来表示 $\alpha_i(x)$。

$$\alpha(x) = (ax+b)l_i^2(x)$$

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

$$l_j'(x_j) = \sum_{\substack{k=0 \\ k \neq j}}^{n} \frac{1}{x_j - x_k}$$

带入具体的值，可得：

$$a = -2l_i'(x_i) =$$
$$b = 1 + 2x_i l_i'(x_i)$$

$$\alpha_j(x) = \left[1 - 2(x - x_j)\sum_{\substack{k=0 \\ k\neq j}}^{n} \frac{1}{x_j - x_k}\right] l_j^2(x)$$

同样的方法，可以求解 $\beta_i(x)$ 的函数值，使用待定系数法。

Hermite 插值多项式的基函数

- 如何求解基函数 $\beta_i(x)$?

$$\beta_i'(x_k) = \begin{cases} 1, & k = i, \\ 0, & k \neq i, \end{cases} \qquad \beta_i(x_k) = 0 \quad (i, k = 0, 1, \cdots, n)$$

- 尝试用 n 次 Lagrange 基函数 $l_i(x)$ 表示 $\beta_i(x)$

$$\beta_i(x) = (cx + d)\, l_i^2(x)$$

❑ ⇨ $\beta_i'(x_k) = c \cdot l_i^2(x) + (cx_k + d) \cdot 2l_i(x_k) \cdot l_i'(x_k) = l_i(x_k)$

$\beta_i(x_k) = (cx_k + d)\, l_i^2(x_k) = 0$

❑ 化简可得 $\begin{cases} c + 2(cx_i + d)l_i'(x_i) = 1 \\ cx_i + d = 0 \end{cases}$ ⇨ $\begin{cases} c = 1 \\ d = -x_i \end{cases}$

❑ ⇨ $\beta_i(x) = (x - x_i)l_i^2(x)$

Theorem 3.6.1.

$$H_{2n+1}(x) = \sum_{i=1}^{n} y_i \alpha_i(x) + m_i \beta_{i(x)}$$

$$\alpha_j(x) = \left[1 - 2(x - x_j)\sum_{\substack{k=0 \\ k\neq j}}^{n} \frac{1}{x_j - x_k}\right] l_j^2(x)$$

$$\beta_j(x) = (x - x_j)l_j^2(x)$$

$$R(x) = f(x) - H_{2n+1}(x) = \frac{f^{(2n+2)}(\xi)}{(2n+2)!}\omega_{2n+1}^2(x)$$

§ 3.7. Several Codes for interpolation

§ 3.7.1. Using Vandermonde Matrix

```python
import numpy as np
import matplotlib as mlp
mlp.use("Agg")
import matplotlib.pyplot as plt
import time
from tqdm import trange
from typing import List, Tuple, Callable, Dict, Any, Union
from scipy.interpolate import lagrange, KroghInterpolator


class InterpolationSolver:
    """
    A class for solving polynomial interpolation problems.

    This solver uses various methods to find the coefficients of the unique
    polynomial that passes through a given set of points. It supports
    pluggable methods and includes time measurement for the solution process.
    """

    def __init__(self, methods: Dict[str, Callable] = None):
        """
        Initializes the Interpolation Solver.

        The default method provided is 'vandermonde' (using the Vandermonde matrix).

        Args:
            methods: A dictionary where keys are the method names (str) and
                     values are the corresponding solving functions (Callable).
                     The signature of a solving function should be:
                     f(points: List[Tuple[float, float]]) -> np.ndarray.
                     Custom methods will be merged with the default ones.
        """
        # Default method: Vandermonde matrix solution
        # * will add more solvers in the future
        self.methods: Dict[str, Callable] = {
            "vandermonde": self._solve_vandermonde,
            "lagrange": self._solve_lagrange,
            "lagrange_fast": self._solve_lagrange_fast,
        }

        if methods:
            self.methods.update(methods)

        self.last_result: Union[np.ndarray, None] = None
        self.last_method: Union[str, None] = None
        self.last_time: Union[float, None] = None

    def _solve_vandermonde(self, points: List[Tuple[float, float]]) -> np.ndarray:
        """
        Solves for the polynomial coefficients using the Vandermonde matrix method.

        For n+1 data points, the method solves the linear system V * a = y,
        where V is the Vandermonde matrix and a is the vector of coefficients.

        Args:
            points: A list of (x, y) coordinate tuples. Must contain at least
```

```
            one point.

    Returns:
        np.ndarray: The array of polynomial coefficients, ordered from
                    highest degree to lowest degree:
                    [a_n, a_{n-1}, ..., a_1, a_0]
                    for the polynomial P(x) = a_n * x^n + ... + a_0.
                    Returns an empty array if no points are given.

    Raises:
        ValueError: If the linear system is singular (e.g., duplicate x-values
                    or ill-conditioned data), preventing a unique solution.
    """
    n_points = len(points)
    if n_points == 0:
        return np.array([])

    n_degree = n_points - 1  # Degree of the polynomial

    # Separate x and y coordinates
    x = np.array([p[0] for p in points])
    y = np.array([p[1] for p in points])
    V = np.vander(x, n_points)

    # Solve the linear system V * a = y
    try:
        coefficients = np.linalg.solve(V, y)
    except np.linalg.LinAlgError as e:
        raise ValueError(
            f"Failed to solve the linear system. The matrix "
            f"might be singular (e.g., duplicate x-values). Error: {e}"
        )

    return coefficients

def _solve_lagrange_fast(self, points: List[Tuple[float, float]]) -> np.ndarray:
    x = np.array([p[0] for p in points])
    y = np.array([p[1] for p in points])
    coeff = lagrange(x, y).coef
    return coeff

def _solve_lagrange(self, points: List[Tuple[float, float]]) -> np.ndarray:
    """
    使用拉格朗日插值法展开并求和，以获得标准多项式系数。

    P(x) = sum_{j=0}^{n} y_j * l_j(x)，其中 l_j(x) 是拉格朗日基多项式。

    Args:
        points: 包含 (x, y) 坐标点的列表。

    Returns:
        np.ndarray: 多项式系数数组，顺序为从高次到低次：
                    [a_n, a_{n-1}, ..., a_1, a_0]。
    """
    n_points = len(points)
    if n_points == 0:
```

```
            return np.array([])

        x = np.array([p[0] for p in points])
        y = np.array([p[1] for p in points])

        # 初始化最终的多项式系数为零。系数顺序: [a_n, ..., a_0]
        final_coeffs = np.zeros(n_points)

        # 迭代计算每个基多项式 l_j(x) 的贡献
        for j in range(n_points):
            x_j = x[j]
            y_j = y[j]

            # compute pi (x - x_j) (k != j)
            roots = np.delete(x, j)

            # 使用 np.poly() 计算多项式 N_j(x) 的系数 (即 x - r1)(x - r2)...
            # 返回的系数顺序是 [a_m, a_{m-1}, ..., a_0]
            numerator_coeffs = np.poly(roots)

            # 计算分母 D_j = product_{k != j} (x_j - x_k)
            denominator = np.prod(x_j - roots)

            if np.isclose(denominator, 0):
                raise ValueError("Error, repeated x value is found.")

            # l_j(x) 的系数 = N_j(x) 的系数 / D_j
            l_j_coeffs = numerator_coeffs / denominator
            term_coeffs = l_j_coeffs * y_j
            final_coeffs = np.polyadd(final_coeffs, term_coeffs)

        return final_coeffs

    def _format_polynomial(self, coefficients: np.ndarray) -> str:
        """
        Formats the polynomial coefficients into a readable string representation.

        Args:
            coefficients: The array of polynomial coefficients
                          [a_n, a_{n-1}, ..., a_0].

        Returns:
            str: The string representation of the polynomial, e.g., "1.0x^3 + 2.0x +
1.0".
        """
        if len(coefficients) == 0:
            return "0"

        terms = []
        n_degree = len(coefficients) - 1

        for i, a in enumerate(coefficients):
            degree = n_degree - i

            if np.isclose(a, 0):
                continue
```

```
            sign = " + " if a > 0 else " - "
            if not terms:
                sign = "" if a > 0 else "-"

            abs_a = abs(a)
            coeff_str = f"{abs_a:.6f}"
            if np.isclose(abs_a, 1) and degree != 0:
                coeff_str = ""

            if degree == 0:
                # Constant term
                term_str = f"{sign}{abs_a:.6f}"
            elif degree == 1:
                # x term
                term_str = f"{sign}{coeff_str}x"
            else:
                # x^k term
                term_str = f"{sign}{coeff_str}x^{degree}"

            terms.append(term_str)

        return "".join(terms).strip() or "0"

    def solve(
        self, points: List[Tuple[float, float]], method: str = "vandermonde"
    ) -> Dict[str, Any]:
        """
        The core function to solve the polynomial interpolation using the specified
method.

        The function measures the time taken by the chosen solving method.

        Args:
            points: A list of (x, y) coordinate tuples for interpolation.
            method: The name of the interpolation method (str) to use, which
                    must exist as a key in `self.methods`. Defaults to "vandermonde".

        Returns:
            Dict[str, Any]: A dictionary containing the solution results:
                            - 'coefficients': np.ndarray of the polynomial
coefficients.
                            - 'method': The name of the method used.
                            - 'time_s': The elapsed time for the calculation in
seconds.
                            - 'polynomial_str': A readable string of the polynomial
P(x).

        Raises:
            ValueError: If the specified `method` is not recognized.
            ValueError: If the underlying solver function fails (e.g., singular
matrix).
        """
        if method not in self.methods:
            raise ValueError(
                f"Unknown interpolation method: '{method}'. "
```

```
                f"Available methods: {list(self.methods.keys())}"
            )

        solver_func = self.methods[method]

        if not points:
            return {
                "coefficients": np.array([]),
                "method": method,
                "time_s": 0.0,
                "polynomial_str": "0",
            }

        start_time = time.time()
        coefficients = solver_func(points)
        end_time = time.time()
        elapsed_time = end_time - start_time
        poly_str = self._format_polynomial(coefficients)

        # Store results in the cache
        self.last_result = coefficients
        self.last_method = method
        self.last_time = elapsed_time

        return {
            "coefficients": coefficients,
            "method": method,
            "time_s": elapsed_time,
            "polynomial_str": poly_str,
        }


if __name__ == "__main__":
    solver = InterpolationSolver()
    NUM_POINTS = 50
    NUM_RUNS = 100
    TEST_METHODS = ["vandermonde", "lagrange", "lagrange_fast"]
    results_by_method = {method: [] for method in TEST_METHODS}

    print(f"--- Efficiency Test ---\nPoints N: {NUM_POINTS}, Repeated Nums:
{NUM_RUNS}")

    for run in trange(NUM_RUNS):
        points_data = [
            (np.random.random() * 100, np.random.random() * 100)
            for _ in range(NUM_POINTS)
        ]

        for method_name in TEST_METHODS:
            try:
                result = solver.solve(points_data, method=method_name)
                results_by_method[method_name].append(result["time_s"])

            except np.linalg.LinAlgError:
                print(
                    f"Warning: {method_name} failed at run {run+1} due to singular
```

```
matrix."
                )
                continue
            except ValueError as e:
                print(f"Warning: {method_name} failed at run {run+1} with error:
{e}")
                continue

    print("\n--- Efficiency Test End ---")

    vandermonde_times = results_by_method["vandermonde"]
    lagrange_times = results_by_method["lagrange"]
    lagrange_fast_times = results_by_method["lagrange_fast"]

    # 打印统计摘要
    print("\n--- Cost Time Count ---")
    for method, times in results_by_method.items():
        if times:
            print(
                f"  {method.ljust(15)}: Mean = {np.mean(times):.6f}, Median =
{np.median(times):.6f}, Min = {np.min(times):.6f}, Max = {np.max(times):.6f}
(N={len(times)})"
            )
        else:
            print(f"  {method.ljust(15)}: No data recorded.")

    # 绘制箱线图
    data_to_plot = [vandermonde_times, lagrange_times, lagrange_fast_times]
    labels = TEST_METHODS

    plt.figure(figsize=(10, 6))
    plt.boxplot(data_to_plot, tick_labels=labels, patch_artist=True, vert=True)

    plt.title(
        f"Polynomial (N={NUM_POINTS}, {NUM_RUNS} repetitions)", fontsize=14
    )
    plt.ylabel("Eecutions times", fontsize=12)
    plt.xlabel("Method", fontsize=12)
    plt.grid(axis="y", linestyle="--", alpha=0.7)
    plt.savefig("./images/interpolation.pdf")
```

§ 3.8. 样条插值

Definition 3.8.1 Spline Interpolation.
$a \leq x_0 < x_1 < \ldots < x_n \leq b$, S in C^2 [a,b] are n times polynomial.

例如，我们考虑三次样条插值：

- 函数在区间内二阶导数连续（非常严格的限制 !）
- $S(x_k) = y_k$
- 函数在每个区间内都是三次多项式

For x in $[x_k, x_{k+1}]$:

- $S(x) = a_k + b_k x + c_k x^2 + d_k x^3$

§ 3.8.1. Definition

We have such properties:

- $S(x_k) = y_k$
- $S(x_k^-) = S(x_k^+)$
- $S'(x_k^-) = S'(x_k^+)$
- $S''(x_k^-) = S''(x_k^+)$

The above equations use $4n-2$ equations to solve $4n$ variables. We need more!

边界条件：实际问题通常对样条函数在两个端点 a, b 处的状态有要求:

$$S(x_0^+) = S(x_n^+)$$

$$S'(x_0^+) = S'(x_n^+)$$

$$S''(x_0^+) = S''(x_n^+)$$

The, we make this Spline Interpolation cyclic!

§ 3.8.2. Another Definition

We can make the interpolation more fluent:

如何求解三次样条插值函数?

- 另一种表示方法

$$S(x) = \begin{cases} s_0(x) \overset{\text{def}}{\equiv} a_0 + b_0x + c_0x^2 + d_0x^3 & \text{若 } x \in [x_0, x_1] \\ s_1(x) \overset{\text{def}}{\equiv} a_1 + b_1x + c_1x^2 + d_1x^3 & \text{若 } x \in (x_1, x_2] \\ \vdots & \\ s_{n-1}(x) \overset{\text{def}}{\equiv} a_{n-1} + b_{n-1}x + c_{n-1}x^2 + d_{n-1}x^3 & \text{若 } x \in (x_{n-1}, x_n] \end{cases}$$

- 插值条件: $S(x_k) = y_k \quad (k = 0, 1, \cdots, n)$
- 二阶导数连续: $\begin{cases} s_{k-1}(x_k) = s_k(x_k^+) \\ s'_{k-1}(x_k^-) = s'_k(x_k^+) \\ s''_{k-1}(x_k^-) = s''_k(x_k^+) \end{cases} \quad (k = 1, \cdots, n-1)$
- 边界条件

§ 3.8.3. Solving Spline Interpolation

在厄尔米特插值的过程中，整体的插值公式如下:

$$H_{2n+1}(x) = \sum_{i=1}^{n} y_i\alpha_i(x) + m_i\beta_{i(x)}$$

For the interval:

$$s_k(x) = y_k\alpha_k(x) + y_{k+1}\alpha_{k+1}(x) + m_k\beta_{k(x)} + m_{k+1}\beta_{k+1}(x)$$

Then solve the $s_k''(x)$ for unknown variables m_i. We can use the continuity:

$$s_{k-1}''(x_k^-) = s_k''(x_k^+)$$

Then, we can have $n-1$ equations for $n+1$ unknown variables:

$$\lambda_k m_{k-1} + 2m_k + \mu_k m_{k+1} = g_k$$

$$\lambda_k = \frac{h_k}{h_{k-1}+h_k}$$

$$\mu_k = \frac{h_{k-1}}{h_{k-1}+h_k}$$

$$g_k = 3(\lambda_k f[x_{k-1}, x_k] + \mu_k f[x_k, x_{k+1}])$$

And the boundaries:

- $s_0'(x_0^+) = m_0 = m_n = s_{n-1}'(x_n^-)$
- $s_0''(x_0^+) = s_{n-1}''(x_n^-)$

如何求解三次样条插值函数?

$h_k = x_{k+1} - x_k$

$m_k = S'(x_k)$

💡思路 1：直接利用分段三次 Hermite 插值

❑ 代入第 10 页推导的公式，并利用 $h_k = x_{k+1} - x_k$ 化简，可得

$$s_k(x) = \frac{(x-x_{k+1})^2[h_k + 2(x-x_k)]}{h_k^3} y_k + \frac{(x-x_k)^2[h_k + 2(x_{k+1}-x)]}{h_k^3} y_{k+1}$$

$$+ \frac{(x-x_{k+1})^2(x-x_k)}{h_k^2} m_k + \frac{(x-x_k)^2(x-x_{k+1})}{h_k^2} m_{k+1}$$

未知数 未知数

❑ 求二阶导数，得

$$s_k''(x) = \frac{6x - 2x_k - 4x_{k+1}}{h_k^2} m_k + \frac{6x - 4x_k - 2x_{k+1}}{h_k^2} m_{k+1}$$

$$+ \frac{6(x_k + x_{k+1} - 2x)}{h_k^3}(y_{k+1} - y_k)$$

Recordings 钳制样条.

- 样条插值需要给出两个边界条件，来保证结果的唯一性
  - 否则仅仅依靠其他约束无法确定唯一的样条差值多项式，必须要对此进行约束
- 边界条件的给出往往有以下两种形式：
  - 完备样条：给出 $f'(x_0), f'(x_n)$
  - Natural Spline: $S''(x_0) = S''(x_n) = 0$

根据不同的边界条件的给出，我们有若干种三转角方程的形式：

§ 3.8.3.1. f_0' and f_n' are known

此时相当于未知数 m_0 and m_n 都是已知量，只需要求解剩下的量。

如何求解三次样条插值函数？——三转角方程

思路 1：直接利用分段三次 Hermite 插值

$$\lambda_k m_{k-1} + 2m_k + \mu_k m_{k+1} = g_k \qquad (k = 1, 2, \cdots, n-1)$$

基于**边界条件 1**，即 $\begin{cases} s_0'(x_0^+) = f_0' \triangleq m_0 \\ s_{n-1}'(x_n^-) = f_n' \triangleq m_n \end{cases}$

⇨ 上述方程组可化简为只含 $m_1, m_2, \cdots, m_{n-1}$ 的 $n-1$ 个方程

$$\begin{bmatrix} 2 & \mu_1 & & & & \\ \lambda_2 & 2 & \mu_2 & & & \\ & \lambda_3 & 2 & \mu_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \lambda_{n-2} & 2 & \mu_{n-2} \\ & & & & \lambda_{n-1} & 2 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_{n-2} \\ m_{n-1} \end{bmatrix} = \begin{bmatrix} g_1 - \lambda_1 m_0 \\ g_2 \\ g_3 \\ \vdots \\ g_{n-2} \\ g_{n-1} - \mu_{n-1} m_n \end{bmatrix}$$

§ 3.8.3.2. f_0'' and f_n'' are known

Based on the boundaries we have, we can manually define g_0 and g_n:

$$g_0 = 2m_0 + m_1 = 3f[x_0, x_1] - \frac{h_0}{2} f_0''$$

$$g_n = m_{n-1} + 2m_n = 6f[x_{n-1}, x_n] + \frac{h_{n-1}}{2} f_n''$$

如何求解三次样条插值函数？——三转角方程

思路 1：直接利用分段三次 Hermite 插值

$$\lambda_k m_{k-1} + 2m_k + \mu_k m_{k+1} = g_k \qquad (k = 1, 2, \cdots, n-1)$$

基于**边界条件 2**，即 $\begin{cases} s_0''(x_0^+) = f_0'' \\ s_{n-1}''(x_n^-) = f_n'' \end{cases}$

⇨ 上述方程组可写成含 $m_0, m_1, \cdots, m_{n-1}, m_n$ 的 $n+1$ 个方程

$$\begin{bmatrix} 2 & 1 & & & & \\ \lambda_1 & 2 & \mu_1 & & & \\ & \lambda_2 & 2 & \mu_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \lambda_{n-1} & 2 & \mu_{n-1} \\ & & & & 1 & 2 \end{bmatrix} \begin{bmatrix} m_0 \\ m_1 \\ m_2 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_{n-1} \\ g_n \end{bmatrix}$$

§ 3.8.3.3. $m_0 = m_n$ and $s_{0''}(x_0^+) = s_{(n-1)''}(x_n^-)$

⇨ 两边除以 $\frac{1}{h_0}+\frac{1}{h_{n-1}}$，化简得 $\mu_n m_1+\lambda_n m_{n-1}+2m_n=g_n$

其中 $\mu_n \triangleq \frac{h_{n-1}}{h_0+h_{n-1}}$，$\lambda_n \triangleq \frac{h_0}{h_0+h_{n-1}}$，$g_n \triangleq 3(\mu_n f[x_0,x_1]+\lambda_n f[x_{n-1},x_n])$

张王优 数值分析：第四节 分段插值 SJTU SAI 28 /

2. 三次样条插值

如何求解三次样条插值函数？——三转角方程

思路 1：直接利用分段三次 Hermite 插值

$$\lambda_k m_{k-1}+2m_k+\mu_k m_{k+1}=g_k \qquad (k=1,2,\cdots,n-1)$$

基于**边界条件 3**，即 $\begin{cases} s_0(x_0) = s_{n-1}(x_n) \\ s_0'(x_0^+) = s_{n-1}'(x_n^-) \\ s_0''(x_0^+) = s_{n-1}''(x_n^-) \end{cases} \Rightarrow \begin{cases} m_0 = m_n \\ s_0''(x_0^+) = s_{n-1}''(x_n^-) \end{cases}$

⇨ 上述方程组可写成含 $m_1,\cdots,m_{n-1},m_n$ 的 n 个方程

$$\begin{bmatrix} 2 & \mu_1 & & & & \lambda_1 \\ \lambda_2 & 2 & \mu_2 & & & \\ & \lambda_3 & 2 & \mu_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \lambda_{n-1} & 2 & \mu_{n-1} \\ \mu_n & & & & \lambda_n & 2 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_{n-1} \\ g_n \end{bmatrix}$$

Recordings 三转角矩阵.

- 三转角方程保证系数矩阵对角元素均为 2，并且非对角元素满足 $\mu_k+\lambda_k=1<2$.
- 系数矩阵具有强对角优势，是非奇异矩阵
- 保证方程有解并且有唯一解

§ 3.8.4. Error Analysis

Theorem 3.8.4.1 Error Analysis for spline interpolation.

$$\max_{a\le x\le b}|f(x)-S(x)| \le \frac{5}{384}\max_{a<x<b}|f^{(4)}(x)|\ h^4$$

$$\max_{a\le x\le b}|f'(x)-S'(x)| \le \frac{1}{24}\max_{a<x<b}|f^{(4)}(x)|\ h^3$$

$$\max_{a\le x\le b}|f''(x)-S''(x)| \le \frac{3}{8}\max_{a<x<b}|f^{(4)}(x)|\ h^2$$

§ 3.9. 深度学习中的插值计算

在深度学习中，输入张量（通常是特征图）往往需要经过若干的处理来调整特征尺寸，具体的技术包括上采样和下采样两种。

Recordings 特征图.
- 特征图可以理解为描述神经网络过程中间状态的高阶张量。
- 例如，在卷积神经网络中：输入张量经过一层卷积核之后就可以得到形状为 (batches × tunnel × height × width) 的一个四阶张量作为特征图。
- 下面的讨论为了简单起见，考虑二阶特征图的情况。

§ 3.9.1. UpSampling

上采样是一种增大特征图尺寸的方法，具体来说对于二维特征图 $H \times W \to H' \times W', H' > H, W' > W$. 上采样的方法主要包括：

- 插值法
- 转置卷积
  - 可学习的一种上采样方法
  - 通过执行类似于反向卷积的操作来增加特征图的尺寸。它通过学习一组可训练的权重来填充新的像素值
  - 广泛应用于各种现代编码器架构中
- 反池化
  - 就是池化的方向过程
  - 反最大池化：对应的最大值的点被恢复，其他地区用 0 填充
  - 通用反池化：基于值的复制过程

下面，我们主要介绍插值在上采样的具体过程。

Recordings 插值采样.
- 插值采样是对离散的张量进行 padding 的操作

§ 3.9.1.1. Nearest Neighbor Interpolation

- 确定放大后新特征图上的每一个目标像素点
- 将新坐标映射回原特征图上的坐标，根据缩放倍数
- 对映射后的坐标进行四舍五入或取整，找到原特征图上距离新点最近的已知像素点
- 将原特征图上最近像素点的值，直接赋给新特征图上的目标位置

§ 3.9.1.2. Bilinear Interpolation

一种线性插值的方式。

- 根据缩放找到四个周围最近的已知像素点 $Q_{11}, Q_{12}, Q_{21}, Q_{22}$.
- 线性插值：
  - $R_1 = \text{interpolation}(Q_{11}, Q_{21})$
  - $R_2 = \text{interpolation}(Q_{12}, Q_{22})$
  - $R = \text{interpolation}(R_1, R_2)$

Recordings Bilinear Interpolation.
在具体的插值过程中，插值如下：
- $Q_{11} = (x_1, y_1)$
- $Q_{21} = (x_2, y_1)$

- $Q_{12} = (x_1, y_2)$
- $Q_{22} = (x_2, y_2)$

则最终根据加权得到的插值公式是：

$$\begin{aligned} V(x,y) = &\frac{x-x_1}{x_2-x_1}\frac{y-y_1}{y_2-y_1}V(x_2,y_2) \\ &+\frac{x-x_2}{x_2-x_1}\frac{y-y_1}{y_2-y_1}V(x_1,y_2) \\ &+\frac{x-x_1}{x_2-x_1}\frac{y-y_2}{y_2-y_1}V(x_2,y_1) \\ &+\frac{x-x_2}{x_2-x_1}\frac{y-y_2}{y_2-y_1}V(x_1,y_1) \end{aligned}$$

Recordings Different Sampling Ways.

- 不同的采样方式对应的精度和计算复杂度不同
- 一般来说，Bilinear Interpolation 对应的采样结果会更加精细，所需的插值时间复杂度也更高
  - 最近邻插值的插值结果更加锐利
  - Bilinear Interpolation 的插值结果更加柔和，因为他结合了更多的原始采样点数据

```python
import torch
import torch.nn.functional as F
from PIL import Image
import numpy as np
import os


def tensor_to_image(
    tensor, image_path, scale_factor, method_name, output_dir="upsampled_images"
):
    """将 PyTorch 张量转换回图片并保存。"""
    # (1, C, H, W) -> (H, W, C)
    output_np = tensor.squeeze(0).permute(1, 2, 0).numpy()

    # 将 [0, 1] 范围的浮点数转回 [0, 255] 的整数
    output_np = (output_np.clip(0, 1) * 255).astype(np.uint8)

    output_img = Image.fromarray(output_np)

    # 构造保存路径
    base_name = os.path.splitext(os.path.basename(image_path))[0]
    save_path = os.path.join(
        output_dir, f"{base_name}_{method_name}_{scale_factor}x.png"
    )

    output_img.save(save_path)
    print(f"成功保存 {method_name} 结果到: {save_path}")


def upsample_zero_padding(input_tensor, scale_factor):
```

```
    """
    通过在现有像素之间插入零值来实现上采样（不使用插值）。

    例如，如果 scale_factor=2，输入 [A, B] 会变为 [A, 0, B, 0]。

    Args:
        input_tensor (torch.Tensor): 输入张量，形状为 (N, C, H, W)。
        scale_factor (int): 上采样的比例因子。

    Returns:
        torch.Tensor: 上采样后的张量。
    """
    if scale_factor <= 1:
        return input_tensor

    N, C, H, W = input_tensor.shape

    # 1. 在 W 维度（宽度）上插入零
    # 目标宽度是 W * scale_factor。需要插入 W * (scale_factor - 1) 列零。
    # 原始张量 [N, C, H, W]

    # 增加一个维度用于保存零，然后重复原始值
    # 例如，如果 scale_factor=2，形状从 (N, C, H, W) -> (N, C, H, W, 1) -> (N, C, H, W,
2)
    temp_tensor = input_tensor.unsqueeze(-1).repeat(1, 1, 1, 1, scale_factor)

    # 现在 temp_tensor 的形状是 (N, C, H, W, scale_factor)。
    # 我们希望在最后一维的每 scale_factor 个元素中，只有第一个是原值，其余是 0。
    # 我们只需要将重复后的 tensor 的第 2 到第 scale_factor 个通道设置为 0
    if scale_factor > 1:
        # 将第 2 到第 scale_factor 个 '副本' 设置为 0。
        # 注意：Python 索引从 0 开始。
        temp_tensor[..., 1:] = 0

    # 将 W 和 scale_factor 合并为一个新的 W' 维度：  W' = W * scale_factor
    # (N, C, H, W, scale_factor) -> (N, C, H, W * scale_factor)
    upsampled_w = temp_tensor.reshape(N, C, H, W * scale_factor)

    # 2. 在 H 维度（高度）上插入零
    # 对 upsampled_w 做同样的操作，但作用于 H 维度。
    # 形状 (N, C, H, W') -> (N, C, H, 1, W')
    upsampled_w = upsampled_w.unsqueeze(3).repeat(1, 1, 1, scale_factor, 1)

    # 将第 2 到第 scale_factor 个 '副本' 设置为 0。
    if scale_factor > 1:
        # 作用于 H 维度（索引 3）
        upsampled_w[:, :, 1:, ...] = 0  # 沿着 H 维度的第二个块及其后的块都设置为 0

    # 将 H 和 scale_factor 合并为一个新的 H' 维度：  H' = H * scale_factor
    # (N, C, H, scale_factor, W') -> (N, C, H * scale_factor, W')
    final_upsampled_tensor = upsampled_w.reshape(
        N, C, H * scale_factor, W * scale_factor
    )

    return final_upsampled_tensor
```

```
def upsample_and_save(image_path, scale_factor, output_dir="images"):
    """
    Reads an image, converts it to a 4D PyTorch tensor (N, C, H, W),
    performs upsampling using Nearest Neighbor and Bilinear interpolation,
    and saves the resulting tensors as images.

    Args:
        image_path (str): The file path to the input image (e.g., 'input.jpg').
        scale_factor (int): The integer multiplier for upsampling the spatial
dimensions (H and W).
        output_dir (str, optional): The directory where the upsampled images will be
saved.
                                    Defaults to "upsampled_images".

    Returns:
        None: The function saves output files to the specified directory.

    Raises:
        FileNotFoundError: If the specified `image_path` does not exist.
        Exception: For other errors during image processing or tensor operations.

    Notes:
        1. The input image is converted to a normalized float tensor [0, 1].
        2. Nearest Neighbor interpolation results in blocky artifacts but is suitable
for
           discrete data like segmentation masks.
        3. Bilinear interpolation results in a smoother image and is generally
preferred
           for feature map scaling.
    """
    # 1. Check and create output directory
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    try:
        # 2. Read the image and convert to PyTorch Tensor (N, C, H, W)
        img = Image.open(image_path).convert("RGB")

        # PIL Image -> NumPy Array (H, W, C)
        img_np = np.array(img, dtype=np.float32) / 255.0

        # NumPy Array (H, W, C) -> PyTorch Tensor (1, C, H, W)
        input_tensor = torch.from_numpy(img_np).permute(2, 0, 1).unsqueeze(0)

        print(f"Original image size: {img.size} (W, H)")
        print(f"Original tensor shape: {input_tensor.shape}")

        # 3. Perform upsampling using different interpolation modes

        # --- A. Nearest Neighbor Interpolation ---
        upsampled_nearest = F.interpolate(
            input_tensor, scale_factor=scale_factor, mode="nearest"
        )

        # --- B. Bilinear Interpolation ---
```

```
        upsampled_bilinear = F.interpolate(
            input_tensor,
            scale_factor=scale_factor,
            mode="bilinear",
            align_corners=False,
        )

        upsampled_zero = upsample_zero_padding(
            input_tensor=input_tensor, scale_factor=scale_factor
        )

        print(f"Upsampled tensor shape: {upsampled_nearest.shape}")

        # 4. Convert Tensor back to Image and save

        def tensor_to_image(tensor, filename, method_name):
            # (1, C, H, W) -> (H, W, C)
            output_np = tensor.squeeze(0).permute(1, 2, 0).numpy()

            # Denormalize [0, 1] to [0, 255] and convert to integer type
            output_np = (output_np.clip(0, 1) * 255).astype(np.uint8)

            output_img = Image.fromarray(output_np)

            # Construct save path
            base_name = os.path.splitext(os.path.basename(image_path))[0]
            save_path = os.path.join(
                output_dir, f"{base_name}_{method_name}_{scale_factor}x.png"
            )

            output_img.save(save_path)
            print(f"Successfully saved {method_name} result to: {save_path}")

        # Save for zero padding
        tensor_to_image(upsampled_zero, "zero", "Zero")

        # Save Nearest Neighbor result
        tensor_to_image(upsampled_nearest, "nearest", "Nearest")

        # Save Bilinear result
        tensor_to_image(upsampled_bilinear, "bilinear", "Bilinear")

    except FileNotFoundError:
        print(f"Error: File not found at {image_path}")
    except Exception as e:
        print(f"An error occurred during image processing: {e}")


if __name__ == "__main__":
    image_file = "./images/Standard.png"
    upsample_factor = 10

    if not os.path.exists(image_file):
        print(f"\n--- Could not find example image '{image_file}' ---")
    else:
        print("--- Starting Upsampling Process ---")
```

```
        upsample_and_save(image_file, upsample_factor)
        print("--- Processing Complete ---")
```

§ 3.9.2. DownSampling

下采样是通过若干方式压缩信息，减小特征图尺寸的方式。可以获得全局语义信息。

- 例如在卷积神经网络中的 Pooling Layer 等操作。
- Strided Convolution
- Attention 操作也可以看做是一种下采样的方式

§ 4. Function Approximation

- 函数插值实现了简单函数 $p(x)$ 在某些特定点下和复杂函数 $f(x)$ 的相等关系，实现拟合，但是在复杂的情况下，会出现荣格现象
- 因此，在度量意义下，我们需要实现函数逼近：
  - 函数逼近
  - 曲线拟合

简单函数类：

- $p_{n(x)}$: polynomial
- $R_{nm}(x) = \frac{P_{n(x)}}{Q_{m(x)}}$
- $T(x)$: Fourier

常见的度量标准：

- 一致逼近：

$$\|f(x) - p(x)\|_\infty = \max_{a \le x \le b} |f(x) - p(x)|$$

- 平方逼近：

$$\|f(x) - p(x)\|_2 = \sqrt{\int_a^b [f(x) - p(x)]^2 \mathrm{d}x}$$

Definition 4.1 范数.

$$\|f\|_\infty = \max_{a \le x \le b} (|f(x)|)$$

满足范数的正定性，齐次性和三角不等式

§ 4.1. 最佳一致逼近

Definition 4.1.1 最佳逼近函数.

Given the functional space Φ, and function $f \in C[a, b]$, if a function $g*(x) \in \Phi$ satisfy:

$$\|f(x) - g*(x)\| = \min_{g(x) \in \Phi} \|f(x) - g(x)\|$$

对于最佳一致逼近：

$$\|f(x)-g*(x)\|_\infty=\min_{g(x)\in\Phi}\|f(x)-g(x)\|_\infty=\min_{g(x)\in\Phi}\max_{a\le x\le b}|f(x)-p(x)|$$

§ 4.1.1. 偏差点和偏差点的性质

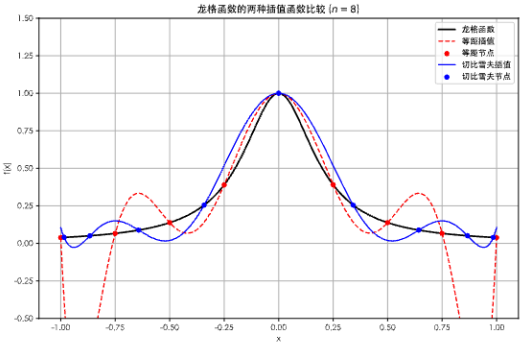
定义上偏差点就是函数值偏差最大值取到的地方，包含正偏差点和负偏差点。

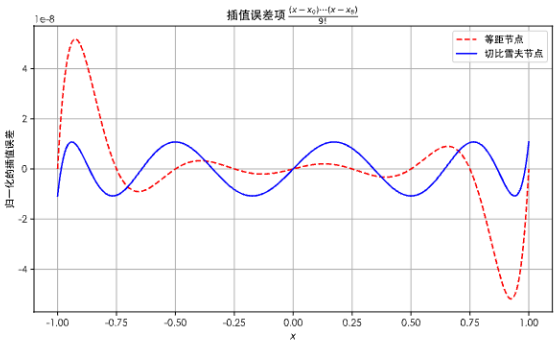
- 可以利用反证法证明一定即存在正偏差点和负偏差点
- 偏差点的偏差程度直接影响了最佳逼近的最小偏差 E_n

但是节点的选取会严重影响具体的函数表现。

Chebyshev 定理

- 回顾：多项式插值方法中，选取节点的位置对插值误差的影响
  - Lagrange 插值余项 $R_n(x)=\frac{(x-x_0)\cdots(x-x_n)}{(n+1)!}f^{(n+1)}(\xi)$
  - 考虑区间 $[-1,1]$ 上的 $f(x)=\frac{1}{1+25x^2}$，选取 9 个插值节点 $(n=8)$





- 思考：怎样选取节点，可使偏差（误差的最大值）取值最小？

Theorem 4.1.1.1 Chebyshev Theorem.
$p_{n(x)}$ 是最佳一致逼近多项式的充分必要条件： 至少有 n+2 个轮流为正负的偏差点：

$$p_n(t_k)-f(t_k)=\pm(-1)^k\ \|p_n(x)-f(x)\|_\infty$$

证明见 PPT，可以根据同号性利用反证法证明

可以推导出下面的推论：

Corollary 4.1.1.1.
$f(x)\in C[a,b]$ 的 n 次最佳一致逼近多项式存在并且唯一，并且一定是 $f(x)$ 的一个 Lagrange 多项式。

- 存在性证明：
- 唯一性证明：同一法

§ 4.1.2. 一次最佳一致逼近多项式

假设 $f(x) = C^2[a,b]$ 并且 $f''(x)$ 在区间 $[a,b]$ 内部不变号。下面求解最佳一直逼近多项式 $p_1(x) = c_0 + c_1 x$.

具体的求解过程：

因为二阶导不变号，故 $f(x)$ 的一阶导数单调，故存在唯一的点 t_2, s.t. $f'(t_2) = c_1$. 剩下的两个偏差点一定在区间的端点。

$$p_1(x) = \frac{f(a) + f(t_2)}{2} + c_1\left(x - \frac{a + t_2}{2}\right)$$

根据拉格朗日中值定理：

$$c_1 = \frac{f(b) - f(a)}{b - a} = f'(t_2)$$

$$c_0 = \frac{f(a) + f(t_2)}{2} - \frac{f(b) - f(a)}{b - a}\frac{a + t_2}{2}$$