《计算物理》第 1 次作业

1900017812 高乐耘*

2022 年 10 月 5 日

目录

I. 三种方法计算 e^{-x}

题中计算 e^{-x} 的三种方法均基于泰勒级数：

$$e^{-x} = \sum_{k=0}^{\infty} (-1)^k \frac{x^k}{k!},\ -\infty < x < +\infty \tag{1}$$

其通项为：

$$a_k = (-1)^k \frac{x^k}{k!},\ k = 0, 1, 2, ... \tag{2}$$

*电子邮件地址: seeson@pku.edu.cn 手机号: 13759115414

部分和为：

$$S_k = \sum_{i=0}^{k} a_i,\ k = 0, 1, 2, ... \tag{3}$$

笔者采用以下统一标准比较三种算法的优劣：比较在一定的浮点数系统下，对于给定的输入 x，算法可以达到的极限精度。具体地，数值变量使用 C 语言定义的双精度浮点数 `double`；算法开始时将 a_k 和 S_k 初始化为 $k = 0$ 时的值；从 $k = 1$ 开始，使用算法给定的通项计算方法计算 a_k，如果中间过程发生任何浮点算数溢出，算法终止；使用 $S_{k-1} + a_k$ 计算 S_k，如果 S_k 与 S_{k-1} 相比没有变化，算法终止；如果 S_k 溢出，算法终止；否则继续累计第 $k + 1$ 项；算法终止后，判断部分和 S_k 与 C 标准库函数 `exp()` 给出的 e^{-x} 相差的绝对值大小，小者更优。

具体地，算法 1 计算通项 a_k 需要以下带有溢出判断的步骤：

1. 计算 x^k：$x = 0$ 时 $x^k = 0,\ \forall k \geq 1$；$x \geq 1$ 时过大的 k 使得 x^k 上溢；
2. 计算 $k!$：k 过大时 $k!$ 上溢；
3. 计算 $x^k/k!$：分子分母相差过大时发生上溢或下溢；
4. 计算 $S_{k-1} + a_k$：$|S_{k-1}| \gg |a_k|$ 时导致假收敛现象；
5. 计算 S_k：$|x| \gg 0$ 时可能上溢。

类似地，算法 2 计算通项 a_k 需要以下带有溢出判断的步骤：

1. 计算 $a_k = -xa_{k-1}/k$：可能发生上溢或下溢；
2. 计算 $S_{k-1} + a_k$：$|S_{k-1}| \gg |a_k|$ 时导致假收敛现象；
3. 计算 S_k：$|x| \gg 0$ 时可能上溢。

以算法 2 为例，笔者编写了以下带有溢出检验的 C 代码实现：

```
/*- enx/enx.c -*/
#define number double

number enx_rec(number x)
{
    number k = 0, ak = 1, Sk = 1;

    while(1) {
        k += 1;
        ak *= -1 * x / k;
        if(ak == 0) {
            fprintf(stderr, "%s(%g): a[%g] underflow\n", __func__, x, k);
            break;
        }
        if(isinf(ak)) {
            fprintf(stderr, "%s(%g): a[%g] overflow\n", __func__, x, k);
            break;
        }
```

```
            if(Sk + ak == Sk) {
                fprintf(stderr, "%s(%g): Sk[%g] = %g convergent,"
                    " with Sk[%g] = %g\n", __func__, x, k - 1, Sk, k, ak);
                break;
            }
            Sk = Sk + ak;
            if(isinf(Sk)) {
                fprintf(stderr, "%s(%g): Sk[%g] overflow\n", __func__, x, k);
                break;
            }
        }
        return Sk;
}
```

在程序目录 enx/ 下执行 `make run` 命令，开始比较三种算法的优劣。比较结果将写入 enx/enx.out 文件中，累加终止的原因则写入 enx/enx.log 中。图 1 作出了各算法给出的 e^{-x} 结果及其标准参考值，表 I 总结了不同 x 输入下各算法停止累加时的详细情况。

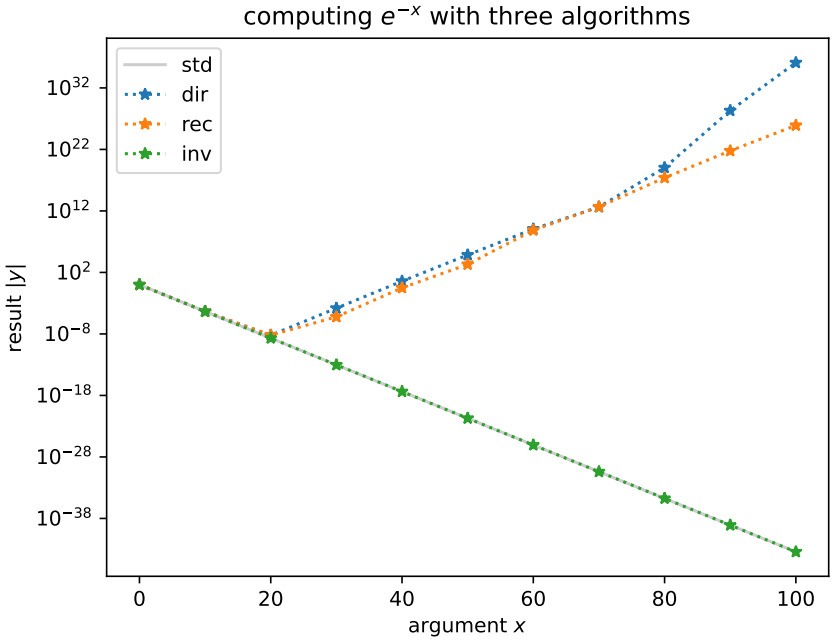

图 1: 各 e^{-x} 算法结果比较。图中纵座标取计算结果绝对值，因负值无法作图。图例中 std, dir, rec 和 inv 分别表示 C 标准库、直接法、递推法和倒数法的结果。

观察图 1，不难看出，随着 x 的增大，直接法和递推法给出的 e^{-x} 结果均严重偏离了标准值，故它们对于本问题而言都不是好的算法。表 I 给出了这一结果出现的原因：一方面，当 x 较大而 k 较小时，通项 a_k 的绝对值较大，而它们的符号是交错的，大数相减导致了有效数字位数严重受损，巨大的偏差导致 S_k 的绝对值很大，随着 k 逐渐增大，a_k 的绝对值又急剧下降，直至与 S_k 绝对值之比小于 $\varepsilon_M/2$ 而使得部分和 S_k 不再变化，出现假收敛现象；另一方面，直接法相比递推法更为糟糕的是，由于其计算通项的方式为分子分母分开计算，二

表 I: 各 e^{-x} 算法累加终点详情

算法和输入	终点 k 值	终止原因	计算结果	结果偏差
`enx_dir(0)`	1	余项为 0	1	0
`enx_dir(10)`	58	假收敛	4.53999×10^{-05}	-3.28364×10^{-13}
`enx_dir(20)`	94	假收敛	6.13826×10^{-09}	$+4.07711 \times 10^{-09}$
`enx_dir(30)`	117	假收敛	-0.00015173	-0.00015173
`enx_dir(40)`	137	假收敛	3.74218	$+3.74218$
`enx_dir(50)`	156	假收敛	69315	$+69315$
`enx_dir(60)`	171	171! 上溢	$-1.1332 \times 10^{+09}$	$-1.1332 \times 10^{+09}$
`enx_dir(70)`	168	70^{168} 上溢	$-4.38173 \times 10^{+12}$	$-4.38173 \times 10^{+12}$
`enx_dir(80)`	162	80^{162} 上溢	$-1.07628 \times 10^{+19}$	$-1.07628 \times 10^{+19}$
`enx_dir(90)`	158	90^{158} 上溢	$-2.0287 \times 10^{+28}$	$-2.0287 \times 10^{+28}$
`enx_dir(100)`	155	100^{155} 上溢	$1.27116 \times 10^{+36}$	$+1.27116 \times 10^{+36}$
`enx_rec(0)`	1	余项为 0	1	0
`enx_rec(10)`	58	假收敛	4.53999×10^{-05}	-9.20846×10^{-14}
`enx_rec(20)`	94	假收敛	6.14756×10^{-09}	$+4.08641 \times 10^{-09}$
`enx_rec(30)`	119	假收敛	6.10304×10^{-06}	$+6.10304 \times 10^{-06}$
`enx_rec(40)`	139	假收敛	0.311695	$+0.311695$
`enx_rec(50)`	160	假收敛	2041.83	$+2041.83$
`enx_rec(60)`	175	假收敛	$7.22746 \times 10^{+08}$	$+7.22746 \times 10^{+08}$
`enx_rec(70)`	194	假收敛	$4.59408 \times 10^{+12}$	$+4.59408 \times 10^{+12}$
`enx_rec(80)`	210	假收敛	$2.45082 \times 10^{+17}$	$+2.45082 \times 10^{+17}$
`enx_rec(90)`	227	假收敛	$-5.8658 \times 10^{+21}$	$-5.8658 \times 10^{+21}$
`enx_rec(100)`	243	假收敛	$8.14465 \times 10^{+25}$	$+8.14465 \times 10^{+25}$
`enx_inv(0)`	1	余项为 0	1	0
`enx_inv(10)`	45	假收敛	4.53999×10^{-05}	0
`enx_inv(20)`	67	假收敛	2.06115×10^{-09}	-8.27181×10^{-25}
`enx_inv(30)`	85	假收敛	9.35762×10^{-14}	-2.52435×10^{-29}
`enx_inv(40)`	102	假收敛	4.24835×10^{-18}	-1.54074×10^{-33}
`enx_inv(50)`	118	假收敛	1.92875×10^{-22}	-3.29138×10^{-37}
`enx_inv(60)`	134	假收敛	8.75651×10^{-27}	0
`enx_inv(70)`	149	假收敛	3.97545×10^{-31}	$+3.50325 \times 10^{-46}$
`enx_inv(80)`	163	假收敛	1.80485×10^{-35}	0
`enx_inv(90)`	178	假收敛	8.19401×10^{-40}	-6.5253×10^{-55}
`enx_inv(100)`	192	假收敛	3.72008×10^{-44}	-2.48921×10^{-59}

者均随 x 和 k 的增大急剧增大，从而发生上溢而使得算法停止（当然，即使算法无限制进行下去，也不可能得到精确的结果，因为开始的大数相减已经引入了巨大偏差）。

从图 1 中看出，倒数法与标准曲线吻合非常好；从表 I 中也看到，实验范围内，倒数法的相对偏差均在机器精度量级。倒数法的通项具有相同符号，避免了大数相减导致的精度损失。然而，这一算法仍然受到机器精度的制约，在通项绝对值与部分和绝对值之比过小的时候出现假收敛现象。如要继续提高该算法的精度，只能选用更加精确的浮点数。以下给出笔者使用 `gcc` 提供的 `__float128` 四精度浮点数类型重复实验（源代码见 enx/enx-quad.c）得出的部分结果：

```
number = __float128
inv: exp(-0) = 1 (+0)
inv: exp(-10) = 4.53999e-05 (-5.87747e-39)
inv: exp(-20) = 2.06115e-09 (-3.58732e-43)
inv: exp(-30) = 9.35762e-14 (-1.09476e-47)
inv: exp(-40) = 4.24835e-18 (-2.00457e-51)
inv: exp(-50) = 1.92875e-22 (-1.83524e-55)
inv: exp(-60) = 8.75651e-27 (+7.46762e-60)
inv: exp(-70) = 3.97545e-31 (+0)
inv: exp(-80) = 1.80485e-35 (-1.15913e-68)
inv: exp(-90) = 8.19401e-40 (-2.8299e-73)
inv: exp(-100) = 3.72008e-44 (-1.72723e-77)
```

在笔者计算机上的 `/usr/lib/gcc/x86_64-linux-gnu/11/include/quadmath.h` 库头文件中可以看到这样的定义：

```
#define FLT128_EPSILON 1.92592994438723585305597794258492732e-34Q
```

此即该 128 位浮点数的机器精度。实验结果表明，使用倒数法计算的 128 位 e^{-x} 偏差均在机器精度量级，这验证了倒数法向更高机器精度浮点数系统推广的可能性。不过值得一提的是，笔者发现本机上 `gcc` 生成的目标代码中对 128 位浮点数的操作均使用了软件模拟而非硬件指令实现，例如使用运行时库函数 `__addtf3()` 模拟加法，由于缺少硬件支持 [1]，如此使用 128 位浮点数的代码性能将大打折扣。

II. 矩阵的模和条件数

A. 行列式

由于 $A = (a_{ij})$ 是上三角矩阵，故有：

$$\det A = \prod_{i=1}^{n} a_{ii} = \prod_{i=1}^{n} 1 = 1 \neq 0 \tag{4}$$

所以 A 非奇异。

B. 逆矩阵

记 $A_k = A_{k\times k}$ 为 A 的 k 阶主子矩阵，$I_n = I_{n\times n}$。对于 A_1，显然有 $A_1 = A_1^{-1} = I_1$。对于 $k \geq 2$，可以对 A_k 按照如下方式分块：

$$A_k = \begin{pmatrix} A_{k-1} & B_{k-1} \\ 0 & 1 \end{pmatrix} \tag{5}$$

其中 B_{k-1} 为 $k-1$ 行 1 列的仅包含元素 -1 的矩阵。假设存在与 B_{k-1} 形状相同的矩阵 C_{k-1}，使得：

$$\begin{pmatrix} A_{k-1} & B_{k-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} A_{k-1}^{-1} & C_{k-1} \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} I_{k-1} & A_{k-1}C_{k-1} + B_{k-1} \\ 0 & 1 \end{pmatrix} = I_k \tag{6}$$

只需：

$$A_{k-1}C_{k-1} + B_{k-1} = 0 \;\Leftrightarrow\; C_{k-1} = A_{k-1}^{-1}(-B_{k-1}) \tag{7}$$

这证明了 C_{k-1} 的存在性。由矩阵逆的唯一性知：

$$A_k^{-1} = \begin{pmatrix} A_{k-1}^{-1} & C_{k-1} \\ 0 & 1 \end{pmatrix} \tag{8}$$

由于 $A_1^{-1} = -B_1 = I_1$，故 $C_1 = I_1$，则：

$$A_2^{-1} = \begin{pmatrix} A_1^{-1} & C_1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{9}$$

进而：

$$C_2 = A_2^{-1}(-B_2) = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \tag{10}$$

故有：

$$A_3^{-1} = \begin{pmatrix} A_2^{-1} & C_2 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 2 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{11}$$

猜想：

$$A_k^{-1}(i;j) = \begin{cases} 0, & j < i, \\ 1, & j = i, \\ 2^{j-i-1}, & j > i. \end{cases} \tag{12}$$

当 $k = 1, 2, 3$ 时，我们已经验证。假设对于任意 $k \geq 3$，结论成立，那么：

$$C_k = A_k^{-1}(-B_k) = \begin{pmatrix} \sum_{j=1}^{k} A_k^{-1}(1;j)(-B_k(j;1)) = 1 + \sum_{j=2}^{k} 2^{j-2} = 2^{k-1} \\ \sum_{j=1}^{k} A_k^{-1}(2;j)(-B_k(j;1)) = 1 + \sum_{j=3}^{k} 2^{j-3} = 2^{k-2} \\ \vdots \\ \sum_{j=1}^{k} A_k^{-1}(k;j)(-B_k(j;1)) = 1 \end{pmatrix} \tag{13}$$

由式 (8) 知，对于 $1 \le i, j \le k$，

$$A_{k+1}^{-1}(i;j) = A_k^{-1}(i;j) = \begin{cases} 0, & j < i, \\ 1, & j = i, \\ 2^{j-i-1}, & j > i. \end{cases} \tag{14}$$

对于 $1 \le j \le k < k+1 = i$，

$$A_{k+1}^{-1}(i;j) = A_{k+1}^{-1}(k;j) = 0 \tag{15}$$

对于 $1 \le k \le k < k+1 = j$，

$$A_{k+1}^{-1}(i;j) = A_{k+1}^{-1}(i;k+1) = C_k(i;1) = 2^{k-i} = 2^{(j-1)-i} = 2^{j-i-1} \tag{16}$$

对于 $i = j = k+1$，

$$A_{k+1}^{-1}(i;j) = A_{k+1}^{-1}(k+1;k+1) = 1 \tag{17}$$

故对于 $k+1$ 时，结论也成立，数学归纳法步骤完成，我们确认 A^{-1} 的矩阵元为：

$$A^{-1}(i;j) \begin{cases} 0, & j < i, \\ 1, & j = i, \\ 2^{j-i-1}, & j > i. \end{cases} \tag{18}$$

C. 无穷模

$A = 0$ 时显然成立。当 $A \neq 0$ 时，

$$\begin{aligned} \|A\|_\infty &= \max_{\|x\| \neq 0} \frac{\|Ax\|_\infty}{\|x\|_\infty} = \max_{\max_j |x_j| \neq 0} \frac{\max_i |\sum_j a_{ij} x_j|}{\max_j |x_j|} \le \max_{\max_j |x_j| \neq 0} \frac{\max_i \sum_j |a_{ij} x_j|}{\max_j |x_j|} \\ &= \max_{\max_j |x_j| \neq 0} \frac{\max_i \sum_j |a_{ij}||x_j|}{\max_j |x_j|} \le \max_{\max_j |x_j| \neq 0} \frac{\max_i \sum_j |a_{ij}| \max_k |x_k|}{\max_j |x_j|} \\ &= \max_{\max_j |x_j| \neq 0} \frac{\max_k |x_k| \max_i \sum_j |a_{ij}|}{\max_j |x_j|} = \max_{\max_j |x_j| \neq 0} \max_i \sum_j |a_{ij}| \\ &= \max_i \sum_j |a_{ij}| \end{aligned} \tag{19}$$

设 i_m 满足 $\sum_j |a_{i_m j}| = \max_i \sum_j |a_{ij}|$，取：

$$x_j^{(m)} = \begin{cases} a_{i_m j}^* / |a_{i_m j}|, & |a_{i_m j}| \neq 0, \\ 0, & |a_{i_m j}| = 0. \end{cases} \tag{20}$$

显然由于 $A \neq 0$，$\|x_j^{(m)}\|_\infty = \max_j |x_j^{(m)}| = 1 \neq 0$。我们有：

$$\begin{aligned} \|A\|_\infty &= \max_{\max_j |x_j| \neq 0} \frac{\max_i |\sum_j a_{ij} x_j|}{\max_j |x_j|} \ge \max_{\max_j |x_j| \neq 0} \frac{|\sum_j a_{i_m j} x_j|}{\max_j |x_j|} \ge \frac{|\sum_j a_{i_m j} x_j^{(m)}|}{\max_j |x_j^{(m)}|} \\ &= \frac{|\sum_{|a_{i_m j}| \neq 0} a_{i_m j} (a_{i_m j}^* / |a_{i_m j}|)|}{1} = \Big| \sum_j |a_{i_m j}| \Big| = \sum_j |a_{i_m j}| = \max_i \sum_j |a_{ij}| \end{aligned} \tag{21}$$

综上，

$$\|A\|_\infty = \max_i \sum_j |a_{ij}| \tag{22}$$

D. 欧式模

U 幺正的充要条件是：

$$U^\dagger U = I \tag{23}$$

U 的欧式模为：

$$\begin{aligned}\|U\|_2 &= \max_{\|x\|_2\neq 0}\frac{\|Ux\|_2}{\|x\|_2} = \max_{|x|\neq 0}\frac{|Ux|}{|x|} = \max_{|x|\neq 0}\frac{\sqrt{(Ux)^\dagger Ux}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{\sqrt{x^\dagger U^\dagger Ux}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{\sqrt{x^\dagger Ix}}{\sqrt{x^\dagger x}} \\ &= \max_{|x|\neq 0}\frac{\sqrt{x^\dagger x}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0} 1 = 1\end{aligned} \tag{24}$$

由于 $U^\dagger$ 也是幺正矩阵，故同样有 $\|U^\dagger\|_2 = 1$。

对于任意与 U 形状相同的矩阵 A，

$$\begin{aligned}\|UA\|_2 &= \max_{\|x\|_2\neq 0}\frac{\|UAx\|_2}{\|x\|_2} = \max_{|x|\neq 0}\frac{|UAx|}{|x|} = \max_{|x|\neq 0}\frac{\sqrt{(UAx)^\dagger UAx}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{\sqrt{x^\dagger A^\dagger U^\dagger UAx}}{\sqrt{x^\dagger x}} \\ &= \max_{|x|\neq 0}\frac{\sqrt{x^\dagger A^\dagger IAx}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{\sqrt{x^\dagger A^\dagger Ax}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{\sqrt{(Ax)^\dagger Ax}}{\sqrt{x^\dagger x}} = \max_{|x|\neq 0}\frac{|Ax|}{|x|} \\ &= \max_{|x|\neq 0}\frac{\|Ax\|_2}{\|x\|_2} = \|A\|_2\end{aligned} \tag{25}$$

由于 U 的幺正性知，$\|Ux\|_2 = \|x\|_2$，故：

$$\|AU\|_2 = \max_{\|x\|_2\neq 0}\frac{\|AUx\|_2}{\|x\|_2} = \max_{\|Ux\|_2\neq 0}\frac{\|AUx\|_2}{\|Ux\|_2} = \max_{\|Ux\|_2\neq 0}\frac{\|A(Ux)\|_2}{\|Ux\|_2} = \|A\|_2 \tag{26}$$

故：

$$K_2(A) = \|A^{-1}\|_2\|A\|_2 = \|A^{-1}U^{-1}\|_2\|UA\|_2 = \|(UA)^{-1}\|_2\|UA\|_2 = K_2(UA) \tag{27}$$

E. 条件数

由式 (18) 知，

$$\|A\|_\infty = \max_i\sum_j|A(i;j)| = \sum_j|A(1;j)| = \sum_{j=1}^n 1 = n \tag{28}$$

$$\|A^{-1}\|_\infty = \max_i\sum_j|A^{-1}(i;j)| = \sum_j|A^{-1}(1;j)| = 1 + \sum_{j=2}^n 2^{j-2} = 2^{n-1} \tag{29}$$

故：

$$K_\infty(A) = \|A^{-1}\|_\infty\|A\|_\infty = n\cdot 2^{n-1} \tag{30}$$

III. 希尔伯特矩阵

A. 引入

选择系数 $c_1, c_2, ..., c_n$ 以最小化

$$D = \int_0^1\left(\sum_{i=1}^n c_i x^{i-1} - f(x)\right)^2 \mathrm{d}x \tag{31}$$

极值条件：

$$\begin{aligned}\frac{\partial D}{\partial c_i} &= \frac{\partial}{\partial c_i}\int_0^1 \left(\sum_{j=1}^n c_j x^{j-1} - f(x)\right)^2 \mathrm{d}x = \int_0^1 \frac{\partial}{\partial c_i}\left(\sum_{j=1}^n c_j x^{j-1} - f(x)\right)^2 \mathrm{d}x \\ &= \int_0^1 2x^{i-1}\left(\sum_{j=1}^n c_j x^{j-1} - f(x)\right)\mathrm{d}x = \int_0^1 \sum_{j=1}^n 2c_j x^{i+j-2}\mathrm{d}x - \int_0^1 2x^{i-1}f(x)\mathrm{d}x \\ &= 2\left(\sum_{j=1}^n c_j \int_0^1 x^{i+j-2}\mathrm{d}x - \int_0^1 x^{i-1}f(x)\mathrm{d}x\right) \\ &= 2\left(\sum_{j=1}^n \frac{c_j}{i+j-1} - \int_0^1 x^{i-1}f(x)\mathrm{d}x\right) = 0\end{aligned} \tag{32}$$

定义 $h_{ij} = 1/(i+j-1)$，$b_i = \int_0^1 x^{i-1}f(x)\mathrm{d}x$，则极值条件可以表为：

$$\sum_{j=1}^n h_{ij}c_j = b_i \tag{33}$$

$H_n = (h_{ij})_{n\times n}$ 即为 n 阶希尔伯特矩阵。

B. 正定性和可逆性

显然，希尔伯特矩阵是实对称矩阵。取 $f(x) = 0$，

$$\begin{aligned}D &= \int_0^1 \left(\sum_{i=1}^n c_i x^{i-1}\right)^2 \mathrm{d}x = \int_0^1 \sum_{1\le i,j\le n} c_i c_j x^{i+j-2}\mathrm{d}x = \sum_{1\le i,j\le n} c_i c_j \int_0^1 x^{i+j-2}\mathrm{d}x \\ &= \sum_{1\le i,j\le n} \frac{c_i c_j}{i+j-1}\end{aligned} \tag{34}$$

另一方面，

$$D = \int_0^1 \left(\sum_{i=1}^n c_i x^{i-1}\right)^2 \mathrm{d}x \ge \int_0^1 0\mathrm{d}x = 0 \tag{35}$$

所以，

$$\sum_{1\le i,j\le n} \frac{c_i c_j}{i+j-1} \ge 0 \tag{36}$$

定义 $c = (c_1, c_2, \dots, c_n)'$，上式即为

$$c' H_n c \ge 0 \tag{37}$$

由于式 (35) 中的被积函数在区间 $[0,1]$ 上是大于等于零的连续函数，积分为零的充分必要条件是该函数在区间 $[0,1]$ 上恒定等于 0，即 $c = 0$。而式 (37) 中的 c 是任意的，故 H_n 必为正定矩阵。考虑齐次线性方程组

$$H_n c = 0 \tag{38}$$

两边左乘 c' 得

$$c' H_n c = 0 \tag{39}$$

由 H_n 的正定性得

$$c = 0 \tag{40}$$

这说明齐次线性方程组 (38) 只有零解，故 H_n 是可逆矩阵。

C. 行列式

$$\log\det H_n = \log\frac{c_n^4}{c_{2n}} = 4\log c_n - \log c_{2n} \tag{41}$$

其中 $c_n = 1!\cdot 2!\cdots(n-1)!$，且

$$\log c_n = \log\prod_{k=1}^{n-1} k! = \sum_{k=1}^{n-1}\log k! = \sum_{k=1}^{n-1}\log\prod_{i=1}^{k} i = \sum_{k=1}^{n-1}\sum_{i=1}^{k}\log i = \sum_{i=1}^{n-1}\sum_{k=i}^{n-1}\log i = \sum_{i=2}^{n-1}(n-i)\log i \tag{42}$$

对于单个 n 而言，求值 $\log c_n$ 使用上式最右边的方法计算较好。但考虑到本题涉及到的 n 是一系列从 1 开始的递增值，笔者使用了递推算法：

$$\log 1! = 0,\ \log c_1 = 0 \tag{43}$$

$$\log k! = \log(k-1)! + \log k\ (\forall k \geq 2),\ \log c_n = \log c_{n-1} + \log(n-1)!\ (\forall n \geq 2) \tag{44}$$

其 C 代码为：

```
/*- hilbert/det.c -*/
#define number double

// 计算 \log c_1 至 \log c_{2n}
void logc(int n, number lc_out[2 * n + 1])
{
  number lf = 0;  // \log k!
  number lc = 0;  // \log c_n = \log 1! + ... + \log (n - 1)!
  int i;

  lc_out[0] = 0.0 / 0.0;
  lc_out[1] = 0;
  for(i = 2; i < 2 * n; ++i)
  {
    lc += lf;
    lf += log(i);
    lc_out[i] = lc;
  }
  lc_out[i] = lc + lf;
}
```

对于求值不超过 n 的所有 $\log\det H_n$，该算法的时空复杂度均为 $\mathcal{O}(n)$。在程序目录 hilbert/ 下执行 `make det.run`，运算结果写入 hilbert/det.out 中。本机运行结果如下：

```
1       1
2       0.0833333
3       0.000462963
4       1.65344e-07
5       3.7493e-12
6       5.3673e-18
```

7	4.8358e−25
8	2.73705e−33
9	9.72023e−43
10	2.16418e−53

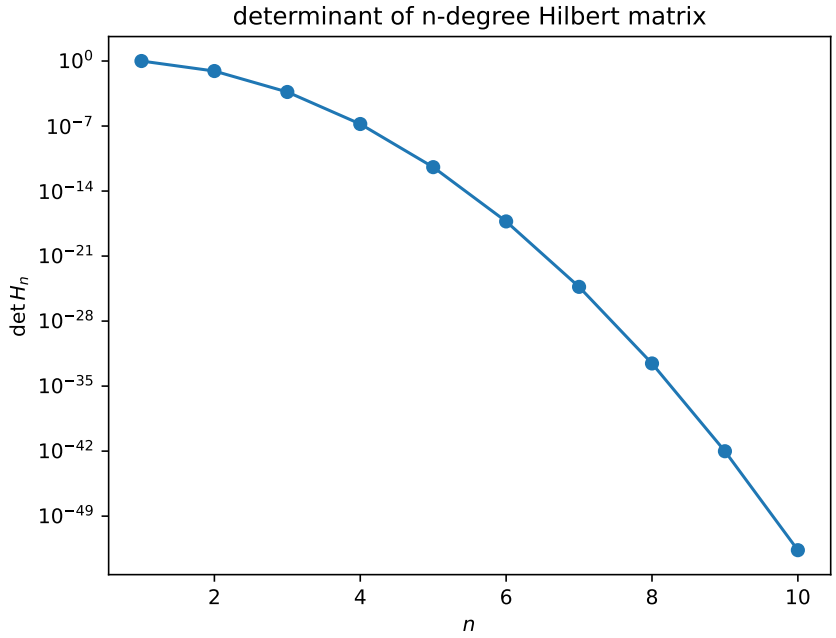

图 2: 希尔伯特矩阵行列式值随阶数变化图

图 2 绘出了 n 从 1 增至 10 时希尔伯特矩阵行列式值的变化情况。从图中明显看出，即使是 n 较小时，随 n 增大，希尔伯特矩阵仍疾速趋向于奇异。

D. 希尔伯特方程

取 $b_n = (1, 1, \ldots, 1)'$，求解希尔伯特方程 $H_n x = b_n$。笔者将高斯消元法和楚列斯基分解法分别实现为 `gem()` 和 `cdm()` 两个 C 函数，其原形为：

```
/*- hilbert/defs.h -*/
#define number double

int gem(int n, int m, number A[n][m]);
int cdm(int n, int m, number A[n][m]);
```

以上两个函数代码以及调用它们的代码必须使用支持 C99 变长数组的 C 编译器编译，一般不能使用 C++ 编译器编译（这里为了避免复杂性并充分利用编译优化，笔者不想使用 C++ 模板或封装矩阵类）。

`gem()` 为列主元高斯消元法，成功时退回 `n`，失败则退回第一个全零列对应指标，失败退回时 `A` 的元素是未定义的。`cdm()` 为楚列斯基分解法，成功时退回 `n`，失败则退回 `A` 的系数矩阵第一个非正本征值所在列对应指标，失败退回时 `A` 的元素也是未定义的。它们分别定义在源文件 hilbert/gem.c#L7 和 hilbert/cholesky.c#L36 中。此外，笔者还为它们分别编写

了单元测试代码 hilbert/gem-test.c 和 hilbert/cdm-test.c，它们编译并与被测试的目标代码链接后生成可执行文件，使用具体测试样例对被测试代码的正确性进行断言（assert）。考虑到解决本题本身也是对两种解方程代码的测试，笔者在 *-test 中仅进行了简单测试。

本题中由于无法直接求得希尔伯特方程的精确解，笔者考虑使用残差的均方值（mean squared residual error）对解的准确程度进行估计。为了看出明显趋势，笔者设定 n 从 1 递增至 200。题解程序源文件为 hilbert/hx1.c，编译链接后生成程序 hilbert/hx1。程序运行完成后求得的解写入 hilbert/hx1.out 中，mse 写入 hilbert/hx1.log 中。用户可以在 hilbert/ 程序目录下直接执行 `make hx1.run` 编译运行之。

下面给出本机运行结果 hilbert/hx1.out 的前 8 行，其中每两行对应一个 n 值，分别为高斯消元法和楚列斯基分解法求得的结果：

```
1
1
−2  6
−2  6
3  −24  30
3  −24  30
−4  60  −180  140
−4  60  −180  140
5  −120  630  −1120  630
5  −120  630  −1120  630
−6  210  −1680  5040  −6300  2772
−6  210  −1680  5040  −6300  2772
7  −336  3780  −16800  34650  −33264  12012
7  −336  3780  −16800  34650  −33264  12012
−8  504  −7560  46200  −138600  216216  −168168  51480
−8  504  −7560  46200  −138600  216216  −168168  51480
```

图 3 作出了程序运行结果 hilbert/hx1.log 中给出的 mse 值，为方便细致比较，图 4 给出了图 3 中 $1 \le n \le 13$ 的子集。这里需要说明较小的 mse 并不意味着解的精确性，因为此处的希尔伯特方程是近奇异的；然而，较大的 mse 则一定能说明解的不准确性。假设即使 x 并不十分准确，但 $H_n x$ 与 b_n 充分接近，残差的平方应当在机器精度的平方量级，那么 mse 也应当在机器精度的平方量级。然而，图 3 中 n 充分大（$n \ge 13$）时 gem 的 mse 振荡范围已在机器精度量级，远远超出机器精度的平方量级，说明此时解已有显著偏差。对于较小的 n（$n \le 13$），随着 n 的增大，gem（蓝线）和 cdm（橙线）的 mse 几乎是同步地在对数坐标轴上直线上升，这验证了即使是较小的 n，问题的病态性也随着 n 的增大迅速增强，与上一小问中的行列式求值结果相互印证。图 4 中显示，希尔伯特矩阵的病态性对 gem（蓝线）和 cdm（橙线）方法的影响几乎是相同的，它们的偏差始终接近。这说明，希尔伯特方程求解问题本身的病态性是制约问题求解精度的本质原因，而非具体的通用正定线性方程组求解方法可以改进或产生任何其它影响的。

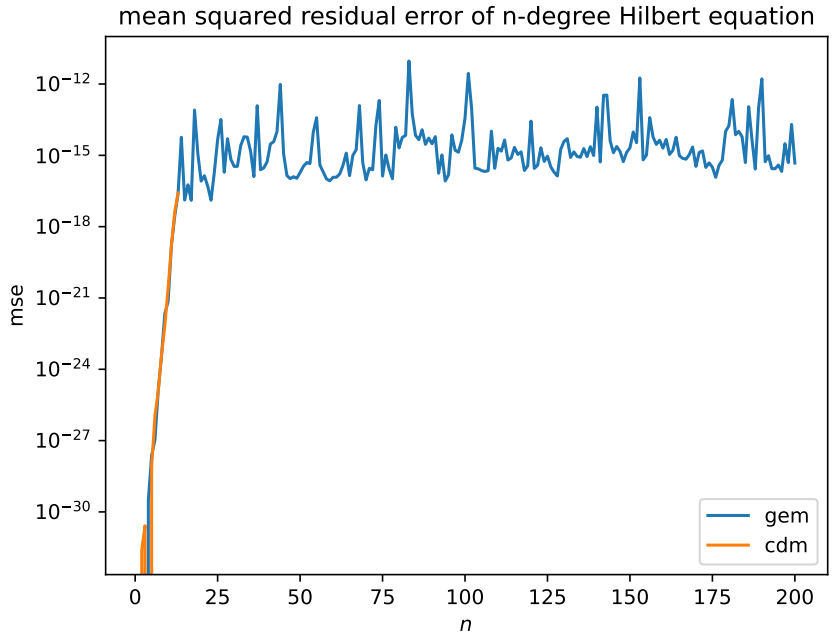

图 3: 高斯消元法和楚列斯基分解法求解希尔伯特方程精确度对比。楚列斯基分解法在 $n \geq 14$ 时出错停止，故橙线只有左半部分。

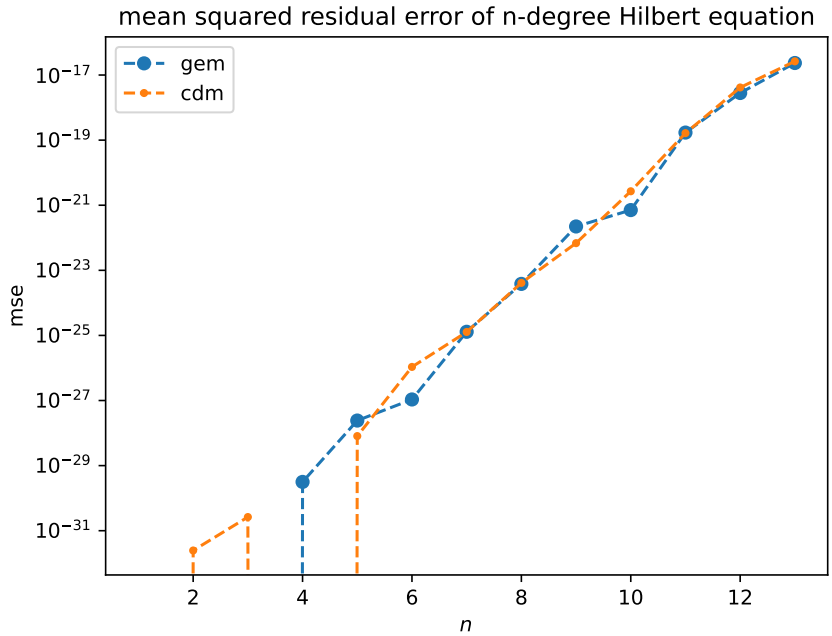

图 4: 高斯消元法和楚列斯基分解法求解希尔伯特方程精确度对比（$1 \leq n \leq 13$）。平行于纵轴的直线表示其下方对应过小的在图中无法表示的数。

IV. 级数求和与截断误差

$$
\begin{aligned}
f(q^2) &= \sum_{\boldsymbol{n}\in\mathbb{Z}^3} \frac{1}{|\boldsymbol{n}|^2-q^2} - \int \frac{1}{|\boldsymbol{x}|^2-q^2}\mathrm{d}^3\boldsymbol{x} = \sum_{i=-\infty}^{+\infty}\sum_{j=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty} \frac{1}{i^2+j^2+k^2-q^2} \\
&- \sum_{i=-\infty}^{+\infty}\sum_{j=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty} \int_i^{i+1}\mathrm{d}x_1 \int_j^{j+1}\mathrm{d}x_2 \int_k^{k+1}\mathrm{d}x_3 \frac{1}{x_1^2+x_2^2+x_3^2-q^2} \\
&= \sum_{i=-\infty}^{+\infty}\sum_{j=-\infty}^{+\infty}\sum_{k=-\infty}^{+\infty} \left(\frac{1}{i^2+j^2+k^2-q^2} - \int_i^{i+1}\mathrm{d}x_1 \int_j^{j+1}\mathrm{d}x_2 \int_k^{k+1}\mathrm{d}x_3 \frac{1}{x_1^2+x_2^2+x_3^2-q^2} \right)
\end{aligned}
\tag{45}
$$

直觉告诉我们，$f(q^2)$ 可能具有有限值。对于截断 $\Lambda>0$，定义积分：

$$
I_\Lambda(q^2) = \int_{|\boldsymbol{x}|\le\Lambda} \frac{1}{|\boldsymbol{x}|^2-q^2}\mathrm{d}^3\boldsymbol{x} \tag{46}
$$

不难得到：

$$
\begin{aligned}
I_\Lambda(q^2) &= \int_{|\boldsymbol{x}|\le\Lambda} \frac{1}{|\boldsymbol{x}|^2-q^2}\mathrm{d}^3\boldsymbol{x} \\
&= 4\pi\int_0^\Lambda \frac{x^2}{x^2-q^2}\mathrm{d}x = 4\pi\int_0^\Lambda \left(1+\frac{q^2}{x^2-q^2}\right)\mathrm{d}x \\
&= 4\pi\int_0^\Lambda \left(1+\frac{q}{2}\left(\frac{1}{x-q}-\frac{1}{x+q}\right)\right)\mathrm{d}x = 4\pi\left(x+\frac{q}{2}\log\left|\frac{x-q}{x+q}\right|\right)\Bigg|_0^\Lambda \\
&= 4\pi\left(\Lambda+\frac{q}{2}\log\left|\frac{\Lambda-q}{\Lambda+q}\right|\right)
\end{aligned}
\tag{47}
$$

注意，如果积分存在瑕点，我们只考虑主值积分，此时积分变量从左右两边趋近于瑕点时的贡献正好抵消。

定义截断于 Λ 的函数级数：

$$
f_\Lambda(q^2) = \sum_{|\boldsymbol{n}|\le\Lambda} \frac{1}{|\boldsymbol{n}|^2-q^2} - \int_{|\boldsymbol{x}|\le\Lambda} \frac{1}{|\boldsymbol{x}|^2-q^2}\mathrm{d}^3\boldsymbol{x} \tag{48}
$$

为确定合适的截断条件，考虑如下递推关系：

$$
\Delta f_l(q^2) \coloneqq f_l(q^2) - f_{l-1}(q^2) = \sum_{l-1<|\boldsymbol{n}|\le l} \frac{1}{|\boldsymbol{n}|^2-q^2} - \int_{l-1<|\boldsymbol{x}|\le l} \frac{1}{|\boldsymbol{x}|^2-q^2}\mathrm{d}^3\boldsymbol{x},\ l\ge 1 \tag{49}
$$

上式中的积分可以由式 (47) 直接得到，而求和则较难求出。考虑到精确求和计算时间复杂度较高，笔者设计了如下并行计算方案来实现：

- 定义 l 的上限 L（可以设置得非常大而我们实际达不到如此之大），并约定最多求解 $1\le l\le L$ 的各个 $\Delta f_l(q^2)$

- 依照目标计算机中央处理器的核心（线程）数分配 N 个工作线程
- 为每个工作线程分配一长度为 $L+1$ 的数组 D（可索引 0 至 L）
- 定义一全局计数器 l，起始值为 1
- 每个工作线程的主循环体原子性地（atomically，笔者使用了互斥锁实现）完成以下操作：
  - 读取 l
  - 如果 $l > L$，退出
  - 递增 l

  而后与其他线程并行（每个线程同一时刻必定拥有不同的 l 值），计算中心为坐标原点，边长为 $2l$ 的立方体表面的所有格点 (i,j,k) 对式 (49) 中求和的贡献（笔者实现时还考虑对称性带来的简并进行了优化，大大减少了运算次数），累加到数组 D 的第 $d = \left\lceil \sqrt{i^2+j^2+k^2} \right\rceil$ 个元素中或因越界而舍弃（注意如果没有越界，这一贡献是对于 $\Delta f_d(q^2)$ 而言的）
- 定义一控制计数器 l_0，从 1 开始递增至 L 或某处不大于 L 的因用户主动操作而中断时的值，控制线程确保所有工作线程均进入完成 $l = l_0$ 或已开始 $l > l_0$ 的状态抑或是已完成了全部工作的状态后，将各工作线程 D 数组的第 l_0 个元素累加后，得到式 (49) 中的求和（由于边长为 $2l_0$ 的立方体一定完整包括了其内切球，至此我们能够确保求和被完整而（如果不计舍入误差而交换律和结合律得到满足的话）精确地求解出来）

这里需要补充说明的是，考虑舍入误差，笔者的求和方案仍然是较优选择。由于边长为 $2l_0$ 的立方体表面格点到中心的距离均在 l_0 量级，按照式 (49) 定义的求和通项在 $l_0 \gg |q|$ 时大小接近，舍入误差一般不会被放大。此外，整体而言，后文中图 6 将表明，$|\Delta f_l(q^2)|$ 的值总体上是缓慢收敛的，相邻项之间大小相近，将它们累加一般也不会放大舍入误差。

笔者使用了 C 语言结合可移植的 Pthread 用户线程库实现了上述方案，控制和工作线程以信号量（semaphore）进行同步，l_0 每次递增时的 $\Delta f_{l_0}(q^2)$ 和 $f_{l_0}(q^2)$ 均实时推送至标准输出（trunc/trunc-mt.c#L218），用户认为精度已经足够或时间无法忍受而在 $l_0 < L$ 时中断程序的机制则使用了 Unix 信号（signal）实现（trunc/trunc-mt.c#L31）。对于 l_0 次计算，可以将 l_0 个独立的信号量分别在各工作线程递增 1 次和在控制进程递减 N 次以获得同步，这一机制引入的开销在 $\mathcal{O}(Nl_0)$ 量级，相比算法 $\mathcal{O}(l_0^3)$ 的复杂度完全可以忽略不计。由于工作线程除等待全局计数器锁外，没有其它等待时间，而全局计数器锁仅在读取和更新全局计数器时被线程拥有，故工作线程几乎是始终忙碌的。我们已将 N 设定为硬件线程数，故理想情况下 CPU 应当也是饱和的。此外，普遍意义下，特别是对于 Linux 的 CFS 调度策略而言，l_0 较大时由于相邻的几个 l 之间用时的相对差异非常小（$\sim l_0^{-1}$），各工作线程将趋于拥有相近的 l 值，那么控制线程也将在最短的时间内得到 $\Delta f_{l_0}(q^2)$ 和 $f_{l_0}(q^2)$ 的计算结果从而能够在第一时间反馈给用户。由此可见，对于本方案所使用的立方体参数扫描空间而言，该交互式并行方案的硬件利用和响应时间均达到了最优。

笔者认为在此有必要以具体例子说明笔者设计的同步机制。以下为源文件 trunc/trunc-mt.c#L300 中控制线程读取并行计算 $\Delta S_l(q^2)$（即 $\Delta f_l(q^2)$ 的求和部分，除去了积分部分）的代码：

```
/*- trunc/trunc-mt.c -*/

number dS_mt(ctl_p ctlp, int l)
{
  int i;
  number d = 0;

  for(i = 0; i < NTHREAD; ++i)
    while(sem_wait(&ctlp->sem[l]))
      if(errno != EINTR)
        err(EXIT_FAILURE, "sem_wait");
  for(i = 0; i < NTHREAD; ++i)
    d += ctlp->job[i].df[l];
  return d;
}
```

函数 `dS_mt()` 的 `mt` 后缀为 multithreading 的缩写，用以与其底层的顺序（sequential）版本 `dS()`（trunc/trunc-mt.c#L236）区别开。大小为 $L+1$ 的 `number` 数组 `ctlp->job[i].df` 即为前文所描述的第 i 个工作线程的数组 D。对于每个工作线程 i 而言，其在完成了对其 D 数组的第 l 个元素的计算后，都会调用 `sem_post()` 函数将信号量 `ctlp->sem[l]` 增加一（trunc/trunc-mt.c#L114 和 trunc/trunc-mt.c#L119）。控制线程在 `dS_mt()` 函数中通过调用 `sem_wait()` 共 `NTHREAD`（即 N）次确保每个线程均完成所需的计算工作后，将所有工作线程的求值结果累加，得到最终结果。

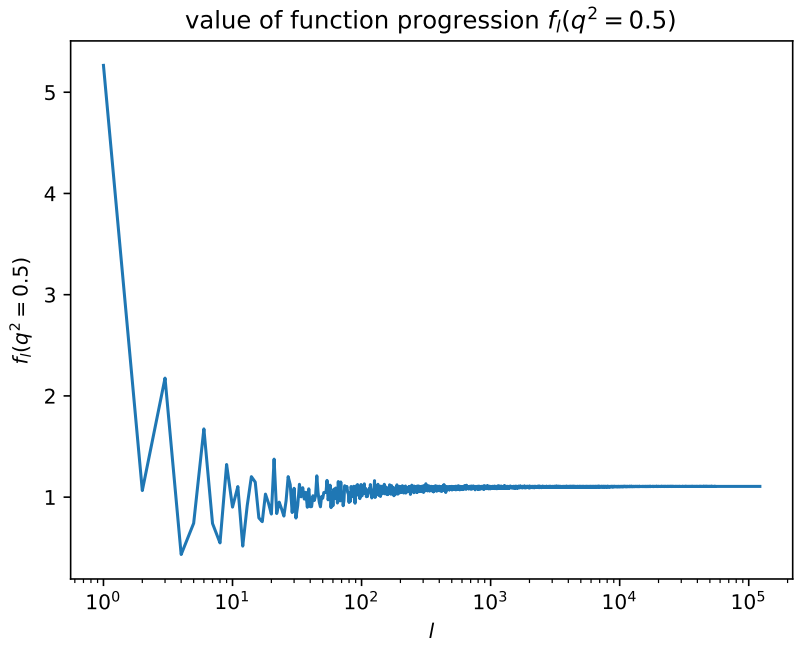

图 5: 函数级数部分和 $f_l(q^2 = 0.5)$ 随指标 l 的变化

笔者在一台多线程服务器上使用 64 个工作线程运行了程序十数小时后使用 `SIGTERM` 信号终止了程序。程序共花费长达数十天之久的 CPU 时间得到 l 从 1 至 122000 的 $\Delta f_l(q^2 =$

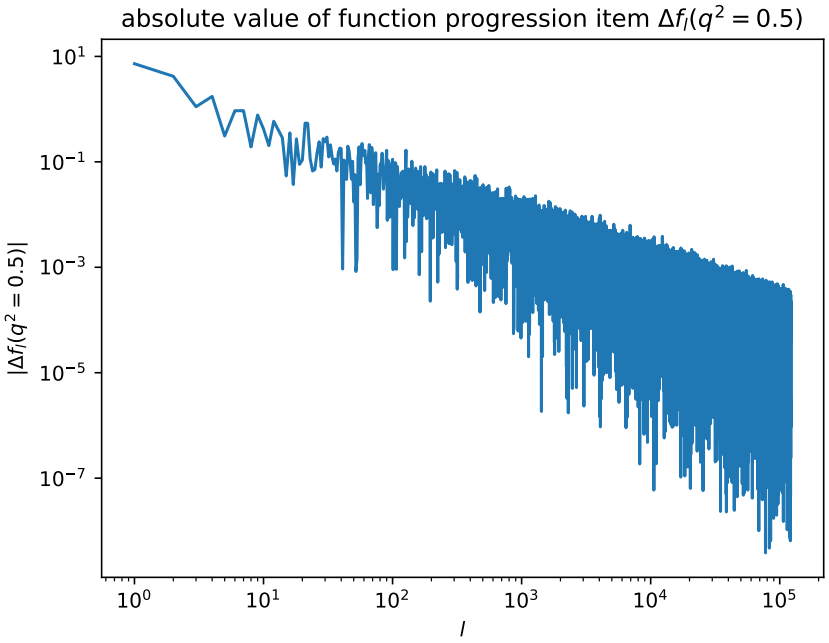

图 6: 函数级数通项绝对值 $|\Delta f_l(q^2 = 0.5)|$ 随指标 l 的变化

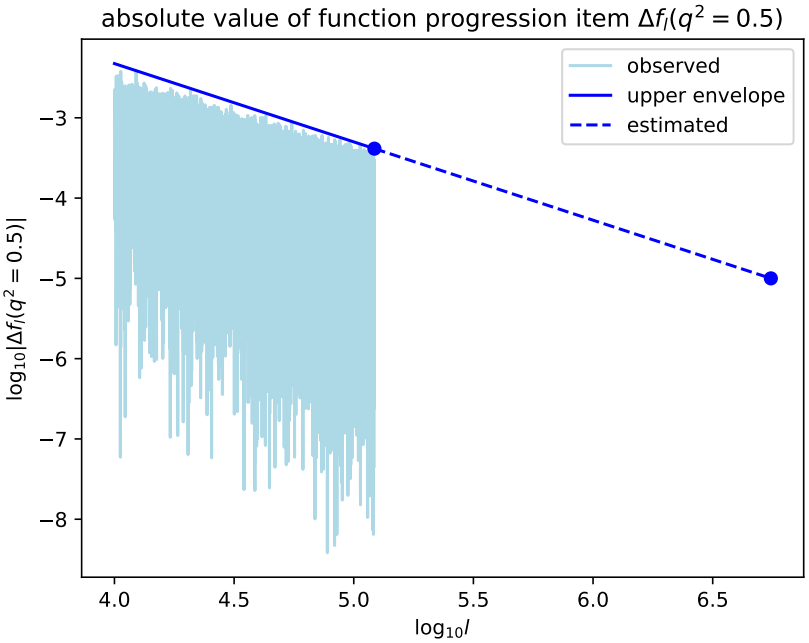

图 7: 函数级数通项绝对值 $|\Delta f_l(q^2 = 0.5)|$ 随指标 l 的变化趋势的拟合，截断处计算精度的估计和 10^{-5} 精度所需计算量的估计。

0.5) 和 $f_l(q^2 = 0.5)$ 的值。其中部分和结果如图 5，级数通项绝对值作于图 6。图 7 选取了图 6 中 $l \geq 10^4$ 的数据进行分析。注意到，图 7 中实测数据关于 l 的上界可以使用直线很好地描述，笔者使用了以下算法确定这一直线的方程：

- 首先使用最小二乘法对所有数据点进行拟合
- 舍弃所有直线下方的点
- 不断重复上述过程，直到只剩下最后两个数据点

此时两点最小二乘法确定的直线即为数据集关于 l 的上界。图 7 中的蓝色实线显示了这一上界。这条直线与数据区域参数 l 上界的交点纵坐标可以用于估计截断于 l 上界处的计算精度（如图 7 中的左上蓝点）。将其延长至纵坐标 -5 处，可以估计计算精度 10^{-5} 所需的计算量（如图 7 中的右下蓝点）。

图 7 中的两个蓝点坐标进行指数换算后分别为 $(122000, 0.0004)$ 和 $(5534703, 0.00001)$，由此得到我们截断于第 122000 项的精度大概为 0.0004，而要达到精度 10^{-5} 需要在大约 $\Lambda = 5534703$ 处截断。从程序输出中读出，$f_{122000}(q^2 = 0.5) = 1.1061$，由此得到求值结果：

$$f(q^2 = 0.5) = 1.1061 \pm 0.0004 \tag{50}$$

以及我们期待的 10^{-5} 精度所需的截断点的预测值：

$$\Lambda = 5534703 \tag{51}$$

此外，笔者猜想，使用式 (45) 中最后一个等号给出的方法进行计算，求和与积分的空间均选择为立方体，也许能以更小的计算量获得相当的精度。

参考文献

[1] https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html

A 项目源代码

参见 https://github.com/lyazj/numphy-01。