

# 计算物理学作业一

许嘉琪 物理学院 1500011417

October 12, 2017

## 1 数值误差的避免

### (a)

由于在机器中浮点数运算是不满足结合律的，因此对于公式中给出的求和 $\sum_{i=1}^N x_i$ 采用不同顺序进行求和，所得出的结果的误差大小也不同。我们只需要找到给定样本 $x_1, x_2, \dots, x_N$ 下，使得舍入误差取到最大值的求和顺序即可。

记所有元素之和为 $S$ ，考虑：

$$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \dots x_N = S \cdot [1 + \sum_{i=2}^{N-1} \frac{\sum_{j=1}^i x_j}{S} \epsilon_i]$$

式中用 $\epsilon_i$ 来表示第 $i$ 次加法所带来的相对舍入误差大小。在最近舍入下，它们应该不大于机器误差的一半： $\epsilon_i \leq \frac{\epsilon_M}{2}$ 。观察这个式子，可以发现最先被求和的数对误差的贡献系数越多，因此采用从最大数开始计算，按数值大小递减的顺序进行加法可以使得舍入误差取到最大值。为了更加方便的给出误差上界，我们把式子中的所有 $x_j$ 均取为样本中的最大值： $x_m$ 。这样：

$$S \cdot [1 + \sum_{i=2}^{N-1} \frac{\sum_{j=1}^i x_j}{S} \epsilon_i] \leq S \cdot [1 + \sum_{i=2}^{N-1} \frac{i x_m}{S} \epsilon_i] = S [1 + \frac{(N-2)(N+1)x_m \epsilon_M}{4S}]$$

最后计算平均值 $\bar{x} = \frac{S}{N} = \bar{x} [1 + \frac{(N-2)(N+1)x_m \epsilon_M}{4S}]$  因此， $\bar{x}$ 的舍入误差的最大可能的上限为

$$\frac{(N-2)(N+1)}{2N} \cdot \frac{\epsilon_M}{2} \approx (N/2) \frac{\epsilon_M}{2}$$

(b)

考虑到抵消现象，第一个公式是更为准确的。当N较大的时候，两个公式的主要计算部分在于求和。对于第一个公式，求和项为 $\sum_{i=1}^N x_i^2$ ；对于第二个公式，求和项为 $\sum_{i=1}^N (x_i - \bar{x})^2$ 。当所求数据集的方差较小的时候，第二个求和公式的每一项都是两个大数相减得到的小数的平方，这会导致减法结果的有效数字减少，虽然没有发生舍入，但误差仍然变大。相比而言，第一个公式的平方项求和就不会出现抵消现象。

(c)

证明：

1. 首先，当 $n = 0$ 时 $I_0 = \int_0^1 dx \frac{1}{x+5} = \ln(x+5) \Big|_0^1 = \ln 6 - \ln 5$
2. 再考虑递推： $I_n = \int_0^1 dx \left( \frac{x^n}{x+5} \right) = \int_0^1 dx \left( \frac{x^{(n-1)}(x+5-5)}{x+5} \right) = -5 \int_0^1 dx \frac{x^{n-1}}{x+5} + \int_0^1 x^{n-1} dx = -5I_{n-1} + \frac{1}{n}$
3. 如果 $I_0$ 有一个微小的误差 $e_0 = \epsilon$ ，则根据递推式： $e_k = 5e_{k-1}$ 即，误差将在每次递推计算后放大5倍。这一点可以用实际测试来得到验证（见Figure 1）：从结果中可以看到：当 $I_0$ 只出现了不到1%的偏差，误差将在递推的过程中传递放大。当进行第9次递推计算后，误差已经达到了 $0.001 * 5^9$ 的量级。结论得到了很好的验证。

```
>>> import math
>>> I0 = math.log(6/5,math.e)
>>> I0
0.1823215567939546
>>> def recursion(i0,n):
    k=1
    for k in range(1,n):
        i0=1/k-5*i0
        print(i0)

>>> recursion(I0,10)
0.08839221603022707
0.05803891984886467
0.04313873408900998
0.03430632955495011
0.02846835222524946
0.024324905540419356
0.02123261515504607
0.018836924224769652
0.016926489987262844
>>> recursion(I0-0.001,10)
0.09339221603022707
0.033038919848864645
0.1681387340890101
-0.5906936704450505
3.1534683522252527
-15.600675094459598
78.14623261515513
-390.6061630757756
1953.1419264899891
```

Figure 1:python code

## 2 矩阵的模与条件数

### (a) 计算矩阵A的行列式，说明A的确不是奇异矩阵

由于A是上三角矩阵，A的行列式即为A所有对角元元素之积： $\det(A) = 1 \neq 0$

### (b) 给出矩阵的逆矩阵 $A^{-1}$ 的形式

考虑A中某元素 $A_{ij}$ 的代数余子式：

1. 如果 $i = j$ ，余子式为 $A^{(n-1) \times (n-1)}$ 的行列式，等于1。
2. 如果 $i < j$ ，即对角线右上方的元素，它们余子式均为零。原因是当第i行被删除之后，元素 $A_{(i+1)i} = 0$ 会移动到对角线的位置，而删除第j(j*i)*列的操作仍然保持矩阵为上三角矩阵。因此，代数余子式为零。这就说明 $A^{-1}$ 同样也为一个上三角矩阵。
3. 如果 $i > j$ ，即对角线左下方的元素，它们的余子式不为零，我们通过逆矩阵的直接定义来求出它们的值： $AA^{-1} = I$ 考虑 $A^{-1}$ 的第j列元素，由于这一列数组和A本身的第i行( $i, j$ )数组进行点乘要等于零，再考虑到A第i行元素的特点，我们可以推断出 $A_{jj}^{-1} = 1, A_{(j-1)j}^{-1} = 1$ 同时有： $A_{kj}^{-1} = \sum_{i=k+1}^j A_{ij}^{-1}$ ，即：每个右上角的元素等于它下方所有元素之和(又因为其实两个值均为1，则每个元素等于下方元素的2倍)。因此，我们可以得出 $A^{-1}$ ，比如 $n=5$ 时： $A^{-1} =$

$$\begin{pmatrix} 1 & 1 & 2 & 4 & 8 \\ 0 & 1 & 1 & 2 & 4 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

### (C)

在实数域上考虑此问题。观察到 $\|A\|_p$ 的定义是与任取的向量x模长之比，我们不妨取在p模定义下的单位向量作为不同的x，直接寻找 $\|Ax\|_p$ 的上确界即可。考虑到无穷模的形式： $\|x\|_\infty = (\sum_{i=1}^n |x_i|^\infty)^{\frac{1}{\infty}} = \max |x_i|$ 。这样，只要x中的元素的绝对值都不大于1，就有 $\|x\| = 1$ 。假设列向量 $b = Ax$ ，那么在 $\|x\| = 1$ 的约束下，A的无穷模就等于 $\max b_i$ 。又由于x中的元素的绝对值都不大于1，那么b中元素的最大值为 $\max \sum_{j=1}^n |a_{ij}|$ 。证明完毕。

(d)

证明：首先，根据定义，么正矩阵满足  $UU^\dagger = I$ 。若考虑欧式模可以看出：  $\|x\|_2^2 = x^\dagger x$ 。因此有：

$$\begin{aligned}\|Ux\|^2 &= (Ux)^\dagger Ux \\ &= x^\dagger U^\dagger Ux \\ &= x^\dagger Ix = \|x\|^2\end{aligned}$$

同理  $\|U^\dagger x\| = \|x\|$ 。因此证明了：

$$\|U\| = \frac{\|Ux\|}{\|x\|} = 1 = \|U^\dagger\|$$

再证明：对于任意复数域上的矩阵A：

$$\|UA\| = \sup \frac{\|UAx\|}{\|x\|} = \sup \frac{\|Ax\|}{\|x\|} = \|A\|$$

因此  $K_2(UA) = \|A\| \cdot \|A^{-1}\| = K_2(A)$

(e)

考虑(a)中的矩阵：A的无穷模为：  $\|A\|_\infty = \max \sum_{j=1}^n |a_{ij}| = 1$ ； $A^{-1}$ 的无穷模为：  $\|A^{-1}\|_\infty = \max \sum_{j=1}^n |a_{ij}^{-1}| = 2^{n-1}$ 。因此，矩阵的条件数为：

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

### 3 Hilbert 矩阵

(a)

为了使得D取极小值，我们首先求D对某一个  $c_j$  的偏导数：

$$\frac{\partial D}{\partial c_j} = \int_0^1 dx [2(\sum_{i=1}^n c_i x^{i-1} - f(x) \cdot x^{j-1})]$$

令其等于零：

$$0 = \int_0^1 dx [\sum_{i=1}^n c_i x^{i+j-2} - f(x)x^{j-1}]$$

即：

$$\sum_{i=1}^n \frac{x^{i+j-1}}{i+j-1} = \int_0^1 dx f(x)x^{j-1}$$

与题目中提供的形式做对比:  $\sum_{i=1}^n H_{ij}c_j = b_j$  可以得到:

$$(H_n)_{ij} = \frac{1}{i+j-1}$$

$$b_i = \int_0^1 f(x)x^{i-1}dx$$

**(b)**

为了证明  $c^T H_n c \geq 0$  我们回到 Hilbert 矩阵的定义:

$$H_n \cdot c = b$$

这样我们可以根据在(a)问中已求出的b的形式表达:

$$\begin{aligned} c^T H_n c &= \sum_i c_i \int_0^1 f(x)x^{i-1}dx \\ &= \int_0^1 \sum_i c_i f(x)x^{i-1}dx \\ &= \int_0^1 f(x) \cdot \left(\sum_i c_i x_{i-1}\right)dx \end{aligned}$$

对于任意的  $c_i (i=1,2,\dots,n)$ , 我们总可以取  $f(x) = \sum_{i=1}^n c_i x^{i-1}$ 。这样, 上式就化为:

$$c^T H_n c = \int_0^1 f^2(x)dx \geq 0$$

仅当  $f(x) = 0$  即  $c_i = 0 (i=1,2,\dots,n)$  时, 等号成立。

根据线性代数的知识, 半正定的实对称矩阵一定可以对角化, 并且每一个特征值都大于零。这就足以说明 Hilbert 矩阵是非奇异的。

**(c)**

对题目中已经给出的  $\det(H_n)$  表达式求 log, 不难算出:

$$\log(H_n) = \sum_{i=1}^{n-1} (2n-3i)\log(i) - \sum_{i=n}^{2n-1} (2n-i)\log(i)$$

再一次, 我们运用 python 进行计算, 所得的结果见 Table 1:

Table 1: 取对数估算 $\det(H_n)$

n	H(n)
1	1.000000000E+00
2	8.333333333E-02
3	4.629629630E-04
4	1.653439153E-07
5	3.749295133E-12
6	5.367299887E-18
7	4.835802624E-25
8	2.737050114E-33
9	9.720234312E-43
10	2.164179226E-53

```
import math
def H(n):
    logH=0
    for i in range(1, n):
        logH = logH + (2*n - 3*i)*math.log(i, math.e)
    for i in range(n, 2*n):
        logH = logH - (2*n - i)*math.log(i, math.e)
    return math.exp(logH)
for i in range(1,11):
    print(H(i))
```

Figure 2:python code

(d)

利用c++进行编程，我们将在三种不同设定下，分析得到的结果见table 2、3:

如Table 2，而当 $n \geq 7$ 之后，两个算法已经都不能给出正确的结果（Cholesky的偏离更大一些）。取float类型变量使得机器精度较低，在这种情况下，两种算法的表现较为类似。

从Table 3中结果可以看到，在这种设定下，Cholesky分解在 $n=4$ 的时候开始出现偏移；而GEM方法直到 $n=8$ 都可以给出精确解。可以说，加入支点遴选的GEM是更精确的算法。可能的原因:

1. Cholesky分解法需要的运算次数明显多于GEM分解，由于计算分解矩阵H时带来误差。
2. 由于在C++语言中，我给矩阵元使用了long double类型，机器精度相当高，这样GEM方法获得优势。
3. 而考虑到Cholesky算法中需要多次开平方运算，其运算精度较低，产生了很大的误差。

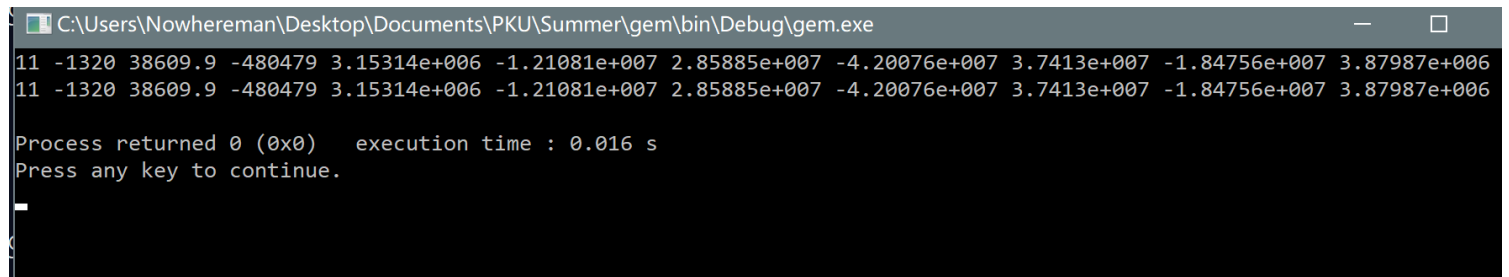
Table 2: Table 2: 数据精度为float时的结果						
n=1	x1	x2	x3	x4	x5	x6
GEM	1					
CHO	1					
2						
GEM	-2	6				
CHO	-2	6				
3						
GEM	2.99999	-23.9999	29.9999			
CHO	2.99999	-23.9999	29.9999			
4						
GEM	-4.00003	60.00005	-180.002	140.001		
CHO	-3.99992	59.9993	-179.999	139.999		
5						
GEM	4.99808	-119.963	629.832	-1119.74	629.869	
CHO	4.99651	-119.935	629.72	-1119.58	629.92	
6						
GEM	-5.83079	205.338	-1649.18	4961.18	-6214.1	2738.49
CHO	5.81654	204.95	-1646.61	4954.49	-6206.7	2735.56

Table 3: Table 3: 数据类型使用long double,开根计算使用float精度的结果

n=1	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10
GEM	1									
CHO	1									
2										
GEM	-2	6								
CHO	-2	6								
3										
GEM	3	-24	30							
CHO	3	-24	30							
4										
GEM	-4	60	-180	140						
CHO	-3.99998	59.9998	-180	140						
5										
GEM	5	-120	630	-1120	630					
CHO	4.99994	-119.999	629.996	-1119.99	629.998					
6										
GEM	-6	210	-1680	5040	-6300	2772				
CHO	-6.00005	210.003	-1680.02	5040.06	-6300.07	2772.3				
7										
GEM	7	-336	3780	-16800	34650	-33264	12012			
CHO	7.00273	-336.109	3781.03	-16804	34657.2	-33270.1	12014			
8										
GEM	-8	504	-7560	46200	-138600	216216	-168168	51480		
CHO	-8.02289	505.15	-7574.31	46274.7	-138795	216486	-168356	51532.3		
9										
GEM	9.14037	-728.952	14003.3	-111862	453938	-1015950	1269070	-828321	219920	
CHO	9	-720	13860	-110800	450450	-1009010	1261260	-823680	218790	
10										
GEM	-10	990	-23760	240240	-1261260	3783780	-6726720	7001280	-3938220	923780
CHO	-11.0391	1074.71	-25493.2	255539	-1332600	3976400	-7038170	7298620	-4092730	957461



第三中设定是数据类型使用long double，并且两个算法的运算精度也都是long double精度。为了简明起见在这里只给出n=11时的程序运行结果：



```
C:\Users\Nowhereman\Desktop\Documents\PKU\Summer\gem\bin\Debug\gem.exe
11 -1320 38609.9 -480479 3.15314e+006 -1.21081e+007 2.85885e+007 -4.20076e+007 3.7413e+007 -1.84756e+007 3.87987e+006
11 -1320 38609.9 -480479 3.15314e+006 -1.21081e+007 2.85885e+007 -4.20076e+007 3.7413e+007 -1.84756e+007 3.87987e+006

Process returned 0 (0x0)   execution time : 0.016 s
Press any key to continue.
```

Figure 3:c++ prompt

可以看出，两种方法给出的结果是基本相同的，这说明在机器精度较高的情况下，两种方法的稳定性都不错。但是，在 $x_3$ 的精确解应该为：38610，但是在这里的结果是38609.9，说明任然存在误差。解决的办法需要重新定义分数类，获得分数间的无损乘除法。