Stirling 公式的数值验证尝试

李洋

导言:

Stirling 公式作为一则近似公式,在统计物理中,计算大数排列组合的近似值时,发挥了重要作用.关于此公式,相关的证明也已完善(见附录三).然而,在一定的数值条件下,Stirling 公式究竟具有何种程度的近似,却是该证明没有涉及的.在物理研究中,能清楚的知道推导过程中,用到的究竟是何种程度的近似,是非常必要的.因此本文将介绍一种利用计算机,应用 C 语言, 对 Stirling 公式进行数值验证的方法(或称之为尝试).以期解决上述问题.

Stirling 公式的数值验证尝试

一、大数的表示和存储

Stirling 公式是统计物理中用到的一则关于 N!的近似公式,其具体形式如下:

$$N! \approx \quad N^N \cdot e^{-N} \cdot \sqrt{2\pi N} \qquad (1)$$

不难看出, Stirling 公式左右两侧将随正整数 N 的增加迅速增大.事实上, 这也正是应用编程完成对其数值检验的难点所在. C 语言中允许变量赋值范围最大 的数据类型为双精度(double)类型 , 其数值范围约为 1.7E-308~1.7E+308.

经计算:

$170! = 7.2574E+306$ (1)

$143^{143} = 1.6333E+308$ (2)

$e^{709} = 8.2184E+307$ (3)

可以看到如果应用普通的数值计算, 由于变量赋值范围的限制, Stirling 公式的验证到 140 左右就不得不停止. 这种限制也广泛存在于 Matlab 等这种成品数值计算程序中.

如果希望继续运算下去, 就不得不提出不同于 *将整个数值存入一个变量* 的计算方法.

引入高精度算法就是解决这一问题的方法之一.

高精度算法的核心思想, 是将大数拆开, 分散存储在多个变量(或称之为数组)中.

a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
3	4	5	4	8	4	6	8	3	1
10^9	10^8	10^7	10^6	10^5	10^4	10^3	10^2	10^1	10^0

具体操作就是, 定义整型一维数组 a[N],并规定数组中不同元素储存一个大数的不同位. 上图就表示了 3,454,846,831 这一数值. 由于数组在内存中的存储地址是连续的, 因此应用指针可以很方便的编写数值读取函数. (但这实际上也是一种相当浪费内存的设法, 这将在以后内容中提到)这样一来, 只需要定义足够大的数组, 就可以获得相对应位数的数. 比如设定 1,000,000 个元素的一维数组, 就可以得到 0 到 9.999^1000,000 大的数, 这远大于双精度类型的变量赋值的范围.

二、运算规则的定义

在定义了数字的表示和储存方法后, 就不得不考虑运算规则的问题. 由于将大数拆分成了许多单独的位, 编译环境自带的加减乘除开方成方运算, 对于整个数组来说就失效了. 而在 Stirling 公式的计算中需要用到以下运算法则(按实现的困难程度升序列举):

·**高精度取位法则**(确定数组中存储的究竟是多少位的数)

·**高精度加法法则**

·**高精度减法法则**

·**高精度数乘法则**(一个小的数乘以整个数组)

·高精度整数幂运算法则
·高精度阶乘法则
·高精度除法法则
·高精度自然底数幂运算
(以上运算法则的算法大多为自行编写，或许不足以作为范例，只做参考和交流)

本文将只详细介绍**高精度除法法则**和**高精度自然底数幂运算**的实现方法，其他的运算规则比较容易理解，具体可参考附录 1 中的源码.

① 高精度除法法则

高精度除法法则的实现实际上是借助于高精度减法.

现计算 a[10]/b[10]:
其中 a[10]~4846931; b[10]~12;

a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
0	**0**	**0**	**4**	**9**	**4**	**6**	**8**	**3**	**1**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**2**
b[9]	**b[8]**	**b[7]**	**b[6]**	**b[5]**	**b[4]**	**b[3]**	**b[2]**	**b[1]**	**b[0]**

Step1：利用**高精度取位法则**计算出两数组存储的数字的位数 pota 和 potb 以及他们的差值 dig.

Step2：利用指针将 b[]的最高位移动到 a[]的最高位.

a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
0	0	0	4	9	4	6	8	3	1
0	0	0	1	2					
			b[1]	b[0]					

Step3：做减法(注意借位运算)直至 a[6]为负，每做一次减法 c[dig]中加一.
(数组 c[]存放的是除法的结果)

Step4:回撤一次减法运算(因为以最高位为负值作为停止标准，多减了一次)

Step5:指针移向下一位，并将 a[6]残余的数值乘 10 加到 a[5]中

a[9]	a[8]	a[7]	a[6]	a[5]	a[4]	a[3]	a[2]	a[1]	a[0]
0	0	0	0	1	4	6	8	3	1
0	0	0	0	1	2				
				b[1]	b[0]				

c[9]	c[8]	c[7]	c[6]	c[5]	c[4]	c[3]	c[2]	c[1]	c[0]
0	0	0	0	4	0	0	0	0	0

Step6:重复上述操作

这里定义的减法运算显然只能精确到个位，如果加入小数点，运算规则将更加复杂.

② 高精度自然对数幂运算

(I)一个不太成功的设想

e^N 的计算，并不像整数计算那样简单，因为其中包含了一个无理数 e. 一种可能的方法是用一个有限位小数(如 2.7183)代替 e. 但仔细考量后就会发现这个方法并不可行. 如前文提到：

$e^{709} = 8.2184E+307$

假设前 708 个 e 都不近似，只是最后一个 e 近似为 2.7183，最终结果的误差将会高达 1.6E+303.有理由相信，如果 709 个 e 全部近似为 2.7183，这一误差将会大到不可忽略的地步. (实际经过计算为：8.2575e+307， 两数相差 3.9E+305)这将严重影响最终公式右侧数值的准确度.

或许，当对 e 取足够多位数(如取到小数点后 100 位)时，结果将最终达到精度的要求，但时如前文所说，如果引入小数点，运算规则将更加复杂. 如此定义算法，得不偿失.

(II)一个看似可行的方法

提到 e^x,很自然可以想到这样一个等式：

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots \tag{2}$$

e^x 的 Taylor 展开式(或更确切的说时 Maclaurin 展开式)提供了一种计算 e^N 的思路. 因为 e^N(N 是一个很大的值)是一个有限的数，因此(2)式中右侧级数一定是收敛的. 也就是其中一项的值满足

$$\lim_{n\to\infty} a_n = \lim_{n\to\infty} \frac{N^n}{n!} = 0 \tag{3}$$

对于一个任意大于 0 的小量ε，当 n 增大到一定程度时, a_n 必将收敛至小于ε. 不妨取ε=1，假设 n ≥ M 时 $a_n < 1$，即：

$$a_M = \frac{N^M}{M!} \le 1 \tag{4}$$

则 e^N 可写作：

$$e^N = 1 + N + \frac{N^2}{2!} + \frac{N^3}{3!} + \cdots + \frac{N^M}{M!} + \Delta(M) \tag{5}$$

要求解这个问题实际上需要一些近似：

(i)第一个近似，忽略Δ(M).

下面研究Δ(M)对整个和的贡献：

$$\Delta(M) = \sum_{j=1}^{\infty} a_{M+j}, \qquad a_{M+j} = a_M \cdot \frac{N}{M+1} \cdot \frac{N}{M+2} \cdot \ldots\ldots \cdot \frac{N}{M+j}; \tag{6}$$

首先考虑 a_{2N-1} 这一项：

$$\begin{aligned} \boldsymbol{a_{2N-1}} &= \frac{N}{1} \cdot \frac{N}{2} \cdot \ldots \cdot \frac{N}{N} \cdot \ldots \frac{N}{2N-1} \\ &= \prod_{\beta=1}^{N-1} \frac{N}{N+\beta} \cdot \frac{N}{N-\beta} \\ &= \prod_{\beta=1}^{N-1} \frac{N^2}{N^2-\beta^2} > 1 \end{aligned}$$

可见, a_{2N-1} 这一项并未收敛至小于 1，换言之：M ≥ 2N, N/M ≤ 1/2; 将这一关系应用到(6)式再进行适当的放缩，就可以得到：Δ(M) ≤ $2a_M$ ≤ 2.

上述关系对于任意 N 是普适成立的，由于 N 一般很大，因此一个小于 2 的数对整个和式结果的影响式极其微弱的Δ(M)可以舍去.

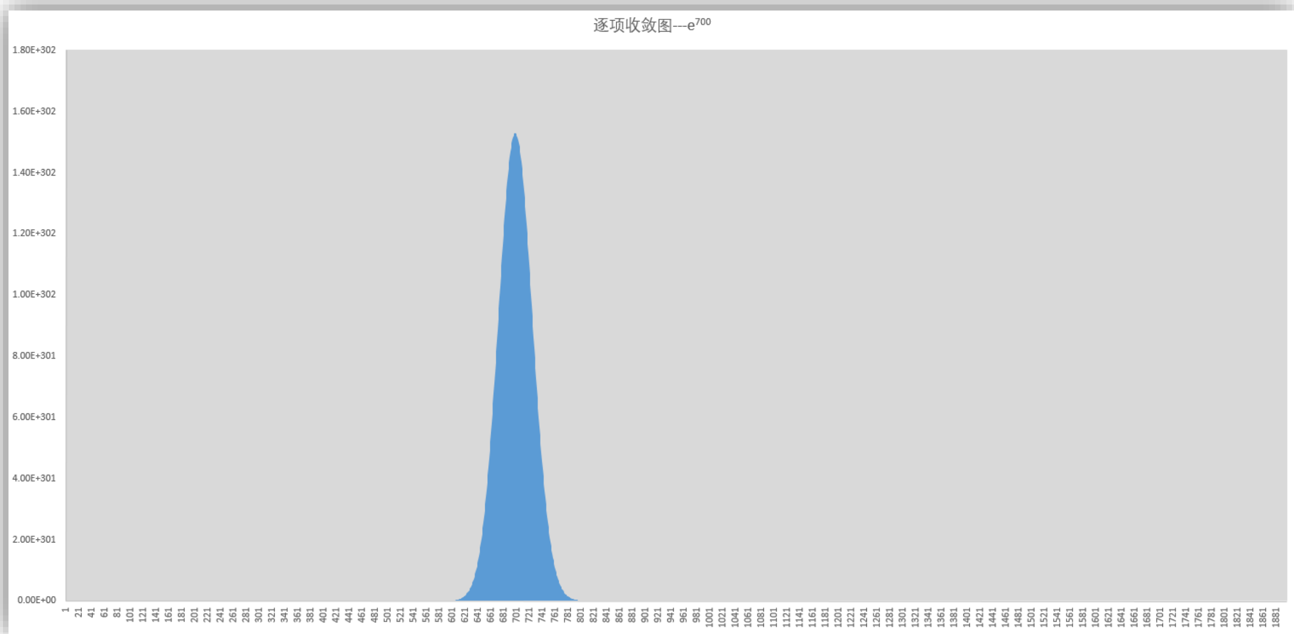
(ii)第二个近似，每项只取整数部分.

通过前文的描述不难发现，e^N 求解的实现，**实际上是要基于高精度除法运算的**. 前文已经提到，为了使算法不至过于复杂，源码中的高精度除法运算只保留到个位. 再将这种除法定义应用到 e^N 的计算中后，就可能产

生一个问题. 由于定义的 e^N 实际上是大量除法结果加和的结果，如果每一项都将小数位忽略，在进行了 M 次加和之后，在除法中忽略的小数位是否会累加到影响整个结果精度的程度.

简言之，要确定除法的近似是否合理，需要求算 M 的量级，并比较它与最后结果的差距.

这里先给出 e^{700}Taylor 展开逐项收敛示意图：



图中横坐标代表 Taylor 级数的不同项，即 a_n. 纵坐标代表对应项的数值.

由图像可以看出，a_n 在 n=700 附近出现一个尖锐的峰，这说明 a_n 的收敛是非常迅速的, M(收敛至 1 时的项数)的量级应该与 N(e 的幂指数)相差不大.

利用计算机程序，对 N 和 M 的关系做进一步观测，结果如下：

N	100	200	300	400	500	600	700	800	900	1000
M	270	541	813	1084	1356	1628	1900	2171	2443	2715
M/N	2.700	2.705	2.710	2.710	2.712	2.713	2.714	2.714	2.714	2.715

由上表数据，发现 M(**M 为 展开式收敛至小于 1 后的 第一项的项数**)与 N 量级确实相差不大，总结数据规律不难发现, M/N 近似满足小于 e 这一条件，即：

$$M < eN \tag{7}$$

而由前文的描述可知, M 满足不等式组：

$$\begin{cases} (M-1)! \leq N^{M-1} & (8.1) \\ M! > N^M & (8.2) \end{cases}$$

由上式易得：

$$M > N \tag{9}$$

是(8) 式成立的一个必要不充分条件(若(8)成立，则(9)必然成立).

可以猜想，(8)式成立的另一个必要不充分条件是(7)式.(若(8)成立，则(7)必然成立)
很遗憾，关于这一点的证明，笔者并没有想到一个较好的方法.

由上面的讨论，我们了解了 M 的量级. 对于参与加和每一项来说，忽略小数位造成的误差不超过 1.0，因而整个求和过程中因忽略小数引起的误差不超过 M.
而 M 被认为是一个与 N 量级相当的数，在 N 取大数的情况下，M 相对于 e^N 可以忽略.

由上面的讨论，可知，用此算法计算出的 e^N 数值，有很大的几率是合理的.(这里之所以这样说，是因为(7)式只是用唯象法得出的结论，并没有进行严格的数学证明)
下面给出其算法：

Step1：判断输入数字的大小，若小于 700，直接运算，大于 700，进行 Taylor 展开运算.
Step2：计算 Taylor 级数中第一项的大小
Step3：并判断是否小于 1.
Step4：如果大于 1，将此项加入到总和中，并进行下一项的计算，重复 2、3 步骤；
如果小于 1，则停止循环，并输出加和结果.

三、计算结果及分析

(一)计算结果

下面列出本程序计算结果：

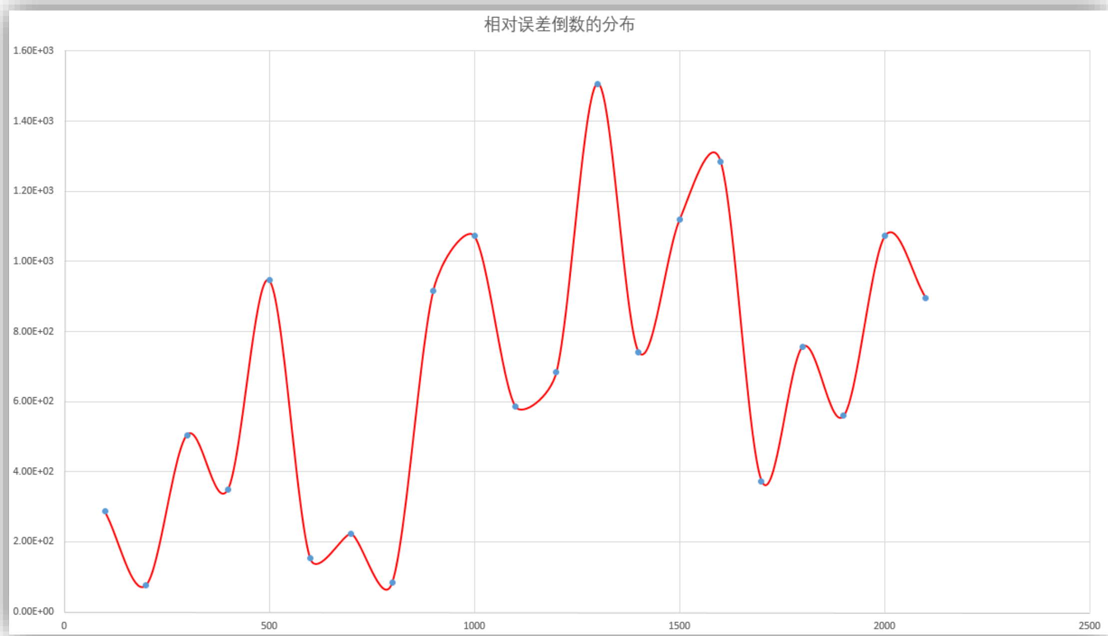
在 Linux(Ubuntu) GCC64 位编译环境下考虑 sqrt(2πN)下输出的结果(表 1)

```
请输入 N 的范围:1<N<10000
请输入数据取点个数:100
请输入要保存的有效数字:6
方程右侧是否引入最后一项修正项?(N/Y):y

======================================================================
```

N	N!	Appro_Val	Differ	1/(Rela_Differ)
0	1E0	0E0	1E0	1E0
100	933262E152	930018E152	324316E150	287E0
200	788657E369	778355E369	103021E368	76E0
300	306057E609	305452E609	605160E606	505E0
400	640345E863	638518E863	182630E861	350E0
500	122013E1129	121884E1129	128968E1126	946E0
600	126557E1403	125736E1403	820821E1400	154E0
700	242204E1684	241119E1684	108415E1682	223E0
800	771053E1971	761969E1971	908391E1969	84E0
900	675268E2264	674531E2264	736867E2261	916E0
1000	402387E2562	402011E2562	375294E2559	1072E0
1100	534370E2864	533459E2864	911079E2861	586E0
1200	635078E3170	634151E3170	927677E3167	684E0
1300	315951E3480	315742E3480	209584E3477	1507E0
1400	346062E3793	345596E3793	465952E3790	742E0
1500	481199E4109	480770E4109	429665E4106	1119E0
1600	527197E4428	526787E4428	410342E4425	1284E0
1700	299835E4750	299032E4750	802416E4747	373E0
1800	612615E5074	611807E5074	808489E5071	757E0
1900	324199E5401	323620E5401	578776E5398	560E0
2000	331627E5730	331318E5730	309117E5727	1072E0
2100	503876E6061	503313E6061	562718E6058	895E0

由于 c 语言不自带绘图软件，因此所有的数据绘图都将借助于 Excel 表格实现. (这确实是一种非常低效的手段，但同时很简单)然而，excel 表格所能承受的最大数字也时双精度范围(最大 1.7E+308)，因此无法绘制出上表中第二列、第三列和第四列(分别代表公式的左侧的数值大小，右侧数值大小，二、三列的差值)的图像，只能给出相对误差倒数的图像.
如下图所示：



图中，横坐标代表带入公式的不同的整数值，纵坐标代表方程两侧相对误差的倒数．也即，纵轴数值越大，Stirling 公式左右两侧近似程度越好．

可以设想，横轴与纵轴应满足较好的正相关关系．

然而，上图所展示的对应关系其实是相当糟糕的．并不是预想的完美的正比关系，而是出现了波动．

关于本图，将在数据分析中具体讨论．

在 Linux(Ubuntu) GCC64 位编译环境下不考虑 sqrt(2πN)下输出的结果(部分)(表 2)

请输入 N 的范围:1<N<700
请输入数据取点个数:700
请输入要保存的有效数字:6
方程右侧是否引入最后一项修正项?(N/Y):n

==

N	N!	Appro_Val	Differ	1/(Rela_Differ)
330	282408E684	620042E682	276208E684	1E0
331	934772E686	204923E685	914279E686	1E0
332	310344E689	679322E687	303551E689	1E0
333	103344E692	225874E690	101085E692	1E0
334	345171E694	753291E692	337638E694	1E0
335	115632E697	251975E695	113112E697	1E0
336	388524E699	845378E697	380070E699	1E0
337	130932E702	284469E700	128088E702	1E0
338	442552E704	960085E702	432952E704	1E0
339	150025E707	324988E705	146775E707	1E0
340	510086E709	110333E708	499053E709	1E0
341	173939E712	375685E710	170182E712	1E0
342	594873E714	128296E713	582043E714	1E0
343	204041E717	439416E715	199647E717	0E0
344	701902E719	150939E718	686808E719	1E0
345	242156E722	519985E720	236956E722	1E0
346	837861E724	179655E723	819895E724	1E0
347	290737E727	622504E725	284512E727	1E0
348	101176E730	216320E728	990135E729	1E0
349	353106E732	753875E730	345568E732	1E0
350	123587E735	263479E733	120952E735	1E0
351	433791E737	923495E735	424556E737	1E0
352	152694E740	324608E738	149448E740	1E0
353	539012E742	114424E741	527569E742	1E0
354	190810E745	404490E743	186765E745	1E0
355	677376E747	143391E746	663037E747	1E0
356	241146E750	509757E748	236048E750	1E0
357	860891E752	181728E751	842718E752	1E0
358	308199E755	649679E753	301702E755	1E0
359	110643E758	232909E756	108314E758	1E0
360	398316E760	837310E758	389943E760	1E0
361	143792E763	301850E761	140773E763	1E0
362	520528E765	109118E764	509616E765	1E0
363	188951E768	395555E766	184996E768	1E0
364	687784E770	143784E769	673405E770	1E0
365	251041E773	524094E771	245800E773	1E0
366	918811E775	191556E774	899655E775	1E0
367	337203E778	702054E776	330183E778	1E0
368	124090E781	258004E779	121510E781	1E0
369	457895E783	950747E781	448388E783	1E0
370	169421E786	351301E784	165908E786	1E0
371	628553E788	130157E787	615537E788	1E0
372	233821E791	483533E789	228986E791	1E0
373	872155E793	180116E792	854143E793	1E0
374	326186E796	672733E794	319458E796	1E0

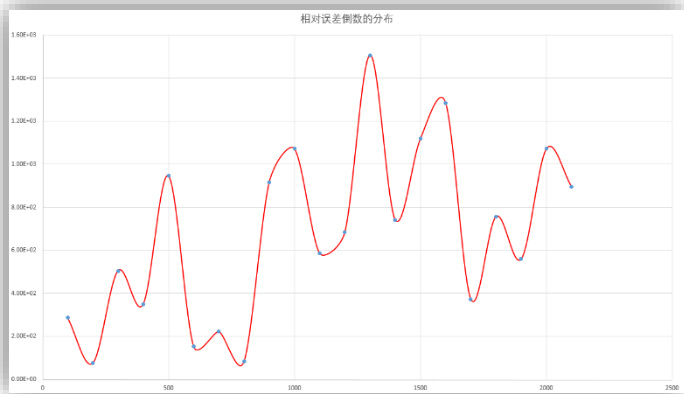
375	122319E799	251938E797	119800E799	1E0
376	459922E801	946029E799	450462E801	1E0
377	173390E804	356179E802	169828E804	1E0
378	655417E806	134457E805	641971E806	1E0
379	248403E809	508922E807	243313E809	1E0
380	943931E811	193136E810	924618E811	1E0
381	359637E814	734882E812	352289E814	1E0
382	137381E817	280357E815	134578E817	1E0
383	526171E819	107236E818	515448E819	1E0
384	202050E822	411253E820	197937E822	0E0
385	777892E824	158126E823	762079E824	1E0
386	300266E827	609578E825	294170E827	0E0
387	116203E830	235601E828	113847E830	1E0
388	450868E832	912957E830	441738E832	1E0
389	175387E835	354683E833	171840E835	1E0
390	684012E837	138149E836	670197E837	1E0

*(由于数据过多，上表只列出了部分数据，要获取更完整的数据，请**运行源文件**或**查看** DATA 中保存的数据样本)*

观察上表发现，如果 Stilring 公式右侧不引入最后一项修正项，最终结果将于左侧差几个数量级．这说明在直接用(1)式做近似时，第三项修正是不可忽略的．

(二)数据分析

忽略修正项所得数据(表 2)的意义已在上一部分说明，重点在于在不忽略修正项的情况下(表 1)算得的数据是否存在问题．



左图为相对误差数值倒数的曲线，这一结果显然是有问题的．(关于为何“显然有问题”将在下文中讨论)

通过验证(N 小于 140，通过 MatLab 验证)，此程序算得的公式左侧 N！与实际值完全符合．而右侧近似值在有效数字保留到 8 位左右时，才与 MatLab 的计算结果出现差异．这说明表 1 的第二列，第三列数据具有很高的可信度．

但是在计算相对误差时，是用二三列的差除以第二列得到．这种操作，随着公式两侧实际数值的不断相互逼近，实际上就变成了不断放大一系列极其微小差别的过程．在计算方程右侧时用到的高精度除法和自然底数幂运算引入了一些近似．这些近似所带来的误差，都将在这一过程中体现出来．

以一个形象的例子来类比这一过程．运算中所做的近似，为最终的结果引入了许多微小的数值涨落．这实际上类似于在信号中引入了背景噪音，当信号不算太微弱时，正如第二列和第三列的数值，这一噪声可以被忽略．但是，如果探测的是一个极其微小的信号(如两个强度极其接近的信号的差)，那么背景噪声将不能被忽略，它甚至可以完全覆盖探测信号．

上述图像所展现的波动，实际上就是由于这种“噪声”引起的．

因此，引入的精确到个位的高精度算法，能较为精确的求得 Stirling 公式两侧数值的大小，但对于相对误差的分析却无能为力．

在电学实验中，可以利用仪器，例如锁相放大器，降低噪声，放大信号．可以推测，在高精度运算中，也可以考虑引入降噪运算．但这些已经超出了笔者所掌握的知识范围，不再做详细讨论．

四、高精度算法的内在问题

上面介绍了在高精度运算在计算本问题时的一定的可行性，但同时这一算法也存在缺．

主要有以下几点：

Ⅰ．内存浪费

在数据的储存过程中，实际上时使用了一个整形数(最大值 4294967296)存放了一个小于 10 的正整数．这是对内存资源的严重浪费．这一问题的解决，可以依靠从新定义一个数组元素中存储的位数来修正．

Ⅱ.运算效率低

在计算 e^N 时，用到了 Taylor 级数展开相加的方式．但是，对于一个大数，如 100,000．在计算其自然底数幂的大小时，就需要进行大于 270,000 次的加和，才能最终收敛．而每一次加和中的运算量也是十分巨大的．经实验，在 N>2000 后，方程右侧的计算就变得极其缓慢了．(2.5GHz, e^2000 大约需要计算 12 分钟)目前这一问题的解决方法只有优化算法和内存调控．

五、一种简化公式的运算方法

(一)算法的原理

文章开头就提到，之所以引入高精度运算，就是因为 N！ 和 N^N 随 N 的增大急速增加，以至于系统所允许的最大数值无法存储其结果．然而，如果从公式的角度出发，对其本身进行简化，那么在算法实现上的复杂程度，可能会以几何递减关系下降．

现对 Stirling 公式进行简化，方程两边同时取对数：

$$lnN! = NlnN - N + \frac{1}{2}\ln(2\pi N) \tag{9}$$

而 lnN!在实际运算时可化为：

$$lnN! = ln1 + ln2 + ln3 + \cdots + lnN \tag{10}$$

这样一来，所以有的数都降低了一个尺度，自然就可以用 double 类型完成全部运算.

此算法较为简单，不再赘述．具体可参考附录 2.

(二)计算结果与数据分析

(i)计算结果

① 在 Linux(Ubuntu16.04) GCC64 位编译环境下考虑右侧修正项编译结果如下

```
========================================================================
****************************lnN!=NlnN+N 验证*********************************
========================================================================
取值范围 1<N<10000
取点个数:1000
是否考虑右侧第三项(Y/N):y
```

N	lnN!	Approve	differ	1/RE_dif	1/N_re_dif
9000	72950.290145	72950.290135	0.000009	7878661687.496268	108000.916063
9010	73041.346052	73041.346043	0.000009	7897283085.521169	108121.222200
9020	73132.413057	73132.413048	0.000009	7915897343.488701	108241.114696
9030	73223.491149	73223.491139	0.000009	7934529107.139473	108360.931642
9040	73314.580314	73314.580304	0.000009	7953190811.729170	108480.842897
9050	73405.680540	73405.680531	0.000009	7971882466.959433	108600.848207
9060	73496.791815	73496.791806	0.000009	7990604082.293540	108720.947315
9070	73587.914127	73587.914118	0.000009	8009342981.439022	108840.967575
9080	73679.047465	73679.047455	0.000009	8028099042.101292	108960.907584
9090	73770.191814	73770.191805	0.000009	8046872141.121353	109080.765935
9100	73861.347164	73861.347155	0.000009	8065687788.355778	109200.888269
9110	73952.513503	73952.513494	0.000009	8084507540.053733	109320.753730
9120	74043.690818	74043.690809	0.000009	8103356948.839007	109440.708334
9130	74134.879097	74134.879088	0.000009	8122248970.365689	109560.926474
9140	74226.078329	74226.078319	0.000009	8141105781.383290	109680.358680
9150	74317.288500	74317.288491	0.000009	8160044064.463139	109800.577871
9160	74408.509601	74408.509592	0.000009	8178985862.337975	109920.533425
9170	74499.741617	74499.741608	0.000009	8197944039.520932	110040.398952
9180	74590.984538	74590.984529	0.000009	8216957982.193829	110160.702779
9190	74682.238352	74682.238343	0.000009	8235975095.350787	110280.739012
9200	74773.503047	74773.503038	0.000009	8255021636.999264	110400.859763
9210	74864.778611	74864.778601	0.000009	8274084303.110383	110520.886979
9220	74956.065031	74956.065022	0.000009	8293162960.878489	110640.819199
9230	75047.362298	75047.362288	0.000009	8312270874.102183	110760.833471

9240	75138.670397	75138.670388	0.000009	8331421487.228025	110881.108390
9250	75229.989319	75229.989310	0.000009	8350560984.258119	111000.927620
9260	75321.319051	75321.319042	0.000009	8369743092.122989	111121.006088
9270	75412.659582	75412.659573	0.000009	8388927208.796792	111240.804419
9280	75504.010899	75504.010890	0.000009	8408153935.902439	111360.861334
9290	75595.372992	75595.372983	0.000009	8427409738.828526	111480.996826
9300	75686.745848	75686.745839	0.000009	8446694616.668588	111601.210561
9310	75778.129457	75778.129448	0.000009	8465994804.601811	111721.320577
9320	75869.523806	75869.523797	0.000009	8485296352.551327	111841.143343
9330	75960.928884	75960.928875	0.000009	8504626692.657558	111961.040999
9340	76052.344680	76052.344671	0.000009	8523999724.272711	112081.196002
9350	76143.771182	76143.771173	0.000009	8543359787.329203	112200.876404
9360	76235.208378	76235.208369	0.000009	8562776437.024210	112320.996254
9370	76326.656258	76326.656249	0.000009	8582221864.295995	112441.189598
9380	76418.114809	76418.114800	0.000009	8601681975.429131	112561.271710
9390	76509.584021	76509.584012	0.000009	8621170761.093494	112681.425815
9400	76601.063882	76601.063873	0.000009	8640674032.530951	112801.466394
9410	76692.554380	76692.554371	0.000009	8660234334.406071	112921.948562
9420	76784.055505	76784.055496	0.000009	8679766276.497276	113041.758623
9430	76875.567245	76875.567236	0.000009	8699340910.756443	113161.823195
9440	76967.089589	76967.089580	0.000009	8718929626.388716	113281.769605
9450	77058.622526	77058.622517	0.000009	8738517852.662998	113401.409155
9460	77150.166044	77150.166035	0.000009	8758177635.312035	113521.676745
9470	77241.720133	77241.720124	0.000009	8777836816.068485	113641.635917
9480	77333.284780	77333.284771	0.000009	8797553354.042984	113762.037211
9490	77424.859976	77424.859967	0.000009	8817283692.304712	113882.316449
9500	77516.445708	77516.445699	0.000009	8836998356.791710	114002.093806
9510	77608.041966	77608.041957	0.000009	8856755736.702425	114122.123383
9520	77699.648739	77699.648730	0.000009	8876541209.132729	114242.216059
9530	77791.266015	77791.266007	0.000009	8896325152.571405	114361.990799
9540	77882.893784	77882.893776	0.000009	8916196386.175943	114482.589108
9550	77974.532035	77974.532026	0.000009	8936036262.316902	114602.486432
9560	78066.180757	78066.180748	0.000009	8955903988.701225	114722.443638
9570	78157.839938	78157.839929	0.000009	8975799552.010534	114842.460311
9580	78249.509568	78249.509559	0.000009	8995737987.724079	114962.728358
9590	78341.189635	78341.189627	0.000009	9015689232.734148	115082.863120
9600	78432.880130	78432.880121	0.000009	9035622830.398642	115202.476716
9610	78524.581041	78524.581032	0.000009	9055599109.302610	115322.339217
9620	78616.292357	78616.292348	0.000009	9075648693.948912	115442.839264
9630	78708.014068	78708.014059	0.000009	9095664944.075621	115562.619714
9640	78799.746162	78799.746153	0.000009	9115708630.043859	115682.454246
9650	78891.488629	78891.488620	0.000009	9135810524.929924	115802.732709
9660	78983.241458	78983.241450	0.000009	9155893685.539078	115922.479354
9670	79075.004639	79075.004631	0.000009	9175988635.039883	116042.082001
9680	79166.778161	79166.778152	0.000009	9196141839.434481	116162.128061
9690	79258.562012	79258.562004	0.000009	9216306795.269491	116282.029229
9700	79350.356183	79350.356175	0.000009	9236545914.513256	116402.572516
9710	79442.160663	79442.160654	0.000009	9256765456.057198	116522.575669
9720	79533.975441	79533.975432	0.000009	9276996438.976435	116642.430541
9730	79625.800506	79625.800498	0.000009	9297286082.336182	116762.730624
9740	79717.635849	79717.635840	0.000009	9317539582.511484	116882.285108
9750	79809.481457	79809.481449	0.000009	9337883498.030907	117002.682292
9760	79901.337322	79901.337313	0.000009	9358222720.162819	117122.728913
9770	79993.203432	79993.203423	0.000009	9378588922.426435	117242.822099
9780	80085.079777	80085.079768	0.000009	9398966030.311649	117362.760942
9790	80176.966346	80176.966338	0.000009	9419386073.626472	117482.945433
9800	80268.863130	80268.863121	0.000009	9439849183.181683	117603.376324
9810	80360.770116	80360.770108	0.000008	9460274458.196026	117723.046020
9820	80452.687296	80452.687288	0.000008	9480758803.143490	117843.161592
9830	80544.614659	80544.614651	0.000008	9501253573.991581	117963.117542
9840	80636.552195	80636.552186	0.000008	9521807674.775707	118083.520860
9850	80728.499892	80728.499883	0.000008	9542355737.249687	118203.560258
9860	80820.457741	80820.457733	0.000008	9562913888.977262	118323.436497
9870	80912.425732	80912.425723	0.000008	9583514982.533567	118443.556130
9880	81004.403853	81004.403845	0.000008	9604142577.349266	118563.715338
9890	81096.392096	81096.392088	0.000008	9624780023.390738	118683.708645
9900	81188.390450	81188.390441	0.000008	9645410460.214771	118803.328913
9910	81280.398904	81280.398895	0.000008	9666100454.520817	118923.396356
9920	81372.417448	81372.417439	0.000008	9686816732.245119	119043.500517
9930	81464.446072	81464.446064	0.000008	9707542430.737549	119163.434247
9940	81556.484767	81556.484758	0.000008	9728311134.079729	119283.609884
9950	81648.533521	81648.533512	0.000008	9749089092.244507	119403.613221
9960	81740.592325	81740.592316	0.000008	9769893107.805746	119523.650372
9970	81832.661168	81832.661160	0.000008	9790689057.637260	119643.304204
9980	81924.740042	81924.740033	0.000008	9811544987.070269	119763.406567
9990	82016.828935	82016.828926	0.000008	9832409725.946045	119883.332021
10000	82108.927837	82108.927828	0.000008	9853317493.992563	120003.497890

Program Complete!!

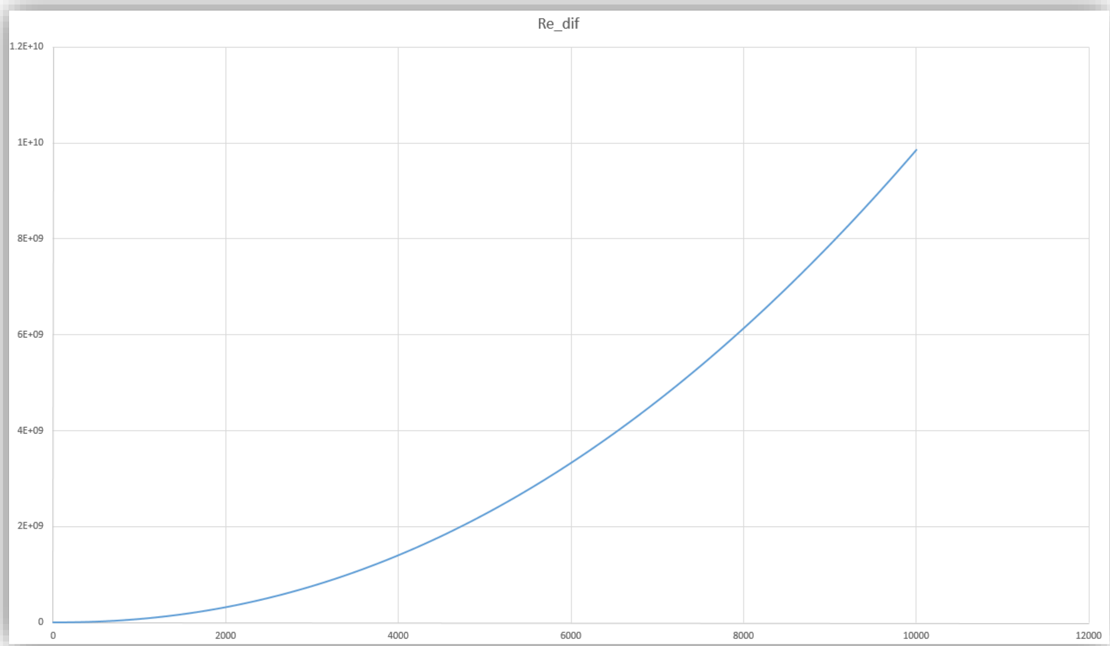
*(由于数据过多，上表只列出了部分数据，要获取更完整的数据，请**运行源文件**或**查看** DATA 中保存的数据样本)*

上表中第一列是方程两侧计算输入的整数值，第二列是公式左侧 lnN！的值的大小，第三列是右侧近似值的大小，第四列两列数值之差(differ)，第五列是**相对误差的倒数**，第六列是在对应的 N 下(1)公式两侧数值相对误差的倒数.

其中前五列是容易理解的，第六列的计算将在数据分析中讨论.

下面给出公式(9)相对误差倒数的变化曲线(D 线).

注：由于左右两侧的变化曲线(暂称之为 L 线与 R 线)符合的过于好，以至于在图中无法分辨出两条线.因此，只给出相对误差倒数的变化曲线.



上图中，横坐标代表带入公式的不同的整数值，纵坐标代表方程两侧相对误差的倒数．即，纵轴数值越大, Stirling 公式左右两侧近似程度越好.

② 在 Linux(Ubuntu16.04) GCC64 位编译环境下不考虑右侧修正项编译结果如下

```
============================================================================
****************************lnN!=NlnN+N 验证*********************************
============================================================================
取值范围 1<N<10000
取点个数:1000
是否考虑右侧第三项(Y/N):n
```

N	lnN!	Approve	differ	1/RE_dif	1/N_re_dif
9000	72950.290145	72944.818707	5.471438	13332.928906	1.004223
9010	73041.346052	73035.874059	5.471993	13348.216385	1.004221
9020	73132.413057	73126.940510	5.472548	13363.504293	1.004218
9030	73223.491149	73218.018047	5.473102	13378.792630	1.004216
9040	73314.580314	73309.106659	5.473655	13394.081396	1.004214
9050	73405.680540	73400.206332	5.474208	13409.370591	1.004211
9060	73496.791815	73491.317055	5.474760	13424.660213	1.004209
9070	73587.914127	73582.438816	5.475311	13439.950262	1.004207
9080	73679.047465	73673.571602	5.475862	13455.240737	1.004204
9090	73770.191814	73764.715401	5.476413	13470.531639	1.004202
9100	73861.347164	73855.870202	5.476963	13485.822966	1.004200
9110	73952.513503	73947.035991	5.477512	13501.114718	1.004197
9120	74043.690818	74038.212758	5.478060	13516.406894	1.004195
9130	74134.879097	74129.400489	5.478608	13531.699494	1.004193
9140	74226.078329	74220.599173	5.479155	13546.992517	1.004190
9150	74317.288500	74311.808798	5.479702	13562.285964	1.004188
9160	74408.509601	74403.029352	5.480248	13577.579832	1.004186
9170	74499.741617	74494.260823	5.480794	13592.874122	1.004183
9180	74590.984538	74585.503200	5.481339	13608.168833	1.004181
9190	74682.238352	74676.756469	5.481883	13623.463965	1.004179
9200	74773.503047	74768.020620	5.482427	13638.759516	1.004177
9210	74864.778611	74859.295640	5.482970	13654.055488	1.004174
9220	74956.065031	74950.581519	5.483513	13669.351878	1.004172
9230	75047.362298	75041.878243	5.484055	13684.648687	1.004170
9240	75138.670397	75133.185801	5.484596	13699.945914	1.004168
9250	75229.989319	75224.504182	5.485137	13715.243558	1.004165
9260	75321.319051	75315.833374	5.485677	13730.541619	1.004163
9270	75412.659582	75407.173365	5.486217	13745.840096	1.004161
9280	75504.010899	75498.524143	5.486756	13761.138990	1.004158
9290	75595.372992	75589.885697	5.487294	13776.438298	1.004156
9300	75686.745848	75681.258016	5.487832	13791.738022	1.004154
9310	75778.129457	75772.641087	5.488370	13807.038160	1.004152
9320	75869.523806	75864.034900	5.488906	13822.338711	1.004150
9330	75960.928884	75955.439442	5.489443	13837.639676	1.004147
9340	76052.344680	76046.854702	5.489978	13852.941053	1.004145
9350	76143.771182	76138.280668	5.490513	13868.242843	1.004143
9360	76235.208378	76229.717330	5.491048	13883.545045	1.004141
9370	76326.656258	76321.164676	5.491582	13898.847658	1.004138
9380	76418.114809	76412.622694	5.492115	13914.150681	1.004136
9390	76509.584021	76504.091373	5.492648	13929.454115	1.004134
9400	76601.063882	76595.570702	5.493180	13944.757958	1.004132
9410	76692.554380	76687.060669	5.493712	13960.062211	1.004130
9420	76784.055505	76778.561263	5.494243	13975.366872	1.004127
9430	76875.567245	76870.072472	5.494773	13990.671941	1.004125
9440	76967.089589	76961.594286	5.495303	14005.977418	1.004123

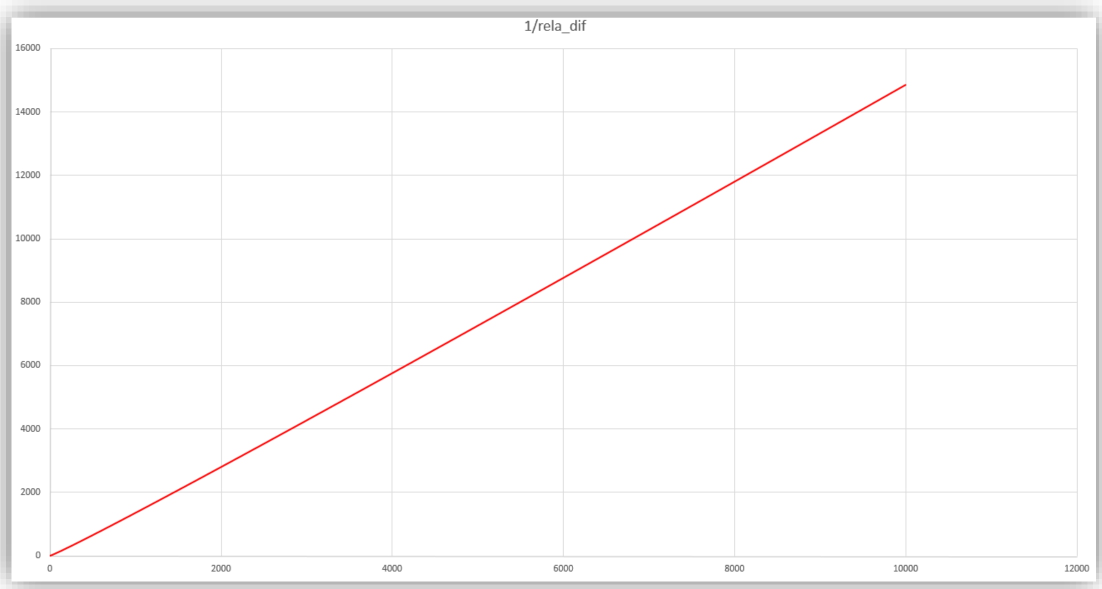
9450	77058.622526	77053.126694	5.495832	14021.283302	1.004121
9460	77150.166044	77144.669683	5.496361	14036.589593	1.004119
9470	77241.720133	77236.223243	5.496889	14051.896290	1.004116
9480	77333.284780	77327.787363	5.497417	14067.203393	1.004114
9490	77424.859976	77419.362031	5.497944	14082.510901	1.004112
9500	77516.445708	77510.947237	5.498471	14097.818814	1.004110
9510	77608.041966	77602.542969	5.498997	14113.127130	1.004108
9520	77699.648739	77694.149217	5.499522	14128.435851	1.004106
9530	77791.266015	77785.765968	5.500047	14143.744974	1.004103
9540	77882.893784	77877.393213	5.500572	14159.054501	1.004101
9550	77974.532035	77969.030940	5.501095	14174.364429	1.004099
9560	78066.180757	78060.679138	5.501619	14189.674760	1.004097
9570	78157.839938	78152.337796	5.502141	14204.985491	1.004095
9580	78249.509568	78244.006904	5.502664	14220.296624	1.004093
9590	78341.189635	78335.686450	5.503185	14235.608156	1.004090
9600	78432.880130	78427.376424	5.503706	14250.920088	1.004088
9610	78524.581041	78519.076814	5.504227	14266.232420	1.004086
9620	78616.292357	78610.787610	5.504747	14281.545151	1.004084
9630	78708.014068	78702.508801	5.505266	14296.858279	1.004082
9640	78799.746162	78794.240377	5.505785	14312.171806	1.004080
9650	78891.488629	78885.982325	5.506304	14327.485730	1.004078
9660	78983.241458	78977.734637	5.506822	14342.800051	1.004076
9670	79075.004639	79069.497300	5.507339	14358.114768	1.004073
9680	79166.778161	79161.270305	5.507856	14373.429881	1.004071
9690	79258.562012	79253.053640	5.508372	14388.745390	1.004069
9700	79350.356183	79344.847296	5.508888	14404.061294	1.004067
9710	79442.160663	79436.651260	5.509403	14419.377592	1.004065
9720	79533.975441	79528.465523	5.509918	14434.694284	1.004063
9730	79625.800506	79620.290075	5.510432	14450.011370	1.004061
9740	79717.635849	79712.124903	5.510945	14465.328849	1.004059
9750	79809.481457	79803.969999	5.511458	14480.646721	1.004057
9760	79901.337322	79895.825351	5.511971	14495.964985	1.004055
9770	79993.203432	79987.690949	5.512483	14511.283640	1.004052
9780	80085.079777	80079.566782	5.512994	14526.602687	1.004050
9790	80176.966346	80171.452841	5.513505	14541.922124	1.004048
9800	80268.863130	80263.349114	5.514016	14557.241952	1.004046
9810	80360.770116	80355.255591	5.514526	14572.562170	1.004044
9820	80452.687296	80447.172261	5.515035	14587.882777	1.004042
9830	80544.614659	80539.099115	5.515544	14603.203773	1.004040
9840	80636.552195	80631.036142	5.516052	14618.525157	1.004038
9850	80728.499892	80722.983332	5.516560	14633.846930	1.004036
9860	80820.457741	80814.940673	5.517068	14649.169090	1.004034
9870	80912.425732	80906.908157	5.517575	14664.491637	1.004032
9880	81004.403853	80998.885773	5.518081	14679.814571	1.004030
9890	81096.392096	81090.873509	5.518587	14695.137891	1.004028
9900	81188.390450	81182.871358	5.519092	14710.461596	1.004026
9910	81280.398904	81274.879307	5.519597	14725.785687	1.004024
9920	81372.417448	81366.897347	5.520101	14741.110163	1.004022
9930	81464.446072	81458.925467	5.520605	14756.435023	1.004020
9940	81556.484767	81550.963659	5.521108	14771.760267	1.004017
9950	81648.533521	81643.011910	5.521611	14787.085895	1.004015
9960	81740.592325	81735.070212	5.522113	14802.411905	1.004013
9970	81832.661168	81827.138554	5.522615	14817.738298	1.004011
9980	81924.740042	81919.216926	5.523116	14833.065074	1.004009
9990	82016.828935	82011.305318	5.523617	14848.392231	1.004007
10000	82108.927837	82103.403720	5.524117	14863.719769	1.004005

Program Complete!!

*(由于数据过多，上表只列出了部分数据，要获取更完整的数据，请**运行源文件**或**查看** DATA 中保存的数据样本)*

不同列的数据解释同上.

下面给出公式(9)相对误差**倒数**的变化曲线(D 线).



对比考虑右侧修正与不考虑右侧修正的两计算数值，不难发现，如果考虑了修正项(sqrt(2πN))，相对误差的减小是非常迅速的．在 N=6000 时就降低到了 10^{-9} 量级(也就是十亿分之一)的水平而同条件下，不引入修正项，只降到了 1/9000 左右．但这仍不影响不引入修正项 Stirling 近似公式成立．因为 1/9000 也是一个相当小的量了．(当然还需要视具体的情况而定)

也即，取对数之后的 Stirling 公式与取对数之前不同，第三项修正是可以丢掉的．

(ii)数据分析

之前提到，高精度运算由于近似而产生的噪声掩盖了相对误差，导致无法算出结果．而这一计算结果上的缺陷恰好被该部分所用的算法弥补了．

下面做简要说明：

不妨把未取对数前的 Stirling 公式左右两侧的值分别用 L 和 R 代替．
则在简化运算公式后：

$$lnL = lnR$$

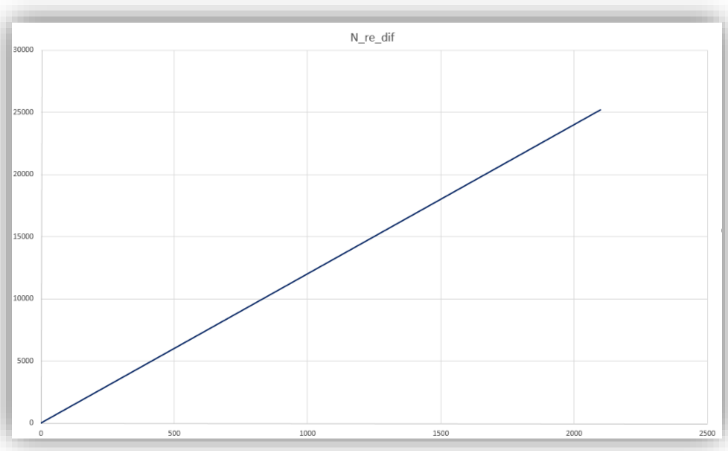
两侧的差(differ)：

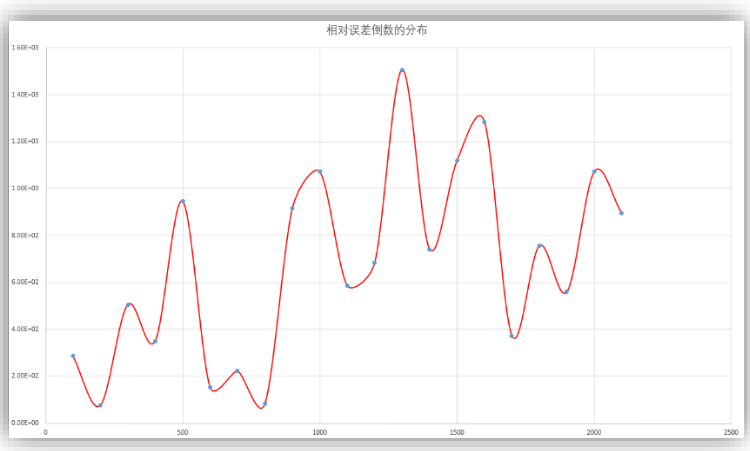
$$d = lnL - lnR = ln\frac{L}{R}$$

对上式做变换得到：

$$\delta = \frac{L-R}{L} = 1 - e^{-d}$$

换言之，依靠取对数后两边的绝对误差d，就可以确定未取对数之前两侧的相对误差δ．
对比两种算法所得相同区间的图像(当然是取对数算法算得的值更准确)，就不难看出高精度算法的结果是不准确的．因此前文才说，那一结果是“显然是有问题的”．





六、总结

经过对 Stirling 公式的数值验证计算，进一步强化了对大数处理的困难度的认识. 在计算过程中，适当的近似是必要的，但同时也可能引起一系列不利的影响. 在进行算法的构思之前，应用数学或物理工具，对问题本身做适当的简化，或许会起到事半功倍的效果.

附录 1：

高精度算法源码：

```
/*
  程序:斯特林公式分析程序

    目的:验证斯特林公式,并计算不同数值下,方程的近似程度

    功能:通过高精度运算计算方程结果

    难点:1,高精度除法的定义
         2,自然对数的高幂次(1000 以上)运算
         3,N!的高精度运算(10000!左右)

   日期:2016 年 9 月 7 日

     预计完成时间:2016 年 9 月 12 日

     预计编程时长:未知

*/

//==========================================================================================
//##########################################################################################
//##########################################################################################
//==========================================================================================



# include <stdio.h>
# include <math.h>
#define CON 100000  //定义数值位数储单元存常量.10,000!为 40000 位左右;100,000!为 500,000 位左右


//**************************************************************************
//=========================声明函数========================================
//**************************************************************************

 int * add(int*, int*);//声明高精度加法运算
 int * mult(int *, double);//声明高精度计算数乘法则函数(大数乘小数)
 int * dec(int *, int *);//声明高精度计算减法法则函数
 int * div(int *, int *);//声明高精度计算除法法则函数

 int * N_1(int);//声明 N!计算函数(阶乘运算)
 int * N_C(int, int);//声明 N^C 计算函数(幂运算)
 int * e_N(int);//声明 e^N 计算函数

 void reput(int*);//定义数组清零函数
 int pot(int *, int);//声明计算数值总位数的函数
 void pri(int *, int);//声明数组数值输出函数

//***************************************************************************
//==========定义全局变量数组,以存储各函数输出结果==========================
//****************************************************************************


 int add_g[CON];
 int mult_g[CON];
 int dec_g[CON];
 int div_g[CON];
 int N_1_g[CON];
 int N_C_g[CON];
 int e_N_g[CON];



//*****************************************************************************
//=======================主函数============================================
//*****************************************************************************


 int main(void)
  {
     int n;//定义取数范围变量
     int d;//定义取数间隔变量
     int point;//定义取点个数变量
     char ch;//定义修正项取舍判断控制符
```

```
    int sig;//定义保留的有效数字变量

    int left[CON];//定义 Stirling 公式左侧 N!的存储数组,可容纳 1,000,000 位的数
    int right1[CON];//定义 Stirling 公式右侧近似值的存储数组 part1,可容纳 1,000,000 位的数
    int right2[CON];//定义 Stirling 公式右侧近似值的存储数组 part2,可容纳 1,000,000 位的数
    double right3;//定义 Stirling 公式右侧近似值的存储数组 part3
    int right12[CON];//定义 Stirling 公式右侧近似值的存储数组 part3&part2,可容纳 1,000,000 位的数
    int right123[CON];//定义 Stirling 公式右侧近似值的存储数组总和,可容纳 1,000,000 位的数
    int differ[CON];//定义 Stirling 公式两侧数值之差的数值存储数组,可容纳 1,000,000 位的数
    int pec_dif[CON];//定义 Stirling 公式两侧相对误差的倒数数值存储数组,可容纳 1,000,000 位的数

    int * cp_str;//定义数组拷贝指针

    printf("请输入 N 的范围:1<N<");
    scanf("%d", &n);
    printf("请输入数据取点个数:");
    scanf("%d", &point);
    printf("请输入要保存的有效数字:");
    scanf("%d", &sig);
    printf("方程右侧是否引入最后一项修正项?(N/Y):");
    scanf(" %c", &ch);
    printf("\n=========================================================================\n");

    printf("\nN                     N!                    Appro_Val                    Differ                   1/(Rela_Differ)\n\n");         //调整输入界面

    d = n / point;      //计算取点间隔

//************************************开始取点计算*************************

    for(int i=0; i<=n; i+=d)//!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
      {
       //======================计算数值并拷贝至主函数===============

         //-----------------方程左侧-------------------

        cp_str = N_1(i);  //初始化指针至 N! 存储数组的 头部

                                                  //printf("左侧赋值完毕\n");

        for (int j=0; j<CON; j++)
           {
             left[j] = *cp_str;   //单个数组元素赋值
                                                  //printf("%d", left[j]);
             cp_str++;   //指针移动

           }             //实现 N!_数组拷贝,遍历一百万个数组元素

                                                  //printf("左侧计算完毕\n");

//printf("\n###################\n");
//pri(left, 10);                  //N!计算数值探针
//printf("\n###################\n");

         printf("%d                     ", i);
         pri(left, sig);    //sig 要保留的有效数字
         printf("                      ");


         //----------------方程右侧--------------------

        cp_str = N_C(i, i);//初始化指针至 N^N 存储数组的 头部
         for (int j=0; j<CON; j++)
           {
             right1[j] = *cp_str;
                                                   //printf("%d", *cp_str);
             cp_str++;
           }            //实现 N^N 存储数组拷贝
                                                    //printf("右侧 1 计算完毕\n");
//printf("\n###################\n");
//pri(right1, 10);                  //N^N 计算数值探针
//printf("\n###################\n");

        cp_str = e_N(i);//初始化指针至 e^N 存储数组的 头部
         for (int j=0; j<CON; j++)
           {
             right2[j] = *cp_str;
                                                     //printf("%d", *cp_str);
             cp_str++;
           }            //实现 e^N 存储数组拷贝

//printf("\n###################\n");
//pri(right2, 10);                  //exp(N)计算数值探针
//printf("\n###################\n");


                                                      //printf("右侧 2 计算完毕\n");

        if('Y' == ch || 'y' == ch)
           right3 = sqrt(2 * M_PI * i);
        else
           right3 = 1.00;                  //判断是否引入第三项修正

                                                      //printf("分步计算完毕\n");

//printf("\n###################\n");
//printf("%lf", right3);                  //right3 计算数值探针
//printf("\n###################\n");

       //====================右侧的数组的合成=========================

         cp_str = div(right1, right2);//初始化指针至 p1*p2 结果 存储数组 头部
```

```
        for (int j=0; j<CON; j++)
          {
            right12[j] = *cp_str;
            cp_str++;
          }           //实现 p1*p2 存储数组拷贝


//printf("\n###################\n");
//pri(right12, 10);                  //right12 计算数值探针
//printf("\n###################\n");
                                                                 //printf("12 计算完毕\n");
        cp_str = mult(right12, right3);//初始化指针至 全部右侧结果 存储数组 头部
        for (int j=0; j<CON; j++)
          {
            right123[j] = *cp_str;
            cp_str++;
          }           //实现 p1*p2*p3 存储数组拷贝
                                                                  //printf("123 计算完毕\n");

         pri(right123, sig);
         printf("                    ");

       //=====================比较两侧数值(作差)=============================
         cp_str = dec(left, right123);//初始化指针至 两侧差的结果 存储数组的 头部
         for (int j=0; j<CON; j++)
           {
             differ[j] = *cp_str;
             cp_str++;
           }           //实现 p1*p2*p3 存储数组拷贝

         pri(differ, sig);
        printf("                    ");


                                                                   //printf("作差计算完毕\n");
       //=======================计算相对误差的倒数===========================
          cp_str = div(left, differ);
          for(int j=0; j<CON; j++)
            {
               pec_dif[j] = *cp_str;
               cp_str++;
            }

        pri(pec_dif, sig);
        printf("\n\n\n");   //结果输出


                                                                 //printf("误差计算完毕\n");

      }

    return 0;
 }
//==============================================================================================
//*********************************************函数体定义**************************************
//==============================================================================================


//1***************************************定义高精度加法运算*********************************
  int * add(int * a, int * b)
    {
      reput(add_g);//清零加和数组

      int pot_a, pot_b, maxpot;

      pot_a = pot(a, CON);
      pot_b = pot(b, CON); //分别计算输入两个数的位数

      if(pot_a > pot_b)
         maxpot = pot_a;
      else
         maxpot = pot_b;  //取两数位数的最大值


      for (int i=0; i<maxpot; i++)
        {
            add_g[i] = (*a) + (*b);
            a++;
            b++;
        }              //对应元素相加,直到最高位.

      for(int i=0; i<(maxpot+2); i++)//两个 maxpot 位的数字相加,最高是 maxpot+1 位
        {
            if(add_g[i]>9)
             {
              add_g[i+1] += add_g[i] / 10;
              add_g[i] = add_g[i] % 10;
             }
        }             //进位处理,取 maxpot+2 位以确保加和空间充足(maxpot+1 也可)

      return add_g;       //返回结果数组的头地址
    }


//2************************************定义高精度计算乘法法则函数**************************
  int * mult(int * a, double b)
    {
      reput(mult_g);//清零数乘数组

      int pot_a;
```

```
        pot_a = pot(a, CON);

         for (int i = 0; i < pot_a; i++)
          {
            mult_g[i] = (*a) * b;
            a++;
          }               //对每一位上对应元素值做关于 b 的乘法;

         for (int i = 0; i < CON; i++)//不能确定 b 的位数,因此遍历全数组空间进行进位
          {
             if (mult_g[i] >= 10)
               {
                  mult_g[i + 1] += mult_g[i] / 10;
                  mult_g[i] %= 10;
               }          //完成进位
          }
         return mult_g;        //输出结果的头地址
    }

  //3**********************************定义高精度计算减法法则函数************************
    int * dec(int * a, int * b)
     {
        reput(dec_g);//清零差数组

        int pot_a, pot_b, maxpot;

        pot_a = pot(a, CON);
        pot_b = pot(b, CON); //计算输入两数值的位数

        if(pot_a > pot_b)
           maxpot = pot_a;
        else
           maxpot = pot_b; //取位数的最大值

        for(int i=0; i<maxpot; i++)
          {
             dec_g[i] = *a - *b;
             a++;
             b++;
          }                  //对应元素相减


        for(int i=0; i<(maxpot-1); i++) //借位运算只持续到第 maxpot-1 位,最高位不再借位
          {
             if(dec_g[i]<0)
               {
                  dec_g[i+1]--;
                  dec_g[i] += 10;
               }
          }               //完成借位运算

        return dec_g;        //返回数组头地址
     }

 //4**************************************定义高精度计算除法法则函数**************************
   int * div(int * a, int * b)
     {
         reput(div_g);//清零商数组

         int * cp1;
         int * cp2;
         int cp_a[CON];
         int cp_b[CON];

         cp1 = a;
         cp2 = b;

         for (int i=0; i<CON; i++)
           {
              cp_a[i] = *cp1;
              cp_b[i] = *cp2;
              cp1++;
              cp2++;
           }
                //拷贝除法数组



         int de_dig;
         int pot_a, pot_b;//定义位数变量
         int borr; //定义进位数变量

         pot_a = pot(cp_a, CON);
         pot_b = pot(cp_b, CON);  //求除数 b 与被除数 a 的位数

                                            //printf("%d %d \n",pot_a, pot_b);

         while((pot_a >= pot_b)&pot_b!=0)
            {
                                            //printf("%d\n", cp_a[2]);

               de_dig = pot_a - pot_b; //计算两个数的位差

               cp_a[pot_a-1] += (cp_a[pot_a]*10);  //最高位的上一位向最高位进 10
               //cp_a[pot_a] = 0;

                                            //printf("\n%d\n", cp_a[pot_a]);

               do
                {
                   for(int i=0; i<pot_b; i++)
```

```
                        cp_a[de_dig+i] -= cp_b[i];  //对应位相减

                     for(int i=0; i<pot_b-1; i++)
                         if(cp_a[de_dig+i]<0)
                           {
                             borr =  (-(cp_a[de_dig+i]/10)+1);

                             cp_a[de_dig+i] += borr * 10;
                             cp_a[de_dig+i+1] -= borr;
                           }                 //借位运算

                      div_g[de_dig]++;

                                                  //printf("%d\n", cp_a[pot_a-1]);

                 }while(cp_a[pot_a-1]>=0); //输出为负(多减一次)的结果

                                                 //printf("bit_over\n" );

                 for(int i=0; i<pot_b; i++)
                 cp_a[de_dig+i] += cp_b[i];  //对应位相加

                 for(int i=0; i<pot_b-1; i++)
                 if(cp_a[de_dig+i]>9)
                   {
                     cp_a[de_dig+i+1] +=  cp_a[de_dig+i] / 10;
                     cp_a[de_dig+i] %= 10;
                   }                   //进位运算
                 div_g[de_dig]--;  //撤回一次作差操作


                 pot_a--;

             }
          return div_g; //输出数据头地址
      }


//5*************************************定义N!计算函数**************************
   int * N_1(int num)
      {
        reput(N_1_g);//清零阶乘数组

        int dig, temp;
        int carry;

        N_1_g[0] = 1;
        dig = 1; //初始化:数字为1,只有一位.

        for (int i=2; i<=num; i++)  //从2到N进行阶乘
          {
             carry = 0;
             for(int j=1; j<=dig; j++)
               {
                  temp = N_1_g[j-1] * i + carry;

                  N_1_g[j-1] = temp % 10;
                  carry = temp /10;     //小于10的部分写进本位,大于的部分写入carry中
               }
               //在旧位数上进行运算

             while(carry) //当旧位数全部赋值完毕,而carry不为0,则创建新位
               {
                  dig++;
                  N_1_g[dig-1] = carry % 10;
                  carry /= 10;
               }
               //数据进位完毕,进入下一个数的乘法循环

          }

        return N_1_g;

      }


//6*************************************定义N^C计算函数**************************
  int * N_C(int cir, int num)
      {
        reput(N_C_g);//        清零幂运算数组

        int dig, temp;
        int carry;

        N_C_g[0] = 1;
        dig = 1;    //初始化:数值为1,位数为一.

        for (int i=1; i<=cir; i++) //控制进行乘法的次数(幂次)
          {
             carry = 0;
             for(int j=1; j<=dig; j++)
               {
                  temp = N_C_g[j-1] * num + carry; //乘法&进位运算 同步进行

                  N_C_g[j-1] = temp % 10;
                  carry = temp /10;
               }
               //在旧位数上进行运算

             while(carry)    //同阶乘的解释
               {
```

```
                dig++;
                N_C_g[dig-1] = carry % 10;
                carry /= 10;
             }
             //如果需要,则创建新位数

        }

      return N_C_g;
   }



//7************************************定义 e^N 计算函数**************************
  int * e_N(int n)
   {
      reput(e_N_g);//清零 exp 数组

   //########N 较小(小于 700)时直接求算####################################

      if(n <= 700)
        {
           double res;
           int i = 0;

           res = exp(n);   //直接计算出结果存入 res 中
                                                   //printf("%lf", res);

           while(res>=1.0)     //将 res 中的值写入数组
             {
                e_N_g[i] = (int)(fmod(res, 10)); //对 res 取余,并强制转换位整型

                                                  //printf("%d", (int)(fmod(res, 10)));
                res /= 10;
                i++;
             }

        }
   //#######如果 N 太大(大于 700),则采用 Taylor 展开求解##########################

      else
        {
           int elem[CON];
           int x_n[CON];
           int n_2[CON];
           int * tep;       //创建拷贝指针
           int j = 1;       //创建 Taylor 展开项数变量

           elem[0] = 1;
           e_N_g[0] = 1;    //准备 Taylor 展开,加和初始化

//printf("\n\n!!!!!!!\n");
//pri(e_N_g, 6);            //初始化位置探针
//printf("\n!!!!!!!\n");

              while(pot(elem, CON)!=0)  //判断 elem 是否收敛至小于 1,以判断是否继续加和
               {
                  tep = N_C(j, n);
                  for(int k=0; k<CON; k++)
                     {
                        x_n[k] = *tep;
                        tep++;
                     }            //拷贝 n^j 数值数组

                  tep = N_1(j);
                  for(int k=0; k<CON; k++)
                     {
                        n_2[k] = *tep;
                        tep++;
                     }             //拷贝 j!数值数组

                  tep = div (x_n, n_2); //计算 n^j/j! 的大小
                  for(int i=0; i<CON; i++)
                     {
                        elem[i] = *tep;     //将新值写入 elem
                        tep++;
                     }                  //计算单个加和项(即 elem)的数值


//pri(elem, 5);
//printf("\n"); //收敛探针
//printf("%d\n", j);

                    j++;

                    for(int i=0; i<CON; i++)
                       e_N_g[i] += elem[i];

                }  //while 结束


               for(int i=0; i<CON-1; i++)
                  {
                     e_N_g[i+1] += e_N_g[i] / 10;
                     e_N_g[i] %= 10;
                  }
          }//else 结束

      return e_N_g;
   }
```

```c
//8***********************************定义计算数值总位数的函数*************************
  int pot(int * p, int len)
    {
        int * cp_str;
        int dig;

        dig = 0;

        cp_str = p + len - 1;   //移动指针至数组尾部

        for (int j=0; j<len; j++)
           {
            if (*cp_str)        //由尾到头(由高位到低位)一次判断,
              {
                dig = len - j;     //遇到非零元则退出停止循环,并记录此时位数
                break;
              }
             cp_str--;
           }

        return dig;              //返回位数

    }



//9***********************************定义数组数值输出函数*************************
  void pri(int * p, int sig)
    {
        int * cp_str;
        int poe_out;
        int pot_p;

        pot_p = pot(p, CON);

        if((pot_p-sig) <= 0)
          poe_out = 0;
        else
          poe_out = pot_p - sig;  //判断有效数字是否大于原数值位数

        if(pot_p==0)
            pot_p = 1;//防止指针溢出

        cp_str = p + (pot_p-1);  //初始化指针至 输出数组尾部

        for (int j=pot_p; j>poe_out; j--)
           {
             printf("%d", *cp_str);     //输出保留的有效数字
             cp_str--;
           }

        printf("E%d", poe_out);   //输出幂次

    }

//10***********************************定义数组归零函数*************************
  void reput(int * a)
    {
        for(int i=0; i<CON; i++)
          {
             *a = 0;

              a++;
          }
    }         //将输入的数组归零




//=====================================================================================
//#####################################################################################
//#####################################################################################
//=====================================================================================


/*

  总结:1,指针的应用在 C 语言中至关重要.这次编程,由于对指正部分知识不熟悉.在函数间传指针过程中遇到很多
       困难.最后不得不放弃函数传指,改用全局变量.计算效率大大降低.今后还需要补习指针的相关知识.
       2,e^N 计算中用到了 Taylor 展开.加和必须在 elem 计算之后立即进行,否则内存溢出.原因未知!
       3,除法计算的定义花费了非常多的时间,主要是因为对传指针知识的生疏,导致在计算除法函数内部,不得不重新
        定义减法,使代码趋于冗长.

  时间:
       2016 年 6 月 10 号 完成
       总计编程时长:22 小时左右

  运行结果:
       在 Linux(Ubuntu) GCC64 位编译器下,运行结果(最大值 1000,取点数 50, 保留 6 位有效数字,考虑右侧修正)
===============================================================================================
请输入 N 的范围:1<N<1000
请输入数据取点个数:50
请输入要保存的有效数字:6
方程右侧是否引入最后一项修正项?(N/Y):y

=======================================================================
//略
=======================================================================

     计算完成耗时:46min
```

```
    参考部分:

        阶乘(N_1)和大数幂运算(N_C)部分是在网络上参考相关资源后编成

*/
```

附录 2:

取对算法源码:

```
/*
   程序:取对数后_斯特林公式_分析程序

    目的:验证取对数后斯特林公式,并计算不同数值下,方程的近似程度

    功能:用 log 函数计算相关数值

    难点:无

   日期:2016 年 9 月 11 日

     预计完成时间:2016 年 9 月 11 日

     预计编程时长:40min

*/



# include <stdio.h>
# include <math.h>

  int main(void)
   {
      double left; //定义左侧数值储存变量
      double right; //定义右侧数值储存变量
      double differ; //定义差值
      double re_dif; //定义相对误差倒数


      int n;//定义取值范围
      int dis; //定义取值间距
      int point; //定义取值个数
      char jud; //定义第三项修正判断符

      printf("\n===========================================================================\n");
      printf("*****************************lnN!=NlnN+N 验证********************************\n");
      printf("===========================================================================\n\n");
                                                  //程序分割线
      printf("取值范围 1<N<");
      scanf("%d", &n);
      printf("取点个数:");
      scanf("%d", &point);
      printf("是否考虑右侧第三项(Y/N):");
      scanf(" %c", &jud);
                       //设置输入参数

      printf("\nN              lnN!              Approve              differ              1/RE_dif\n");
      printf("__________________________________________________________________________________________\n\n");
                        //表头

      dis = n / point;  //计算取点间隔

      for(int i=0; i<=n; i+=dis)
        {
          if(i==0)
            {
              left = 0;
              right =0;
              differ = 0;
              re_dif = 1;
            }
          else
            {
               left = 0.00; //初始化方程左侧

               for(int j=1; j<=i; j++)
                  left += log(j);   //计算 lnN!

               if('Y' == jud || 'y' == jud)
                  right = (i * log(i)) - i + (0.500 * log(2 * M_PI * i));
               else
                  right =(i * log(i)) - i;    //计算方程右侧

               differ = left - right; // 计算左右差值

               re_dif = left / differ; // 计算相对误差的倒数
            }

           printf("\n\n%d           %lf              %lf            %lf             %lf", i, left, right, differ, re_dif);

        }
```

```
    printf("\n\n________________________________________________________________________________\n\n");
    printf("Program Complete!!\n\n");
    return 0;

}



/*
    总结:
        本程序相对简单,只需掌握 math.h 中 log 的相关用法即可迅速完成编程.

    时间:
        完成日期:2016 年 9 月 11 日
        耗时:45min 左右
    结果:
        在 Linux(Ubuntu)  GCC64 位编译环境下的结果:
==========================================================================================================
//%略%
==========================================================================================================


*/
```

附录 3:

Stirling 公式的证明:

令 $a_n = \dfrac{n!}{n^{n+\frac{1}{2}}e^{-n}}$

则 $\dfrac{a_n}{a_{n+1}} = \dfrac{(n+1)^{n+\frac{3}{2}}}{n^{n+\frac{1}{2}}(n+1)e}$

$= \dfrac{(n+1)^{n+\frac{1}{2}}}{n^{n+\frac{1}{2}}e}$

$= \left(1+\dfrac{1}{n}\right)^n \left(1+\dfrac{1}{n}\right)^{\frac{1}{2}} \dfrac{1}{e}$

所以 $\dfrac{a_n}{a_{n+1}} > 1$ 即 $a_n > a_{n+1}$，即单调递减，又由积分放缩法有 $\ln n! > \left(n+\dfrac{1}{2}\right)\ln n - n$

即 $n! > n^{n+\frac{1}{2}}e^{-n}$，即 $a_n > 1$

由单调有界定理 a_n 的极限存在，

设 $A = \lim\limits_{n\to+\infty} a_n$

$A = \lim\limits_{n\to+\infty} \dfrac{n!}{n^{n+\frac{1}{2}}e^{-n}}$

利用Wallis公式，$\dfrac{\pi}{2} = \lim\limits_{n\to+\infty} \dfrac{\left[\frac{(2n)!!}{(2n-1)!!}\right]^2}{2n+1}$

$$
\begin{aligned}
\frac{\pi}{2} &= \lim_{x\to+\infty} \frac{\left[\frac{(2n)!!}{(2n-1)!!}\right]^2}{2n+1} \\
&= \lim_{x\to+\infty} \frac{\left[\frac{(2n)!!(2n)!!}{(2n)!}\right]^2}{2n+1} \\
&= \lim_{x\to+\infty} \frac{2^{4n}\left[\frac{(n!)^2}{(2n)!}\right]^2}{2n+1} \\
&= \lim_{x\to+\infty} \frac{2^{4n}\left[\frac{(An^{n+\frac{1}{2}}e^{-n})^2}{A(2n)^{2n+\frac{1}{2}}e^{-2n}}\right]^2}{2n+1} \\
&= \lim_{x\to+\infty} \frac{2^{4n}\left(2^{-2n-\frac{1}{2}}A\sqrt{n}\right)^2}{2n+1} \\
&= \lim_{x\to+\infty} \frac{2^{4n}A^2 2^{-4n-1} * n}{2n+1} \\
&= \frac{A^2}{4}
\end{aligned}
$$

所以 $A = \sqrt{2\pi}$

$\lim\limits_{n\to+\infty} \dfrac{n!}{n^{n+\frac{1}{2}}e^{-n}} = \sqrt{2\pi}$

即 $\lim\limits_{n\to+\infty} \dfrac{\sqrt{2\pi n}\, n^n e^{-n}}{n!} = 1$

证明参考：百度百科