数值分析第二次大作业

e 的数值计算

张蔚桐 2015011493 自 55

目录

1 Taylor 级数展开求 e

1.1 原理分析

考虑对函数 $f(x)=\exp(x)$ 进行 Maclaurin 展开

$$\exp(x)=\sum_{i=0}^{\infty}\left.\frac{\mathrm{d}^i f(x)}{i!\mathrm{d}x^i}\right|_{x=0}x^i=\sum_{i=0}^{\infty}\frac{x^i}{i!} \tag{1}$$

将 $x=1$ 带入（1）式得到

$$\mathrm{e}=\exp(1)=\sum_{i=0}^{\infty}\frac{1}{i!} \tag{2}$$

显然级数以阶乘的速度下降，因此可以通过级数求和来迅速计算 e，截取（2）前 n 项得到

$$\mathrm{e}\approx\sum_{i=0}^{n}\frac{1}{i!} \tag{3}$$

1.2 数值计算方法

（3）式虽然在理论上具有可计算性，然而对于 $n!$ 的求取会对计算带来巨大的困难，实际上，20! 已经超出了 C++ 标准定义的 long long 长整型的表示范围，$\frac{1}{i!}$ 则会更早的由于有效数字的问题带来精度丢失。因此我们需要对（3）式进行计算上的处理。

展开（3）式，得到

$$\mathrm{e}\approx 1+1+\frac{1}{2!}+\frac{1}{3!}+\cdots+\frac{1}{n!} \tag{4}$$

类似秦九韶算法，对计算次序进行优化得到（5）式

$$\mathrm{e}\approx 2+\frac{1}{2}(1+\frac{1}{3}(1+\cdots(\frac{1}{n-1}(1+\frac{1}{n})))) \tag{5}$$

采用逐次递推阶乘的原算法需要消耗 $\mathcal{O}(n)$ 次加法，$\mathcal{O}(n)$ 次除法，而采用了次序优化之后的算法仍需要 $\mathcal{O}(n)$ 次加法，$\mathcal{O}(n)$ 次除法，且常数没有发生变化，然而此时的计算已经从根本上减少了大数加小数的误差，而这种误差在顺行计算原公式时是很明显的。

1.3 方法误差分析

引入 Maclaurin 展式的 Lagrange 余项对方法误差进行分析

$$\exp(x)=\sum_{i=0}^{n}\left.\frac{\mathrm{d}^i f(x)}{i!\mathrm{d}x^i}\right|_{x=0}x^i+\left.\frac{\mathrm{d}^{n+1}f(x)}{(n+1)!\mathrm{d}x^{n+1}}\right|_{x=\xi}\xi^{n+1},\xi\in(0,x) \tag{6}$$

带入相关参数到（6）式中得到

$$\mathrm{e}=\exp(1)=\sum_{i=0}^{n}\frac{1}{i!}+\exp(\xi)\frac{\xi^{n+1}}{(n+1)!},\xi\in(0,1) \tag{7}$$

因此得到误差项为（8）式

$$R(n,\xi)=\exp(\xi)\frac{\xi^{n+1}}{(n+1)!} \tag{8}$$

显然 $R(n,\xi)$ 对于 ξ 而言单调递增，因此得到误差项上界为（9）式

$$R(n)\leq\sup_{\xi\in(0,1)}\left(\exp(\xi)\frac{\xi^{n+1}}{(n+1)!}\right)=\frac{\mathrm{e}}{(n+1)!} \tag{9}$$

从（9）式可以看出，误差随着计算次数 n 的增大而明显增大，从附录中我们知道 double（long double）类型可以得到的 15 位有效数字，又知道 $2<\mathrm{e}<3$，因此我们只需要保证 $R(n)\leq 10^{-16}$ 既可以保证所有的有效数字均被利用，下面求取 n 的范围：

n 范围的求取:

$$\frac{\mathrm{e}}{(n+1)!} \leq 10^{-16} \Rightarrow 16\ln(10) \leq \ln((n+1)!) - 1$$

应用 Stirling 公式近似，方便起见记 $m = n+1$ 得到约束条件变为

$$16\ln(10) + 1 \leq \ln(\sqrt{2\pi m}(\frac{m}{\mathrm{e}})^m) = \ln(\sqrt{2\pi m}) + m\ln(\frac{m}{\mathrm{e}}) = \ln(\sqrt{2\pi m}) + m\ln(m) - m$$

整理约束条件为

$$16\ln(10) + 1 - \frac{\ln(2\pi)}{2} \leq (m+\frac{1}{2})\ln(m) - m$$

求方程数值解可以得到 $m \geq 15 \Rightarrow n \geq 14$ ■

从上述证明可以看出当 $n \geq 14$ 时，方法误差就小于了存储误差下界，因此可以看出误差的收敛性是很好的。实际计算中 $n = 100$ 远远超出了这个精度。

1.4 存储误差分析

这里分析（5）式带来的存储误差问题，我们依赖的数据结构为 C++ 中的 double（long double），他的存储有效位数请见附录。首先我们要给出算法的整体流程如算法 1: 我们对算式中的每一步进行分析，首先初始化 eTaylor

Algorithm 1 Taylor 级数计算 e

Require: 迭代次数 n
Ensure: e 的数值计算值 eTaylor
  初始化 eTaylor $= \frac{1}{n+1}$
  for i 从 n 到 1 **do**
    递增 eTaylor
    eTaylor = eTaylor $/i$
  end for
  递增 eTaylor
  return eTaylor

过程中，eTaylor 的有效数字仍保持 15 位，在第一个循环开始之后，递增 eTaylor 过程中出现第一次精度损失，将之前的从 $\frac{1}{n+1}$ 开始的 15 位有效数字和 1 相加，必然得到从 1 开始的十五位有效数字，亦即精确到 10^{-14}。另外，显然 eTaylor 此时在 1 和 2 中间，下一步除法时，有效数字仍为 15 位。

第二个循环开始之后，首先递增过程依然将精度损失到 10^{-14}，之后每次循环均保持这这个运算有效数字，直到循环结束，此时 eTaylor 的值为 e-1，下一步递增工作仍然保持着上述精度，因此运算精度可以保留到 10^{-14}，即小数点后 14 位上。

上述分析的是单步存储误差，然而由于程序不断迭代，存储误差会被不断增大，因此需要估计迭代传递的存储误差，这里我们记 $\Delta = 5 \times 10^{-15}$ 估计为每步的有效数字引起的误差（上面已经证明运算精度可以达到 10^{-14}）

主要的迭代过程是一个反向迭代，可以表示为 (10) 式

$$y_{n-1} = \frac{y_n + 1}{n} \tag{10}$$

取存储误差和传递误差分析得到误差传递为（11) 所示

$$\delta_{y_{n-1}} = \frac{\delta_{y_n}}{n} + \Delta \tag{11}$$

处理这个式子立刻得到可递推的（12）式

$$\frac{\delta_{y_{n-1}}}{(n-1)!} = \frac{\delta_{y_n}}{n!} + \frac{\Delta}{(n-1)!} \tag{12}$$

反向计算等差数列可以得到确定的传递误差精度为（13）式

$$\delta_{y_{n-1}} = \Delta \sum_{i=0}^{n} \frac{1}{i!} \approx \Delta \mathrm{e} \tag{13}$$

进一步可以说明传递误差精度是受限的，因此上面分析的单步误差精度是有意义的。

1.5 计算结论分析和复杂度分析

综合前两节的叙述，采用 $n = 100$ 的设定之后，使用 Taylor 公式对 e 的计算可以达到小数点后 15 位，实际编程操作后确认了这个结论，我们得到的 eTaylor 和实际的 e 分别为（14）式所示。程序耗时 $1.2\mu s$，测试 CPU 主频 2GHz

$$\begin{aligned} \mathrm{eTaylor} &= 2.71828\ 18284\ 59045 \\ \mathrm{e} &\approx 2.71828\ 18284\ 59045\ 23536\ 02874 \end{aligned} \tag{14}$$

程序属于线性复杂度（按照加法和乘法次数计算），可推广性很好。且收敛速度为 $\mathcal{O}(n!)$，不需要很大的 n 就可以让计算收敛。同时，Taylor 方法的另外一个优点是，计算精度随着 n 的增大而单调递增，由于较大的 n 必然包括较小的 n 的计算项，提高 n 的值只可能是的后面的项受到计算精度的影响而消失，不影响前面的项。

2 数值求解微分方程得到 e 的数值解

2.1 原理分析

考虑含边界条件的微分方程（15）式的解析解：

$$\frac{\mathrm{d}y}{\mathrm{d}x} = y;\ y(0) = 1 \tag{15}$$

自然得到方程的解析解为 $y = \exp(x)$，计算 $y(1)$ 即可得到 e 的值，因此只需要设计计算（15）式的方法

2.2 数值计算方法

采用四阶 Runge-Kutta 方法对微分方程（15）式进行求取，其常用的基本迭代方式为（16）式所示，其中 h 为分割的小区间长度。

$$\begin{cases} y_{n+1} = y_n + \dfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ \quad K_1 = y_n \\ \quad K_2 = y_n + \dfrac{h}{2}K_1 \\ \quad K_3 = y_n + \dfrac{h}{2}K_2 \\ \quad K_4 = y_n + hK_3 \end{cases} \tag{16}$$

将区间从 0 到 1 分割成 n 段，按照递推进行计算即可得到 e 的值

2.3 方法误差分析

参考数值分析教材可以知道，四阶 Runge-Kutta 方法的截断误差为 $\mathcal{O}(h^5)$，因此对整个区间上的截断误差为 $\mathcal{O}(h^4)$，当 $n > 10^4$ 即 $h < 10^{-4}$ 时，可以保留到 16 位有效数字，此时方法误差上限小于存储误差下限

2.4 存储误差分析

由递推式（16）式知道，y, K 这些参数受到加法的影响，其有效数字被确定在了 15 位，可以精确到小数点后 14 位，因此这个算法的计算精度应为小数点后 14 位。当然，如果 h 过小，h 的作用就会被明显的削弱，由于 $1 \leq y \leq \mathrm{e}$，而可以看出在 K_4 项上出现了 h^3 项，转到 y 的递推式上就出现了 h^4 项，因此我们必须保证 h^4 在有效数字范围内，同时从上节方法误差分析中我们可以看出 $h < 10^{-4}$ 时才能保证方法误差小于存储误差，因此作为一种折中，实验操作中 $n = 10^4, h < 10^{-4}$

上述分析的是单步存储误差，然而由于程序不断迭代，存储误差会被不断增大，因此需要估计迭代传递的存储误差，这里我们记 $\Delta = 5 \times 10^{-15}$ 估计为每步的有效数字引起的误差（上面已经证明运算精度可以达到 10^{-14}）

我们按照（16）式定义的方式来计算由于迭代造成的存储误差。首先误差分析得到（17）式

$$\begin{cases} \delta_{y_{n+1}} = \delta_{y_n} + \dfrac{h}{6}(\delta_{K_1} + 2\delta_{K_2} + 2\delta_{K_3} + \delta_{K_4}) + \Delta \\ \delta_{K_1} = \delta_{y_n} \ \text{（由于没有计算过程，不含 } \Delta \text{ 项）} \\ \delta_{K_2} = \delta_{y_n} + \dfrac{h}{2}\delta_{K_1} + \Delta \\ \delta_{K_3} = \delta_{y_n} + \dfrac{h}{2}\delta_{K_2} + \Delta \\ \delta_{K_4} = \delta_{y_n} + h\delta_{K_3} + \Delta \end{cases} \tag{17}$$

将（17）式带入分析得到（18）式

$$\begin{cases} \delta_{K_2} = (1 + \dfrac{h}{2})\delta_{y_n} + \Delta \\ \delta_{K_3} = (1 + \dfrac{h}{2} + \dfrac{h^2}{4})\delta_{y_n} + (1 + \dfrac{h}{2})\Delta \\ \delta_{K_4} = (1 + h + \dfrac{h^2}{2} + \dfrac{h^3}{4})\delta_{y_n} + (1 + h + \dfrac{h^2}{2})\Delta \\ \delta_{y_{n+1}} = \delta_{y_n} + \dfrac{h}{6}((5 + 3h + h^2 + \dfrac{h^3}{4})\delta_{y_n} + (5 + 2h + \dfrac{h^2}{2})\Delta) + \Delta \leq (1 + h)(\delta_{y_n} + \Delta) \end{cases} \tag{18}$$

式（18）中最后一步进行了一次近似，方便之后的求取。配系数得到（19）式

$$\begin{gathered} \delta_{y_{n+1}} = (1 + h)\delta_{y_n} + (1 + h)\frac{(1 + h) - 1}{h}\Delta \\ \delta_{y_{n+1}} + \frac{1 + h}{h}\Delta = (1 + h)(\delta_{y_n} + \frac{1 + h}{h}\Delta) \end{gathered} \tag{19}$$

利用等比数列相关知识立刻得到最终的误差分析式为（2.4）式。

$$\delta_{y_n} = (1 + h)^n(\frac{1 + 2h}{h}\Delta) - \frac{1 + h}{h}\Delta \approx \frac{\mathrm{e} + 2h\mathrm{e} - 1 - h}{h}\Delta \tag{20}$$

我们可以明确的发现这个式（）是发散的。这说明 h 的值不能取得太小。当然式（）只是一个近似式。通过上文关于主项的分析我们大致可以判断出 h 取值为多少时可以保证上式的相对稳定。综合所有分析可以取 $h = 10^{-4}$

2.5 计算结论分析和复杂度分析

综合前两节的叙述，采用 $n = 10^4$ 的设定之后，使用微分方程对 e 的计算可以达到小数点后 14 位，实际编程操作后确认了这个结论，我们得到的 eDEQN 和实际的 e 分别为（21）式所示。程序耗时 $243.3\mu s$，测试 CPU 主频 2GHz

$$\begin{gathered} \mathrm{eDEQN} = 2.71828\ 18284\ 59040 \\ \mathrm{e} \approx 2.71828\ 18284\ 59045\ 23536\ 02874 \end{gathered} \tag{21}$$

程序属于线性复杂度（按照加法和乘法次数计算），可推广性较好，相比于 Taylor 方法的 $\mathcal{O}(n!)$ 的收敛速度，这种方式的收敛速度是 $\mathcal{O}(n^4)$，因此得到较好解的速度比上种方法慢得多。同时，相比于 Taylor 展开方法，一味提高 n 的值可能使计算过程中 h 项由于舍入误差消失，因此这种方法不如上面 Taylor 展开方法。

3 数值积分求解 e 的数值解

3.1 原理分析

考虑积分方程（22）式的根为 e，因此可以考虑通过求解此积分方程来计算 e 的数值解。

$$f(x)=\int_1^x \frac{1}{t}\mathrm{d}t=1 \tag{22}$$

3.2 数值计算方法

首先我们对 $f(x)$ 以 h 为间隔进行采样得到一组 $f(x_k)$，显然他们之间的递推关系为（23）式

$$\begin{cases} f(1)=0 \\ f(x_k)=f(x_{k-1})+\int_{x_{k-1}}^{x_k}\frac{1}{t}\mathrm{d}t \end{cases} \tag{23}$$

利用数值积分方法求取每一个小区间上的积分值，为了提高精度，我们可以将 h 取的充分小，并利用梯形公式求解。这样得到了一组 $f(x_k)$。

通过寻找 $f(x_k)-1$ 的过零点位置 x_z 进行线性插值（弦截法求根）可以得到零点 x^* 的位置为（24）式

$$\begin{cases} f(x_z)>1 \\ f(x_{z-1})<1 \\ x^*=x_{z-1}+\frac{1-f(x_{z-1})}{f(x_z)-f(x_{z-1})} \end{cases} \tag{24}$$

积分公式采用梯形公式，因为这不需要增加内部节点的数量，相对速度也比较快，实际上，由于 h 已经取的很小，Cotos，Simpson，梯形三种积分公式的差别不大。

3.3 方法误差分析

方法误差包括两个方面，一方面是积分公式带来的误差，一方面是插值带来的误差。积分公式如果采用梯形公式，则在分割节点的控制之下积分公式表现为复合梯形公式，其余项可以表征为（25）式，其中积分函数 $f(t)=1/t$

$$|R_n(f)|=-\frac{\mathrm{e}-1}{12}h^2f''(\xi)\Big|_{\xi\in(0,\mathrm{e}+h)}<\frac{\mathrm{e}-1}{6}h^2 \tag{25}$$

如果采用复合 Simpson 公式，则余项可以表征为（26）式，其中积分函数 $f(t)=1/t$

$$|R_n(f)|=-\frac{\mathrm{e}-1}{180}\frac{h}{2}^4f^{(4)}(\xi)\Big|_{\xi\in(0,\mathrm{e}+h)}<\frac{\mathrm{e}-1}{120}h^4 \tag{26}$$

可以看到复合 Simpson 公式带来更小的误差

另一部分的误差来源于插值求根公式，线性插值误差表征为（27）式，其中插值原函数 $f(x)=\ln(x)$

$$R_n(x)=|\frac{f''(\xi)}{2}(x-x_1)(x-x_1)|\Big|_{\xi\in(\mathrm{e}-h,\mathrm{e}+h)}\leq\frac{h^2}{8\mathrm{e}^2} \tag{27}$$

综合两方面的误差，我们可以认为误差在 $\mathcal{O}(h^2)$ 量级上，当 $n=1/h=10^8$ 时，方法误差上限小于舍入误差下限。

Algorithm 2 数值积分计算 e

Require: 分割间隔 N

Ensure: e 的数值计算值 eInt

初始化 $iter = 1, h = \frac{1}{N}, p = 0, n = 0$

为 p 加下一个积分区间段的值

while $p < 1$ **do**

$n = p$;

$iter \mathrel{+}= h$;

为 p 加下一个积分区间段的值

end while

计算 eInt$= iter + h\frac{1-n}{p-n}$

return eInt

3.4 存储误差分析

首先写出我们的算法过程如算法 2所示 我们可以看出，在计算 n 和 p 的过程中，收到加法的限制，p 最终可以保证精确到小数点后 14 位，n 最终可以保证精确到小数点后 15 位，精度的损失出现在线性插值过程中。$\frac{1-n}{p-n}$ 将出现位数损失，根据程序计算流程可以得到 $\ln(\mathrm{e}-h) < n < 1 < p < \ln(\mathrm{e}+h)$。因此得到

$$\begin{cases} -\frac{h}{\mathrm{e}} < \ln(\mathrm{e}-h) - 1 < n - 1 < 0 \Rightarrow 1 - n < \frac{h}{\mathrm{e}} \\ 0 < p - n < 2\frac{h}{\mathrm{e}} \end{cases} \tag{28}$$

$1-n$ 和 $p-n$ 均只能在（28）式确定的值，但只能保留到小数点后 14 位。这里做一个近似的估计，我们设 $N = \frac{1}{h} = 10^m$，则可以看出有效数字缩减为 $14-m$ 位

但是受到前面 iter 项的限制，存储精度可以保留到小数点后 14 位。

下面讨论积分公式中的存储误差限制，积分原函数为 $f(t) = 1/t$，要求 t 变化时，变化的主项必须保证，也就是说 $f(t+h) - f(t)$ 的数值必须非零，否则会产生明显的存储误差。利用 Taylor 公式展开得到 $f(t+h) - f(t) = -\frac{h}{t^2}$，因此我们需要要求 $\frac{h}{t} > 10^{-14}$，于是得到 $h > 10^{-13}$

下面分析迭代误差，由于程序计算过程中对于积分累加过程中将误差直接进行了累加，因此从过程中得到的累加误差高达 $n\Delta$（记 $\Delta = 5 \times 10^{-15}$ 估计为每步的有效数字引起的误差（上面已经证明运算精度可以达到 10^{-14}））。因此积分方法的误差发散的也最严重。结合上面的方法误差可以达到 h^2 的分析，下面求解导致误差发散不会淹没方法的最小 h：由于误差传递从 10^{-15} 量级上损失 h 个有效数字，而方法误差达到 h^2 量级，因此得到两个方法的交汇在 $2\left|\lg h\right| = 15 - \left|\lg h\right|$ 上，得到 $h = 10^{-5}$，对应方法误差和存储误差均在 10^{-10} 级别上，因此预计可以达到 10^{-9} 级别的精确度。和前两个相比相差很多。

3.5 计算结论分析和复杂度分析

由于涉及的分割区间大小 h 过小，因此在计算过程中根据系统实现的不同也会出现存储误差，经过测试，当 $n = \frac{1}{h} = 10^7$ 时出现明显的存储误差，主要的存储误差出现在了对 $h = 1/n$ 的存储，大数和小数的相加等问题，另外，这个算法的计算复杂度是 $\mathcal{O}(N) = \mathcal{O}(\frac{1}{h})$ 的，当 h 选取的过小时，计算时间出现明显的上升。实际测试中选取了 $N = 10^5$，可以精确到小数点后 9 位。这一点和之前理论分析得到的估计值一致。

实际编程操作后确认了这个结论，我们得到的 eInt 和实际的 e 分别为（29）式所示。可以精确到小数点后 10 位程序耗时 13ms，测试 CPU 主频 2GHz

$$\begin{aligned} \mathrm{eDEQN} &= 2.71828\ 18284\ 60041 \\ \mathrm{e} &\approx 2.71828\ 18284\ 59045\ 23536\ 02874 \end{aligned} \tag{29}$$

程序属于线性复杂度（按照加法和乘法次数计算），相比于 Taylor 方法的 $\mathcal{O}(n!)$ 的收敛速度，这种方式的收敛速度是 $\mathcal{O}(n^2)$，因此得到较好解的速度比上种方法慢得多。同时，相比于 Taylor 展开方法，一味提高 n 的值可能使计算过程中 h 项由于舍入误差消失，因此这种方法不如上面两种方法。

4 小结

从上述讨论中我们可以看出，Taylor 展开方法相对于后面两种方法有着更好的收敛速度，更低的计算代价和更小的计算误差，因此在实际计算过程中更推荐使用 Taylor 展开方法计算 e 的数值

附录

C++ 标准中 double 的存储方式和有效值

C++ 中的基本数据类型 double（又称 long double）是符合 IEEE 754 浮点数运算标准下的 64 位浮点数基本数据结构。按照 IEEE 754 的要求，其包括 1 位二进制符号位，11 位二进制指数位和 52 位二进制尾数位，其表示方法是二进制科学计数法。因此可以认为 double 的二进制有效数字为 52 位，转 10 进制得到 $\log_{10} 52 \approx 15.6$。因此我们可以认为一个 double（long double）可以提供 15 位有效数字，实际的测试中可以验证这个有效数字位数是正确的。