数值分析第一章上机

By 211870125 陈睿硕

§1 问题

Q1 求计算机的规范化浮点数的上溢值 (OFL)、下溢值 (UFL) 和计算机的机器精度 (ε_{mach})。

Q2 编写并测试子程序，计算 $y = x - \sin x$，使得有效位的丢失最多 1 位。

Q3 计算 $y_n = \int_0^1 x^n e^x dx \quad (n \geq 0)$。

Q4 考虑由

$$\begin{cases} x_0 = 1 \quad x_1 = \dfrac{1}{3} \\ x_{n+1} = \dfrac{13}{3}x_n - \dfrac{4}{3}x_{n-1} \quad (n \geq 1) \end{cases}$$

归纳定义的实数序列。将初值改为 $x_0 = 1$ 和 $x_1 = 4$ 数值稳定吗？

§2 算法思路

I 对Q1的解答

问题即求解 $-L$（即单精度数最小指数）、U（即单精度数最大指数）、t（即单精度数精度）。这是因为利用公式：

$$OFL = 2^U(2 - 2^{-(t-1)}), \tag{1}$$

$$UFL = 2^{-L}, \tag{2}$$

$$\varepsilon_{mach} = 2^{-t}, \tag{3}$$

可以求得问题之解。

下面我们将分四步求解上溢值，下溢值和机器精度。

i 求解 $t + L$

我们将 $a = 1$ 不断进行"÷2" 操作，直到其变为 0，记录下操作的次数 $count$。

在 a 变为 0 的前一刻，a 的数值应该是单精度浮点数的最小正值，即 $2^{-L-(t-1)}$。这是因为非规格数的指数始终默认为 $-L$。从 $a = 1$ 到 $a = 2^{-L-(t-1)}$ 共进行了 $L + t - 1$ 步，故最后当 a 恰变为 0 时，共进行了 $t + L$ 步。则有:

$$t + L = count = 150, \tag{4}$$

ii 求解 t

我们令 $c = 3, d = 1$, 然后不断对两者进行 "$\times 2 + 1$" 操作，直到式 $c - 1 = 2b$ 不成立，记录下操作的次数，其即为 $t - 1$。

这是因为规格数的有效数字不能超过 t 位，一旦超过 t 位便会产生舍入误差。易见每次操作都将 c、d 的有效位数增多一位，当式 $c - 1 = 2b$ 第一次不成立时，c 的有效数字恰超过 t 位，d 的有效数字恰为 t 位，故实际上有：

$$d = (\underbrace{11\cdots\cdots1}_{t\text{个}})_2 = 2^t - 1, \tag{5}$$

此时也正好进行了 $t - 1$ 次操作，如此可求得 $t = 24$。

iii 求解上溢值

结合式 (5) 改写式 (1) 为：

$$OFL = 2^{U-t+1}(2^t - 1) = 2^{U-t+1}d$$

故类似于 II 中的方法，我们不断将 d 进行 "×2" 操作，直至发生舍入。记录下舍入前一刻的 d 的值，即为上溢值 OFL。求得上溢值 $OFL \approx 3.40282 \times 10^{38}$。

iv 求解下溢值及机器精度

由 $t = 24$ 及式 (4) 知 $L = 150 - t = 126$, 利用式 (2) 计算得 $UFL \approx 1.17549 \times 10^{-38}$。

由式 (3) 计算得 $\varepsilon_{mach} \approx 5.96046 \times 10^{-8}$

II 对Q2的解答

根据精度损失定理，我们可知当 $1 - \frac{\sin x}{x} \geq \frac{1}{2}$ 时，精度损失不超过一位。故 $|x| \geq 2$ 时，我们调用 python 中 math 库的 sin 函数直接计算即可。

$|x| \leq 2$ 时，我们对 $x - \sin x$ 进行泰勒展开：

$$x - \sin(x) = \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} - \frac{x^9}{9!} + \frac{x^{11}}{11!} - \cdots$$

为了保证使用泰勒级数截断近似后不会丢失超过一位，我们研究 $x - \sin x$ 泰勒展开的拉格朗日余项：

$$R_n(x) = \frac{(-1)^{n+1}\cos(\xi)}{(n+1)!}x^{n+1}$$

我们希望 $|R_n(x)| \leq 2^{-24}$，事实上当 $n = 15$ 时，有 $|R_n(x)| \leq \frac{2^{15}}{15!} < 2^{-24}$。故 $|x| \geq 2$ 时，我们可以按照如下公式计算：

$$x - \sin(x) \approx \frac{x^3}{3!} - \frac{x^5}{5!} + \frac{x^7}{7!} - \frac{x^9}{9!} + \frac{x^{11}}{11!} - \frac{x^{13}}{13!}$$

III 对Q3的解答

易见 $y_0 = e - 1$，故计算 y_0 时会发生舍入误差，若按照递推公式直接迭代计算 y_n，这个舍入误差和每次迭代时计算 e 的舍入误差因多次与远大于一的数相乘而变得十分巨大，导致算法不稳定。

通过分部积分，我们可以得到 y_n 的递推公式：

$$y_{n+1} = e - (n+1)y_n$$

利用上述递推公式我们可以理论上地算出：

$$\int_0^1 x^n e^x dx = (-1)^n n!(e-1) + (-1)^n n! e \sum_{k=1}^{n} (-1)^k \frac{1}{k!}$$

注意到

$$e^{-x} - 1 = \sum_{k=1}^{\infty} \frac{(-x)^k}{k!} = \sum_{k=1}^{n} \frac{(-x)^k}{k!} + R_n(x)$$

其中，$R_n(x) = \frac{e^{-cx}}{(n+1)!}x^{n+1}, c \in (0,1)$。

故有

$$\left|\int_0^1 x^n e^x dx\right| = \left|(-1)^n n!(e-1) + (-1)^n n! e(e^{-1} - 1 - R_n(1))\right| = \frac{e^{1-c}}{(n+1)} \leq \frac{e}{n+1}$$

故当 n 足够大时（这里我们取界限为 2^{24}），我们可以认为 $y_n = 0$，并将递推公式改写为：

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

如此由后向前逆向递推，可以使误差不断乘远小于 1 的数从而阶乘级缩小。

IV 对Q4的解答

使用递推公式及初值可以求得 $x_n = \frac{1}{3^n}$（$x_0 = 1 \;\; x_1 = \frac{1}{3}$ 时），$x_n = 4^n$（$x_0 = 1 \;\; x_1 = 4$ 时）。使用 python 程序直接计算即可，但要注意的是应计算 $\frac{1}{3^n}$ 而非 $\left(\frac{1}{3}\right)^n$，后者会将舍入误差幂次计算，从而增大误差 (见附录代码 4中的 **test.py**)。

§3 结果分析

I 对Q1的分析

这个方法是通过找出 t、U、L 的值利用既知公式计算上溢值、下溢值及机器精度的，故和理论分析的上述三值完全相等。利用 C++ 的 char 指针打印出所算出的三值的单精度表示，也和我们理论分析的形式无异，故正确性是肯定的。

II 对Q2的分析

我们将算法在 python 中实现后，在 $x = 10^{-t}, t = 0, 1, 2, \cdots, 10$ 时直接计算和用泰勒展开计算 $x - \sin x$，得到如下结果：

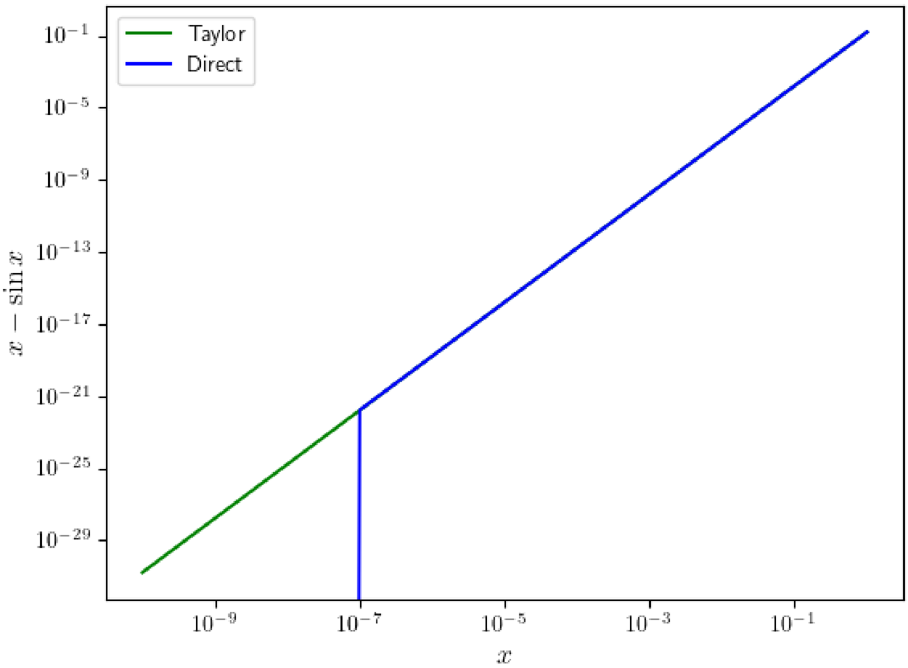


图 1: 直接计算与用泰勒展开计算结果

可以看到两种方法的计算结果在 10^{-7} 量级以上极其相近，这也就说明了泰勒展开的近似较为准确。但直接计算的方法在 10^{-7} 量级以下发生严重偏移，这正是相减相消后精度不够导致的。

并且我们知道 $x - \sin x \sim O(x^3)(x \to 0)$, 这也和图 1中图像的线性以及直线的斜率相符合。

III 对Q3的分析

考虑到 y_n 十分接近 0，计算的近似值可能会在 0 的上下波动，我们对两种方法计算出的 $|y_i|, i = 1, 2, \cdots, 100$ 作图 2。

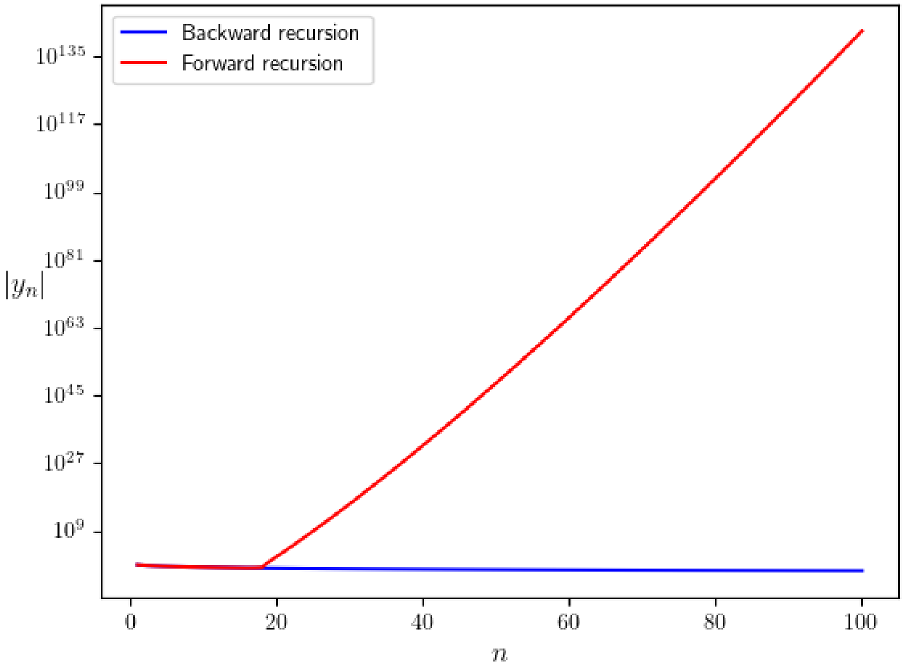
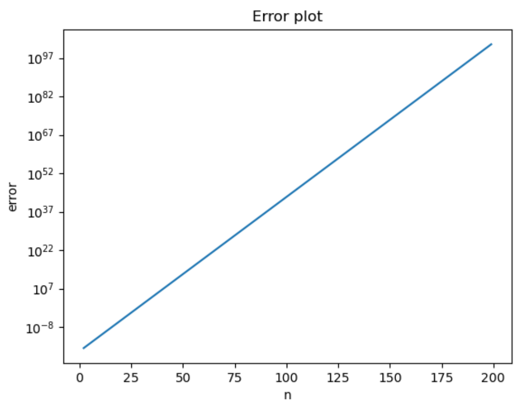


图 2: 两种递推法算出的 $|y_n|$

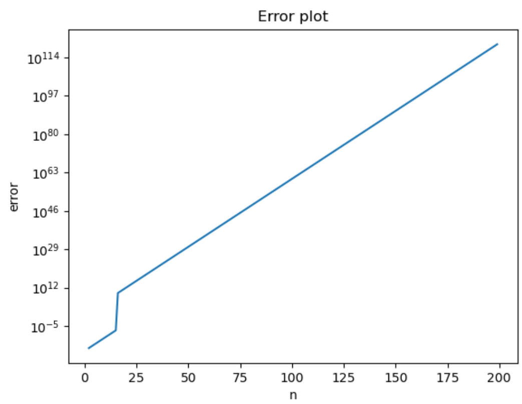
可以看到正向递推法求得的 $|y_n|$ 在 n 不较小的情况下表现出了指数级增长，这显然不符合 $|y_n|$ 的单调变化。而逆向递推法求得的 $|y_n|$ 始终平稳地减小，这是符合实际的。

IV 对Q4的分析

利用递推公式迭代计算 $x_n(n \geq 2)$，并计算其与单精度表示的 $\frac{1}{3^n}$ 之差，得到图 3(a)。



(a) x_n 与 $\frac{1}{3^n}$ 之间的误差



(b) x_n 与 4^n 之间的误差

图 3: Q4 误差图

可以看到随着 n 的增大，误差呈指数型增长，在 $n = 200$ 是更是达到了惊人的 10^{97} 量级，说明递推地正向计算是不稳定的。我们可以发现，误差主要来自于非 3 倍数的整数除以 3 发生的舍入误差在数列的正向递推中不断累积（$\frac{1}{3^n}$ 计算时也发生了舍入误差，但与数列递推产生的误差相比可以忽略

不计），我们记

$$x_n = \frac{1}{3^n}(1+\varepsilon_n) \tag{$*$}$$

记计算 $\frac{1}{3}$ 时产生的舍入误差为 ε。易见 $\varepsilon_0 = 0$，$\varepsilon_1 = \varepsilon$。将 $(*)$ 式代入递推公式得（事实上递推公式中计算 $\frac{13}{3}$ 与 $\frac{4}{3}$ 也会发生舍入误差，但当 n 不过小时，此舍入误差的量级远远小于 ε_n，故在下面的推导中忽略不计）：

$$\varepsilon_{n+1} = 13\varepsilon_n - 12\varepsilon_{n-1} (n \geq 1)$$

解得 $\varepsilon_n = \frac{1}{11}(12^n - 1)\varepsilon$，故可见实际误差（即 x_n 与 $\frac{1}{3^n}$ 之间的误差）满足：

$$\widehat{\varepsilon_n} = \frac{1}{11}\left[4^n - \left(\frac{1}{3}\right)^n\right]\varepsilon \approx \frac{4^n}{11}\varepsilon$$

$\widehat{\varepsilon_n}$ 确实如图 3(a)关于 n 指数增长。并且结合 Week 1 的上机作业可知，$\varepsilon_{mach} \approx 5.96046 \times 10^{-8}$，$\frac{1}{3}$ 的舍入误差 ε 量级上应比 ε_{mach} 小，于是 $\varepsilon_2 \approx \frac{16}{11}\varepsilon$ 的量级小于 10^{-8}，与图 3(a)的起始点位置相符合。

类似地，当 $x_0 = 1$，$x_1 = 4$ 时，得到误差图图 3(b)。

可以看到误差也是呈指数型增长。事实上，若类似上文的定义，应有 $\varepsilon_0 = \varepsilon_1 = 0$，那为何还会发生舍入误差的累积呢？这是因为其实由于计算 $\frac{13}{3}$ 与 $\frac{4}{3}$ 时发生的舍入误差，$\varepsilon_2 \neq 0$，故同前推导，误差依然会呈指数增长。但此处与上例不同的是，折线在 $n = 15$ 处发生了一次“突变”，我认为这是 $\varepsilon_i(i = 2, 3 \cdots, 15)$ 不足以远大于计算 $\frac{13}{3}$ 与 $\frac{4}{3}$ 时发生的舍入误差导致的。

§4 附录：程序代码

代码 1: First Week.cpp

```cpp
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    float a=1,b=1;
    int t_plus_L=0;
    while(a!=0)
    {
        b=a;
        a/=2;
        t_plus_L++;
    }
    float c=3,d=1,t=1;
    while(c-1==2*d)
    {
        d=2*d+1;
        c=2*c+1;
        t++;
    }
```

```
    float u=2*d;
    while(u/2==d)
    {
        u*=2;d*=2;
    }
    cout<<"上溢值为: "<<d<<endl<<"下溢值为: "<<pow(2,t-t_plus_L)<<endl
    <<"机器精度为: "<<pow(2,-t)<<endl;
    return 0;
}
```

代码 2: Second Week 1.py

```
import math
import matplotlib.pyplot as plt
plt.rcParams['font.sans-serif'] = ['Microsoft YaHei']
def calc_y(x):
    if abs(x)>=2:
        y=x-math.sin(x)
        return y
    else:
        rad = x
        return rad**3/6 - rad**5/120 + rad**7/5040 - rad**9/362880 + rad
            **11/39916800 -rad**13/6227020800
def precise(x):
    return x-math.sin(x)
a = [10**i for i in range(-10,1)][::-1]
b = [calc_y(n) for n in a]
c = [precise(n) for n in a]
plt.plot(a,b,label='使用泰勒级数计算',color='green')
plt.plot(a,c,label='直接计算',color='blue')
plt.xlabel('x')
plt.xscale('log')
plt.yscale('log')
plt.legend()
plt.show()
```

代码 3: Second Week 2.py

```
import numpy as np
import math
import matplotlib.pyplot as plt
plt.rc('text',usetex=True)
y1=np.zeros(2**24+1)
```

```
n=2
for i in list(range(0,2**24))[::-1]:
    y1[i]=(math.e-y1[i+1])/(i+1)
if n>2**24:
    print("y_n=",0)
else:
    print("y_n={:.23f}".format(y1[n]))
y2=np.zeros(101)
y2[0]=math.e-1
for j in range(0,100):
    y2[j+1]=math.e-(j+1)*y2[j]
y2=[abs(k) for k in y2]
fig,ax=plt.subplots()
ax.set_xlabel(r'$n$',fontsize=13)
ax.set_ylabel(r'$\left|y_{n}\right|$',rotation=0,fontsize=13)
ax.plot(range(1,101),y1[1:101],label='Backward recursion',color='blue')
ax.plot(range(1,101),y2[1:101],label='Forward recursion',color='red')
ax.legend()
plt.yscale('log')
plt.show()
```

代码 4: Second Week 3

```
##Second_Week_3A.py
import matplotlib.pyplot as plt
import numpy as np
x=np.zeros(200,dtype=float)
x[0]=1
x[1]=1/3
ep=np.zeros(200,dtype=float)
for i in range(2,200):
    x[i]=x[i-1]*13/3-x[i-2]*4/3
    ep[i]=np.abs(1/np.power(3,i)-x[i])
plt.plot(np.arange(2,200), ep[2:200])
plt.xlabel('n')
plt.ylabel('error')
plt.yscale('log')
plt.title('Error plot')
plt.show()

##Second_Week_3B.py
import matplotlib.pyplot as plt
```

```
import numpy as np
x=np.zeros(200,dtype=float)
x[0]=1
x[1]=4
ep=np.zeros(200,dtype=float)
for i in range(2,200):
x[i]=x[i-1]*13/3-x[i-2]*4/3
ep[i]=np.abs(np.power(4,i)-x[i])
plt.plot(np.arange(2,200), ep[2:200])
plt.xlabel('n')
plt.ylabel('error')
plt.yscale('log')
plt.title('Error plot')
plt.show()


##test.py
a=float(1/3)**200
b=float(1/3**200)
print(a==b)
```