数值分析第 2 次上机作业

学号：221840189，姓名：王晨光

§1 问题一

1.1 问题

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

1.2 算法思路

由精度丢失定理，$1-\frac{sinx}{x} \geqslant \frac{1}{2}$ 时，$x-sinx$ 至多丢失 1 个精度，可以直接代入计算. 当 $1-\frac{sinx}{x} < \frac{1}{2}$ 时，考虑 $x - sinx$ 的 Taylor 级数展开：

$$x - sinx = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{x^{2n+1}}{(2n+1)!}.$$

令 $a_n = \frac{x^{2n+1}}{(2n+1)!}$ 往证上述级数中每两项在进行计算时的有效位丢失至多为 1 位：

$$\begin{aligned} 1 - \frac{a_{n+1}}{a_n} &= \frac{x^2}{(2n+1)(2n+3)} \\ &\geqslant 1 - \frac{1.9^2}{3 \times 6} \\ &\geqslant 0.79 \\ &\geqslant \frac{1}{2} \end{aligned}$$

故当 $1 - \frac{sinx}{x} \leqslant \frac{1}{2}$ 时，通过 Taylor 级数计算，有效位的丢失也至多为 1 位.

Algorithm 1 计算 $x - sinx$

Require: 实数 x.

Ensure: $y = x - sinx$ 的值.

function $y(x)$
  $y \leftarrow 0, n \leftarrow 1$
  if $1 - \frac{sinx}{x} \geqslant \frac{1}{2}$ **then**
    $y \leftarrow x - sinx$
  else
    $a_n \leftarrow (-1)^{n+1}\frac{x^{2n+1}}{(2n+1)!}$
    while $|a_n| \geqslant 10^{-16}$ **do**
      $y \leftarrow y + a_n, n \leftarrow n + 1$
      $a_n \leftarrow (-1)^{n+1}\frac{x^{2n+1}}{(2n+1)!}$
    end while
  end if
end function

1.3 结果分析

由上述证明可知，该算法可以使得有效位的丢失限定在一位. 以下是部分运算结果：

x	y
-0.0001	-1.666666666666667e-13
-0.001	-1.6666666666666669e-10
-0.01	-1.6666583333333337e-07
-0.1	-0.0001665833531718475
-1.0	-0.1585290151921035
-10.0	-10.54402111088937
0.0001	1.666666666666667e-13
0.001	1.6666666666666669e-10
0.01	1.6666583333333337e-07
0.1	0.0001665833531718475
1.0	0.1585290151921035
10.0	10.54402111088937

表 1: $y = x - sinx$ 的部分计算结果

§2 问题二

2.1 问题

计算

$$y = \int_0^1 x^n e^x \, dx \quad (n \geqslant 0).$$

由分部积分法得 $y_{n+1} = e - (n+1)y_n$，数值不稳定.

2.2 算法分析

计算得 $y_0 = \int_0^1 e^x \, dx = e - 1 \approx 1.718 \cdots \rightarrowtail \hat{y_0}$. $\hat{y_0}$ 无法避免舍入误差，且由于递推公式 $y_{n+1} = e - (n+1)y_n$ 中 y_n 前的系数 $n+1$ 大于 1，并且会随着 n 线性增长，那么初始值 $\hat{y_0}$ 的误差会被阶乘级别不断放大：

$$\epsilon_n = y_n - \hat{y_n} = n \cdot (y_{n-1} - \hat{y_{n-1}}) = \cdots = n! \cdot (y_0 - \hat{y_0}) = n! \cdot \epsilon_0.$$

由此可见该算法的误差会爆炸式地增长，算法极其不稳定.

2.3 结果分析

n	y_n	$\hat{y_n}$
0	1.7182818284590453	1.7182818284590453
2	0.7182818284590453	0.7182818284590455
4	0.46453645613140715	0.46453645613141115
6	0.34468454164698736	0.34468454164710893
8	0.27436153301797617	0.2743615330247846
10	0.22800151548644187	0.22800151609920905
12	0.19509993116082067	0.19510001204609795
14	0.17052370130176747	0.17053842242224038
16	0.1514608855385012	0.15499395445201403
18	0.13623989097759065	1.2173589785125256
19	0.1297238998848238	-20.41153876327894
20	0.12380383076256998	410.94905709403787

表 2: 积分递推式所得估计值 $\hat{y}$ 与实际值 y 的比较

实际值可以通过 Python 的 Scipy 库中的 `integrate.quad` 函数计算得到.

从表格可以看出，误差被不断放大，以至在 $n = 16$ 之后呈现爆炸式增长，甚至会交替地出现负值，故该算法极不稳定.

§3 问题三

3.1 问题

考虑由

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

定义的实数序列，算法不稳定.

将初值改为 $x_0 = 1, x_1 = 4$ 数值稳定吗？

3.2 算法分析

通过特征根法处理 $x_{n+1} = \dfrac{13}{3}x_n - \dfrac{4}{3}x_{n-1}$，求得通项公式 $x_n = A \cdot \dfrac{1}{3^{n-1}} + B \cdot 4^{n-1}$，分别代入 $x_0 = 1, x_1 = 1/3$ 与 $x_0 = 1, x_1 = 4$ 得到序列 1 通项公式为 $x_n = \dfrac{1}{3^n}$，序列 2 通项公式为 $x_n = 4^n$，由此可以直接得到递推公式的各项实际值，并与计算机通过递推公式计算得到的序列值做比较，通过实验验证算法的数值稳定性.

由初值 $x_0 = 1, x_1 = 1/3$ 给出的实数序列算法不稳定，原因在于初值 $x_1 = 1/3$ 在存储中会出现舍入误差，在计算 x_2 时，x_1 造成的误差会被系数 $\dfrac{13}{3}(\dfrac{13}{3} > 1)$ 扩大；计算 x_3 时，误差来源于两项相减，但 $\dfrac{13}{3} \cdot \dfrac{13}{3} \gg \dfrac{4}{3}$，误差无法相互抵消，如此递推，误差无法收敛，会不断放大，最终导致算法数值不稳定.

由初值 $x_0 = 1, x_1 = 4$ 给出的实数序列算法理论上稳定，因为初值不存在舍入误差，不会在计算过程中被放大，误差来源为每次计算时系数的舍入误差产生的，但系数的误差相当小且不会被放大，故算法应该是数值稳定的.

3.3 结果分析

n	x_n	$\hat{x_n}$
0	1.0	1.0
3	0.03703703703703703	0.03703703703703626
6	0.0013717421124828527	0.0013717421124321456
9	5.0805263425290837e-05	5.0805260179967644e-05
12	1.8816764231589195e-06	1.8814687224716613e-06
15	6.969171937625627e-08	5.63988753916179e-08
18	2.5811747917131946e-09	-8.481608402251475e-07
21	9.559906635974793e-11	-5.444739336201271e-05
24	3.540706161472145e-12	-0.0034846392899683535
27	1.3113726523970905e-13	-0.22301691478444863
30	4.856935749618853e-15	-14.2730825462131

表 3: $x_0 = 1, x_1 = \dfrac{1}{3}$ 时通项估计值 $\hat{x_n}$ 与实际值 x_n 的比较

可以看到，误差被不断放大，甚至在 $n = 18$ 时出现了负值，通项收敛于 0 的性质被完全改变. 故该算法不稳定.

n	x_n	$\hat{x_n}$
0	1	1.0
2	16	15.999999999999998
4	256	255.9999999999999
6	4096	4095.999999999998
8	65536	65535.99999999997
10	1048576	1048575.9999999995
12	16777216	16777215.999999993
14	268435456	268435455.99999988
16	4294967296	4294967295.999998
18	68719476736	68719476735.99997
20	1099511627776	1099511627775.9995

表 4: $x_0 = 1, x_1 = 4$ 时通项估计值 $\hat{x_n}$ 与实际值 x_n 的比较

可见该算法数值稳定.

§4 结论

1. 结合精度丢失定理与 Taylor 展开，我们可以求出令 $y = x - sinx$ 丢失精度尽可能少的优良算法.

2. 对于数值计算而言，很多时候误差项难以完全避免，但如果产生误差的项前的系数不能被很好的控制，会导致误差随着迭代次数增多被严重放大，使得算法极不稳定. 故应在实际的程序数值计算中避免产生误差的项前的系数大于 1 的情况，或者使大于 1 的系数快速递减收敛于 1.

§5 附录: 程序代码

注: 3 道题目共用同一个 ipynb 程序文件.

```
import math
def x_sinx(x):
    if 1-math.sin(x)/x>=0.5:
        return x-math.sin(x)
    else:
        n=1
        y=0.
        while abs((x**(2*n+1.))/(math.factorial(2*n+1)))>1e-17:
            an=(-1)**(n+1)*(x**(2*n+1.))/(math.factorial(2*n+1))
            y=y+an
            n=n+1
        return y

# 初始化列表用于存储结果
results1 = []

x = -1e-4
while x >= -10:
    y = x_sinx(x)
    results1.append((x, y))
    x *= 10
# 从 1e-4 开始逐项乘以 10 直到 10
x = 1e-4
while x <= 10:
    y = x_sinx(x)
    results1.append((x, y))
    x *= 10

# 打印结果
for x, y in results1:
    print(f"x={x},y={y}")
```

```
# 初始化列表用于存储结果
results2 = []

x = -1e-4
while x >= -10:
    y = x-math.sin(x)
    results2.append((x, y))
    x *= 10
# 从 1e-4 开始逐项乘以 10 直到 10
x = 1e-4
```

```
while x <= 10:
    y = x-math.sin(x)
    results2.append((x, y))
    x *= 10

# 打印结果
for x, y in results2:
    print(f"x={x},y={y}")
```

```
results3=[]
from scipy.integrate import quad
def integral(y,n):
    result= math.e - (n+1)*y
    return result

def origin_function(x):
    return math.exp(x)

y0=quad(func=origin_function,a=0,b=1)[0]
y=y0
for n in range(21):
    results3.append((n,y))
    y=integral(y,n)

# 打印结果
for n, y in results3:
    print(f"n={n},y={y}")
```

```
def integral_function(x,n):
    return x**n*math.exp(x)

results4=[]
for n in range(21):
    y=quad(func=integral_function,a=0,b=1,args=(n,))
    results4.append((n,y[0]))

# 打印结果
for n, y in results4:
    print(f"n={n},y={y}")
```

```
results5=[]
for n in range(0,31,3):
    y=(1/3)**n
    results5.append((n,y))

for n,y in results5:
```

```
    print(f'n={n},y={y}')
```

```
results7=[]
x0=1.
x1=1/3
for n in range(31):
    if n==0:
        y=x0
        results7.append((n,y))
        continue
    elif n==1:
        y=x1
        results7.append((n,y))
        continue
    else:
        y=13/3*x1-4/3*x0
        results7.append((n,y))
        x0=x1
        x1=y

for n,y in results7:
    if n%3==0:
        print(f'n={n},y={y}')
```

```
results6=[]
for n in range(0,21,2):
    y=(4)**n
    results6.append((n,y))

for n,y in results6:
    print(f'n={n},y={y}')
```

```
results8=[]
x0=1.
x1=4
for n in range(0,21,1):
    if n==0:
        y=x0
        results8.append((n,y))
        continue
    elif n==1:
        y=x1
        results8.append((n,y))
        continue
    else:
        y=13/3*x1-4/3*x0
```

```
15          results8.append((n,y))
16          x0=x1
17          x1=y
18
19  for n,y in results8:
20      if n%2==0:
21          print(f'n={n},y={y}')
```