计算物理第二次作业

吴远清 2018300001031

作业1

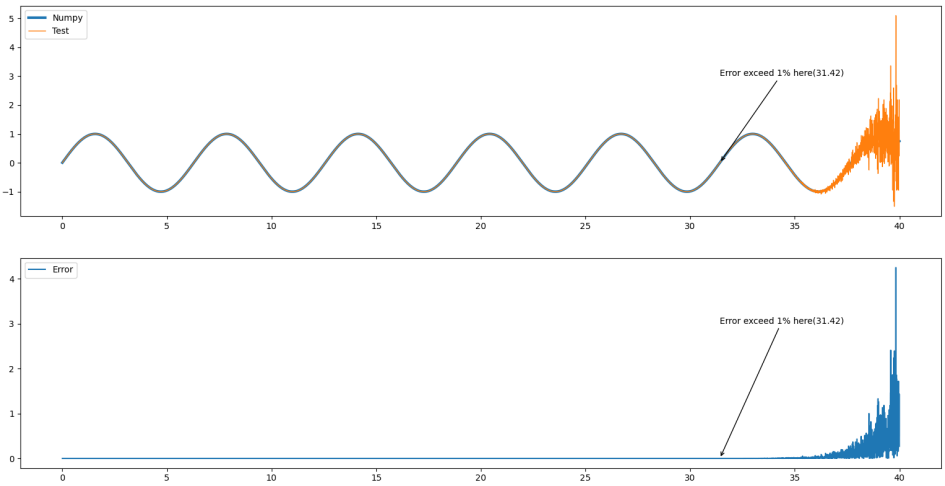
使用numpy自带的sin函数计算的结果作为真实值, 与自行实现的算法对比, 相对误差阈值设为1%, 超过则认为已发散.

代码:

```
import numpy as np
import matplotlib.pyplot as plt

def sin(x):
    eps = np.float32(10**-8)
    term = np.float32(x)
    sum = np.float32(x)
    n = 1
    while(True):
        n += 1
        term = -x*x*term/((2*n-1)*(2*n-2))
        sum += term
        if np.abs(term/sum) < eps:
            break
    return sum

testUnit = np.float32(0.005)
x,y1,y0,err = [],[],[],[]
maxAllowError = 0.01
firstErrorPoint = None
for i in range(1,8000):
    c = sin(i*testUnit)
    r = np.sin(i*testUnit)
    x.append(i*testUnit)
    y0.append(r)
    y1.append(c)
    if firstErrorPoint == None:
        if (np.abs(c-r)/r) > maxAllowError:
            firstErrorPoint = (i * testUnit, c)
    err.append(np.abs(c-r))
plt.subplot(2,1,1)
plt.plot(x, y0, label = 'Numpy', linewidth = 3)
plt.plot(x, y1, label = 'Test', linewidth = 1)
plt.annotate('Error exceed 1% here({:.2f})'.format(firstErrorPoint[0]),
firstErrorPoint, xytext = (firstErrorPoint[0], firstErrorPoint[1]+3), arrowprops
= dict(arrowstyle = '->'))
plt.legend()
plt.subplot(2,1,2)
plt.plot(x, err, label = 'Error')
plt.annotate('Error exceed 1% here({:.2f})'.format(firstErrorPoint[0]),
firstErrorPoint, xytext = (firstErrorPoint[0], firstErrorPoint[1]+3), arrowprops
= dict(arrowstyle = '->'))
plt.legend()
```

```
plt.show()
```

运行结果:



结果表明,$x > 31.42$时, 计算发散.

作业2

计算过程与结果中变量均为numpy.float32类型.

作为对比, 我们使用sympy库的Rational函数, 将所有项通分, 最后求值, 已此作为真实值.

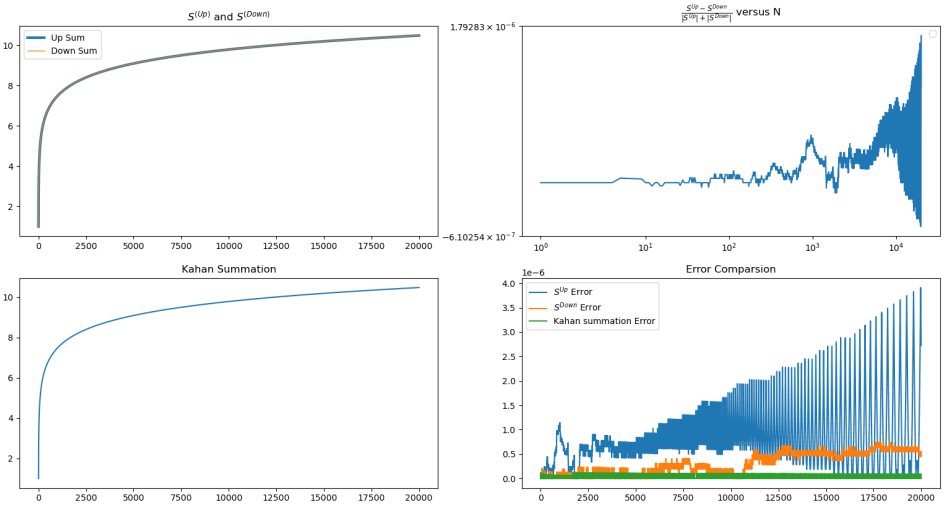
代码:

```
import numpy as np
import matplotlib.pyplot as plt
from sympy import Rational

def UpSum(n):
    term = np.float32(0)
    s = np.float32(0)
    for i in range(1, n + 1):
        term = np.float32(1/i)
        s += term
    return s

def DownSum(n):
    term = np.float32(0)
    s = np.float32(0)
    for i in range(n, 0, -1):
        term = np.float32(1/i)
        s += term
    return s

def KahanSum(n):
    s = np.float32(0)
    err = np.float32(0)
    term = np.float32(0)
```

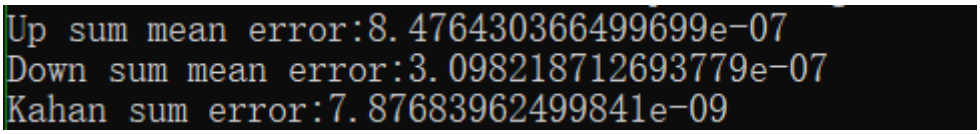
```
    for i in range(n, 0, -1):
        term = np.float32(1/i)
        y = np.float32(term - err)
        t = np.float32(s + y)
        err = (t - s) - y
        s = t
    return s

def TrueSum(n):
    s = Rational(1/1)
    for i in range(2, n + 1):
        s = s + Rational(1/i)
    return s.evalf()

def calcAll(n):
    err = np.zeros(n, dtype = np.float32)
    u = np.zeros(n, dtype = np.float32)
    d = np.zeros(n, dtype = np.float32)
    k = np.zeros(n, dtype = np.float32)
    t = np.zeros(n, dtype = np.float32)
    for i in range(1, n + 1):
        u[i - 1] = UpSum(i)
        d[i - 1] = DownSum(i)
        k[i - 1] = KahanSum(i)
        t[i - 1] = TrueSum(i)
        if i%int(n/10) == 0:
            print("{}/{}".format(i, n))
    return u, d, k, t

if __name__ == '__main__':
    totalNum = 2000
    u, d, k, t = calcAll(totalNum)
    err1 = (u - d) / (np.abs(u) + np.abs(d))
    err2 = np.abs(t - u)/t
    err3 = np.abs(t - d)/t
    err4 = np.abs(t - k)/t
    n = np.linspace(1, totalNum, totalNum)
    plt.subplot(2,2,1)
    plt.title('$S^{(Up)}$ and $S^{(Down)}$')
    plt.plot(n, u, label = 'Up Sum', linewidth = 3)
    plt.plot(n, d, label = 'Down Sum', linewidth = 1)
    plt.legend()
    plt.subplot(2,2,2)
    plt.title('$\\frac{S^{Up}-S^{Down}}{|S^{Up}|+|S^{Down}|}$ versus N')
    plt.xscale('symlog')
    plt.yscale('symlog')
    plt.plot(n, err1)
    plt.legend()
    plt.subplot(2,2,3)
    plt.title('Kahan Summation')
    plt.plot(n, k)
    plt.subplot(2,2,4)
    plt.title('Error Comparsion')
    plt.plot(n, err2, label = '$S^{Up}$ Error')
    plt.plot(n, err3, label = '$S^{Down}$ Error')
    plt.plot(n, err4, label = 'Kahan summation Error')
    plt.legend()
    plt.show()
```

```
    print('Up sum mean error:{}\nDown sum mean error:{}\nKahan sum error:
{}'.format(np.mean(err2), np.mean(err3), np.mean(err4)))
```

运行结果:



结果分析与解答:



可以明显的观察到, $S^{(Up)}$的误差远大于$S^{(Down)}$和Kahan Summation, 并且, 在多数情况下, 同样为降序求和, Kahan Summation的误差小于简单的$S^{(Down)}$.

首先,回答题目中的问题: $S^{(Up)}$为何的误差大于$S^{(Down)}$?

原因是, 对于i较大时,$\frac{1}{i}$较小,降序求和时, 对于较大的i, sum值也较小, 导致$\frac{1}{i}$被略去的位数较少, 误差较小.

但对于升序求和, 当i增长到较大时, 此时的sum值已经增长的较大, 使得$\frac{1}{i}$被略去的位数多, 误差较大.

最后, 我们考虑Kahan Summation为什么能减小误差?

我们考虑这样的式子:

$$S = A + B$$

$$Err = S - A - B$$

可以发现, 通过这样, 就可以计算除$A + B$过程中产生的误差, 并在下一步中进行补偿, 以此减小误差.

作为对上述解释的补充证明, 可以发现, 明显和$S^{(Up)}$与$S^{(Down)}$不同的, Kahan Summation的误差基本与n无关.