Computational Physics, PHYS-UA 210, Problem Set #2

Zhonghan (Brian) Wang

Due: September 22, 2017

2. As we have seen, subtractive cancellation occurs when summing a series with alternating signs. As another example, consider the finite sum

$$S_N^{(1)} = \sum_{n=1}^{2N} (-1)^{-n} \frac{n}{n+1}. \tag{3.10}$$

If you sum the even and odd values of n separately, you get two sums:

$$S_N^{(2)} = -\sum_{n=1}^{N} \frac{2n-1}{2n} + \sum_{n=1}^{N} \frac{2n}{2n+1}. \tag{3.11}$$

All terms are positive in this form with just a single subtraction at the end of the calculation. Yet even this one subtraction and its resulting cancellation can be avoided by combining the series analytically to obtain

$$S_N^{(3)} = \sum_{n=1}^{N} \frac{1}{2n(2n+1)}. \tag{3.12}$$

Although all three summations $S^{(1)}$, $S^{(2)}$, and $S^{(3)}$ are mathematically equal, they may give different numerical results.

(a) Write a double-precision program that calculates $S^{(1)}$, $S^{(2)}$, and $S^{(3)}$.

```
def S1(n=None):
    summands=np.arange(1,2*n+1, dtype=np.float64)
    for i in np.arange(1,2*n+1, dtype=np.int64):
        summands[i-1]=((-1)**i)*((i)/(i+1))
    ans1=np.sum(summands)
    return(ans1)

def S2(m=None):
    nsummands=np.arange(1,m+1, dtype=np.float64)
    for i in np.arange(1,m+1, dtype=np.int64):
        nsummands[i-1]=(2*i-1)/(2*i)
    psummands=np.arange(1,m+1, dtype=np.float64)
    for i in np.arange(1,m+1, dtype=np.int64):
        psummands[i-1]=(2*i)/(2*i+1)
    ans2=np.sum(psummands-nsummands)
    return(ans2)

def S3(l=None):
    tsummands=np.arange(1,l+1, dtype=np.float64)
    for i in np.arange(1,l+1, dtype=np.int64):
        tsummands[i-1]=(1)/((2*i)*(2*i+1))
    ans3=np.sum(tsummands)
    return(ans3)
```

I first generate all the summands in an array then sum them. It's faster than summing it term by term.

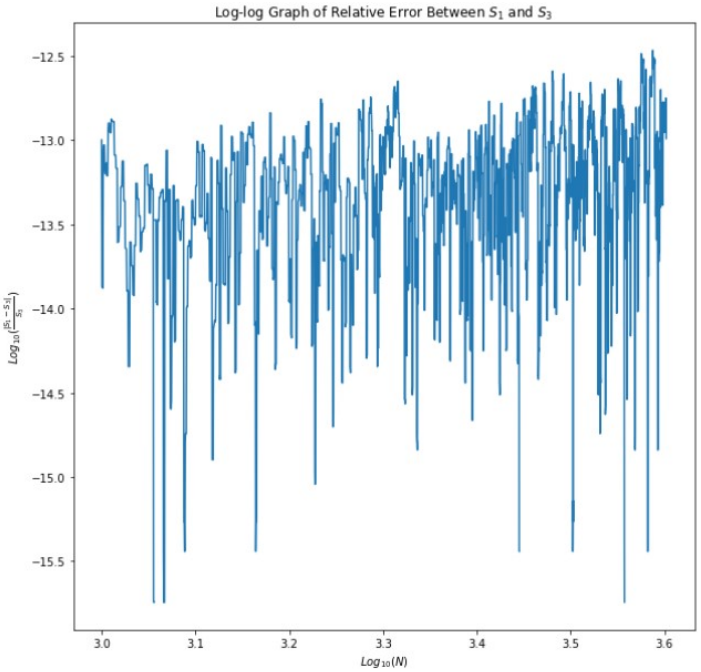
(b) Assume S^3 to be the exact answer. Make a log-log plot of the relative error vs. the number of terms, that is, of $\log_{10}|(S_N^{(1)} - S_N^{(3)})/S_N^{(3)}|$ vs. $\log_{10}(N)$. Start with $N = 1$ and work up to $N = 1,000,000$. (Recollect that $\log_x = \ln x / \ln 10$.) The negative of the ordinate in this plot gives an approximate value for the number of significant figures.

```
b=np.arange(1000,4001, dtype=np.int64);
error1=np.arange(1000,4001, dtype=np.float64);
for i in np.arange(1,3001):
    error1[i]=np.absolute((S1(b[i])-S3(b[i]))/S3(b[i]));
```

I decide to start at 1000 because some of the error terms are zero for lower N. I can't go up to $N = 1,000,000$ because it takes too long for the algorithm to generate many points above $N = 100,000$.

```
error1[0]=10**(-13)
x1=np.log(b)/np.log(10)
y1=np.log(error1)/np.log(10)

plt.figure(figsize=(10,10))
plt.title(r"Log-log Graph of Relative Error Between $S_1$ and $ S_3$")
plt.xlabel(r"$Log_{10}(N)$")
plt.ylabel(r"$Log_{10}(\frac{|S_1 - S_3|}{S_3})$")
plt.plot(x1,y1)
plt.show()
```

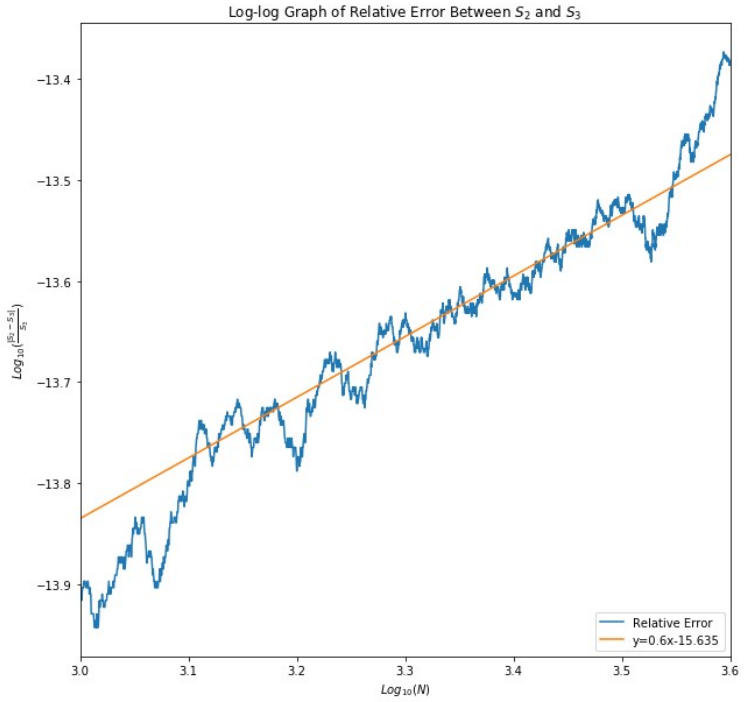


There doesn't seems to be any relation between the error of S1 and S3, just random fluctuations.

```
t=np.arange(1000,4001, dtype=np.int64);
error2=np.arange(1000,4001, dtype=np.float64);
for i in np.arange(1,3001):
    error2[i]=np.absolute((S2(b[i])-S3(b[i]))/S3(b[i]));

error2[0]=10**(-13.9)
x2=np.log(t)/np.log(10)
y2=np.log(error2)/np.log(10)
x3=np.linspace(3.0,3.6,1000)
y3=0.60*x3-15.635

plt.figure(figsize=(10,10))
plt.title(r"Log-log Graph of Relative Error Between $S_2$ and $ S_3$")
plt.xlabel(r"$Log_{10}(N)$")
plt.ylabel(r"$Log_{10}(\frac{|S_2 - S_3|}{S_3})$")
plt.xlim(3.0,3.6)
plt.plot(x2,y2,label="Relative Error")
plt.plot(x3,y3,label="y=0.6x-15.635")
plt.legend(loc=4)
plt.show()
```



(c) See whether straight-line behavior for the error occurs in some region of your plot. This indicates that the error is proportional to a power of N.

The relative error between S2 and S3 is more interesting as we can see an almost linear relationship between the relative error and some power of N. I guessed a line to show the linearity.