SPS4050 Homework 1

Note: this homework set will be graded out of 18 points, with a maximum allowed score of 21 points. Remember to include your code alongside any output or by-hand derivations.

1. **(5 pts total)** The Taylor series expansion of e^x is

$$e^x = \varepsilon_N + \sum_{i=0}^{N} \frac{x^i}{i!},$$

where ε_N is the (usually unknown) error associated with the truncated summation. Note that as $N \to \infty$, $\varepsilon_N \to 0$.

   (a) **(3 pts)** Compute the first 21 partial sums of $e^{0.5}$—that is, N from 0 to 20—and **plot** the absolute value of the error as a function of N. To compute the error, use the true value $e^{0.5} = 1.6487212707001281$. (Your plot should use a logarithmic y axis.)

   (b) **(2 pts)** Compute the first 101 partial sums of e^{10}, and **plot** the absolute value of the error as a function of N. To compute the error, use the true value $e^{10} = 22026.465794806717$. (Your plot should use a logarithmic y axis.)

2. **(5 pts)** In this problem you will benchmark a "brute force" approach to matrix multiplication against the library version contained in `numpy`. The brute force code is reproduced at right.

```
import numpy as np

N = 100
C = np.zeros((N,N))

for i in range(N):
  for j in range(N):
    for k in range(N):
      C[i,j] += A[i,k]*B[k,j]
```

   For $N = 50$, 150, and 500, create two arrays `A` and `B` filled with uniform random numbers (see the demo from class). Time the matrix multiplication using the code at right, and again using the `numpy` function `matmul`. Plot the six data points (three values of N and two multiplication methods), and compute the scaling for each method.

   (This problem also demonstrates why using pre-written libraries is far preferable to writing your own from scratch, when that option exists.)

3. **(5 pts total)** Consider two true values $\tilde{x} = 12345678912345678.0$ and $\tilde{y} = 12345678912345677.0$. If we try to evaluate $\tilde{x}^2 - \tilde{y}^2$ we will experience catastrophic cancellation: each squaring operation leads to a rounding error, which is exacerbated by the subtraction. Write code to do the following tasks:

   (a) **(2 pts)** Carry out the calculation $\tilde{x}^2 - \tilde{y}^2$ using integers entered by hand (not recast using the built-in `int()` function).

   (b) **(2 pts)** Carry out the calculation $\tilde{x}^2 - \tilde{y}^2$ using floats, which should show catastrophic cancellation when compared to the result from part (a).

   (c) **(1 pt)** Carry out the calculation $(\tilde{x}+\tilde{y})\cdot(\tilde{x}-\tilde{y})$, which is mathematically identical to $\tilde{x}^2-\tilde{y}^2$, using floats. Does your answer match the integer result or the catastrophic-cancellation result? Why?

4. **(3 pts total)** The previous problem considered catastrophic cancellation. Rewrite the following expressions using analytical manipulations to avoid catastrophic cancellation for large values of x. (Consider testing your "before" and "after" expressions to see whether they do avoid catastrophic cancellation.)
   (a) **(1 pt)** $\sqrt{x+1} - \sqrt{x}$
   (b) **(1 pt)** $\dfrac{1}{x+1} - \dfrac{2}{x} + \dfrac{1}{x-1}$
   (c) **(1 pt)** $\dfrac{1}{\sqrt{x}} - \dfrac{1}{\sqrt{x-1}}$

5. **(5 pts)** The code below does not include subtraction, large numbers, or fractions; there is nothing to suggest that catastrophic cancellation might occur. This is, instead, an example of how roundoff errors can accumulate slowly, then undergo an irreversible collapse in relatively few operations.

   (a) **(1 pt)** Run the code at right and include its output as part of your turned-in solutions.

   (b) **(2 pts)** Modify the code, and plot `f(5.0,n)` for all integers between 0 and 100 inclusive.

   (c) **(2 pts)** Explain why you get the output you see in part (b).

```
from math import sqrt

def f(x,n):
  for i in range(n):
    x = sqrt(x)
  for i in range(n):
    x = x**2
  return x

for x in (0.5, 5.0):
  print( x, f(x,100) )
```