Computational physics problems

Roy Rinberg

problems:

2.4.3.2 ,

Write a program that determines the underflow and overflow limits (within a factor of 2) for Python on your computer.

a) Check where under- and overflow occur for double-precision floating-point numbers (floats). Give your answer in decimal

b) Check where under- and overflow occur for double-precision floating-point numbers

c) Check where under- and overflow occur for integers. Note: There is no exponent stored for integers, so the smallest integer corresponds to themost negative one. To determine the largest and smallest integers, youmust observe your program's output as you explicitly pass through the limits. You accomplish this by continually adding and subtracting 1. (Because integer arithmetic uses two's complement arithmetic, you should expect some surprises.)

```
underflow for floats at 1074
done with underflow for float 0.000000
```

The smallest number my computer can calculate using a float is 2^-1074 . Though, this seems a little odd.

```
underflow for integers at 63
done with underflow for integers -9223372036854775808
```

The most negative integer (prior to python automatically converts the number to a float, I believe) is this (- 2^63)

```
overflow for float at 1023
done with over flow for floats: inf
```

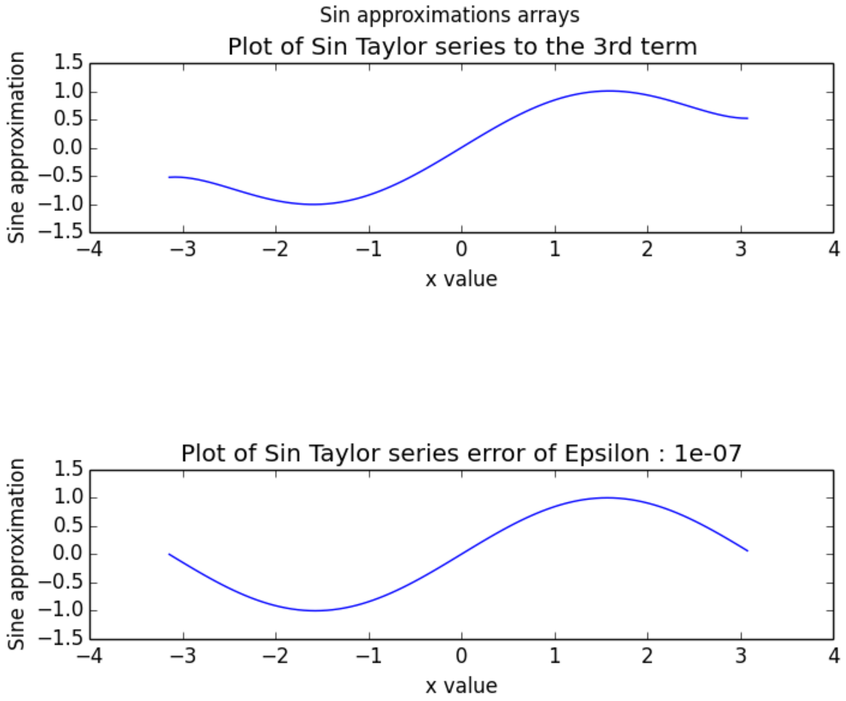
The most positive float (prior to python automatically converts the number to inf) is this (2^1023)

```
overflow for integers at 62
done with overflow for integers 4611686018427387904.000000
```

This last calculation was made by doubling an integer, and looking when that integer became cast into a float, then halving the float.

2^63

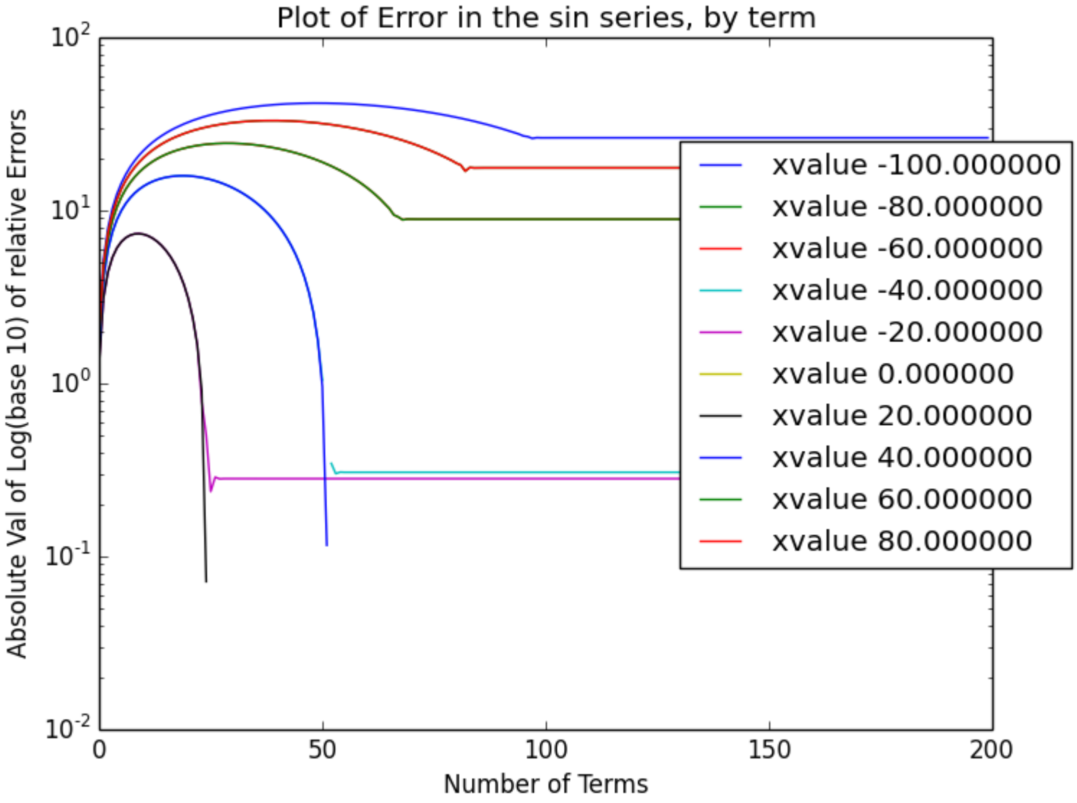
Questions 2.5 and 3.3.1 are very similar in spirit, so I have answered both in spirit, without spending the time to answer each question asked individually.



For this problem, I created a set of functions that would evaluate the Taylor approximation of a sine series to a certain term. One function would evaluate the series such that the relative error (approximate value – actual value)/actual value was less than some epsilon (note that clearly when I say ‘actual’ value, I really mean, more accurate sin value computed by numpy).

For a table of approximated sin values from –Pi to Pi, their difference from a more accurate sin, and their relative error, see the attached text file.

For sufficiently small values of x, the algorithm converges; and for larger values of x, the algorithm also converges, but to the wrong answer:



Here is a plot of the log (base 10) of the absolute value of the relative error of the sine series. You can see that all the values do converge – this makes sense mathematically too, as a factorial grows faster than a polynomial for n goes to infinity, so the alternating sum must converge. Clearly here the sum does not converge to 0, so the values converge to incorrect results.

I worked to plot: the x-value that would be the first convergent value, for a given number of terms, but I could not get it to provide me with a meaningful result. So then I thought about it, and I applied the ratio test – such that I would say that the series converges for some epsilon if A_{n+1}/A_n <Epsilon

$$\frac{A_{n+1}}{A_n} < \varepsilon$$

So $\frac{x^2}{(2n+1)*(2n)} = \varepsilon$, will solve for you the n (number of terms) required to conver for a given x, and a given epsilon.

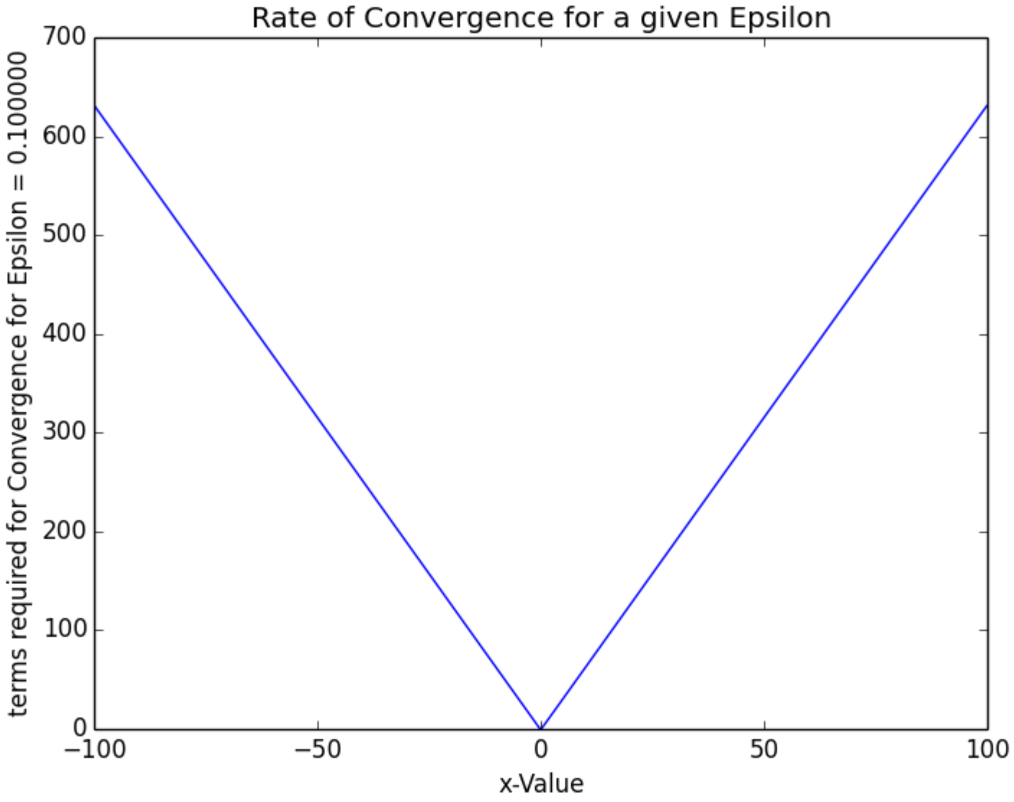
Let $c = x^2/\varepsilon$

$$4n^2 + 2n = c$$

$$n = \frac{-2 \pm \sqrt{(4 + 16c)}}{8}$$
$$c = \delta^2$$
$$\delta = \frac{x}{\sqrt{\varepsilon}}$$
$$n \approx -\frac{1}{4} \pm \frac{\delta}{2}$$

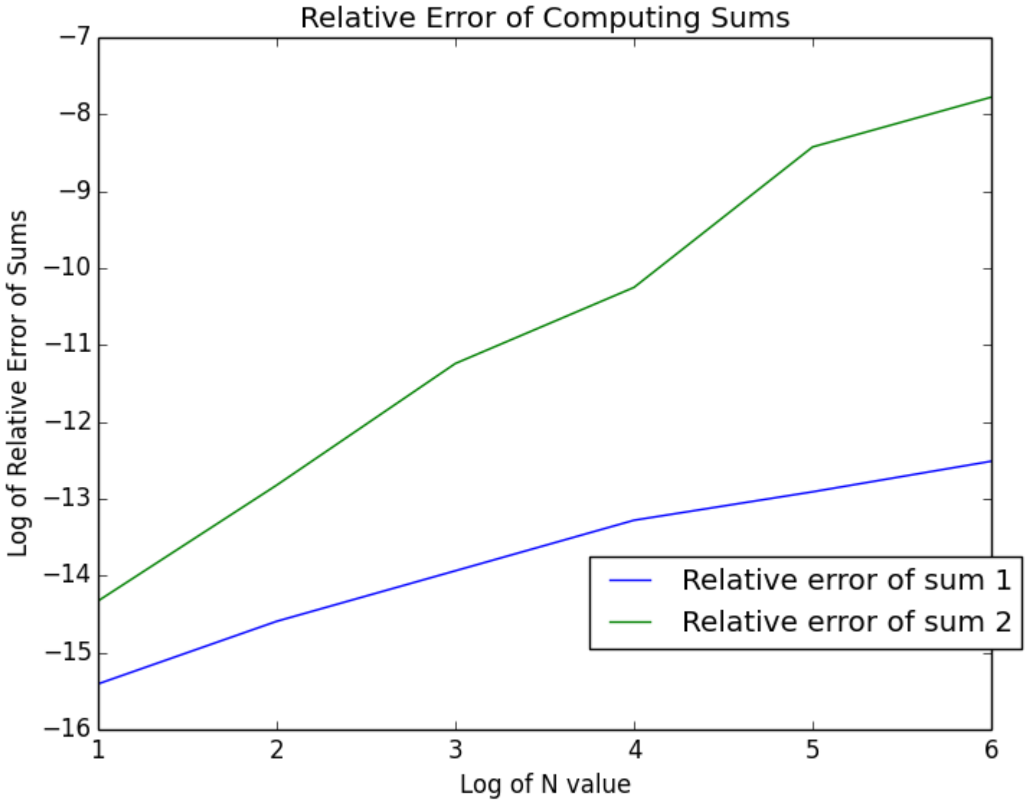
(specifically for $4c \gg 1$ which is true when $x = \sqrt{\varepsilon}$)

This is a somewhat surprising result, as it states that for each unit you move away from x=0, you nearly require 1 more term (times some constant of proportionality, dependent on the specific epsilon) in order to *converge* for the Taylor series of a sine function.
I am not 100% this is true, but I couldn't find any flaws with what I did



Lastly, let us look at problem 3.1.2.2

a. Write a double-precision program that calculates S(1), S(2), and S(3) :See the attached Python file b.



For some reason, I received errors when I attempted to do plt.loglog, instead of plt.plot – so the data is still logarithmic.

c. Straight line behavior roughly occurs in the entire plot. So, clearly error is proportional to some power 10^{aN}