18.335 Problem Set 1

Due February 24, 2023 at 11:59pm. You should submit your problem set **electronically** on the 18.335 Gradescope page. Submit **both** a *scan* of any handwritten solutions (I recommend an app like TinyScanner or similar to create a good-quality black-and-white "thresholded" scan) and **also** a *PDF printout* of the Julia notebook of your computer solutions. A **template Julia notebook is posted** in the 18.335 web site to help you get started.

Problem 0: Pset Honor Code

Include the following statement in your solutions:

> *I will not look at 18.335 pset solutions from previous semesters. I may discuss problems with my classmates or others, but I will write up my solutions on my own. <your signature>*

Problem 1: Jupyter notebook

On the course home page, "launch a Julia environment in the cloud" and open the "Floating-Point-Intro.ipynb" notebook in the notes folder. Read through it, get familiar with Julia and the notebook environment, and play! (You don't need to submit a notebook print out or turn in work for this question.)

Problem 2: Floating point

(From Trefethen and Bau, Exercise 13.2.) The floating point numbers $\mathbb{F}$ can be written compactly in the form (e.g., see (13.2) in Trefethen and Bau) $x = \pm(m/\beta^t)\beta^e$, with integer base $\beta \geq 2$, significand $\beta^{-t} \leq m/\beta^t \leq 1$, and integer exponent e. This floating point system includes many integers, but not all of them.

(a) Give an exact formula for the smallest positive integer n that does not belong to $\mathbb{F}$.

(b) In particular, what are the values of n for IEEE single and double precision arithmetic?

(c) Figure out a way to verify this result for your own computer. Specifically, design and run a program that produces evidence that $n-3$, $n-2$, and $n-1$ belong to $\mathbb{F}$ but n does not. What about $n+1$, $n+2$, and $n+3$?

(In part (c), you can use Julia, which employs IEEE double precision by default. However, unlike Matlab, Julia distinguishes between integer and floating-point scalars. For example, `2^50` in Julia will produce a 64-bit integer result; to get a 64-bit/double floating-point result, do e.g. `2.0^50` instead.)

Problem 3: Funny functions

(a) Write a function `L4(x,y)` in Julia to compute the L_4 norm $(|x|^4 + |y|^4)^{1/4}$ of two scalars x and y. Does your code give an accurate answer for `L4(1e-100,0.0)`? What about `L4(1e+100,0.0)`? Without using arbitrary-precision (`BigFloat`) calculations, **fix your code** so that it gives an answer whose relative error $\frac{|\text{computed}-\text{correct}|}{|\text{correct}|}$ is within a small multiple of `eps()` $= \epsilon_{\text{machine}}$ (a few "ulps", or "units in the last place") of the exactly rounded answer for all double-precision x and y. (You can test your code by comparing to `L4(big(x),big(y))`, i.e. arbitrary-precision calculation.)

(b) Write a function `cotdiff(x,y)` that computes $\cot(x) - \cot(x + y)$. Does your code give an accurate answer for `cotdiff(1.0, 1e-20)`? Without using arbitrary-precision (`BigFloat`) calculations, **fix your code** so that it gives an accurate `Float64` answer (within a few ulps) even when $|y| \ll |x|$ (without hurting the accuracy when y and x are comparable!). (Hint: one option would be to switch over to Taylor expansion when $|y|/|x|$ is sufficiently small, but a simpler solution is possible by applying some trigonometric identities.)

Problem 4: Addition, another way

Here you will analyze $f(x) = \sum_{i=1}^{n} x_i$, but you will compute $\tilde{f}(x)$ in a different way from the naive sum considered in class. In particular, compute $\tilde{f}(x)$ by a recursive divide-and-conquer approach, recursively dividing the set of values to be summed in two halves and then summing the halves:

$$\tilde{f}(x) = \begin{cases} 0 & \text{if } n = 0 \\ x_1 & \text{if } n = 1 \\ \tilde{f}(x_{1:\lfloor n/2 \rfloor}) \oplus \tilde{f}(x_{\lfloor n/2 \rfloor + 1:n}) & \text{if } n > 1 \end{cases},$$

where $\lfloor y \rfloor$ denotes the greatest integer $\leq y$ (i.e. y rounded down). In exact arithmetic, this computes $f(x)$ exactly, but in floating-point arithmetic this will have very different error characteristics than the simple loop-based summation in class.

(a) For simplicity, assume n is a power of 2 (so that the set of numbers to add divides evenly in two at each stage of the recursion). Prove that $|\tilde{f}(x) - f(x)| \leq \epsilon_{\text{machine}} \log_2(n) \sum_{i=1}^{n} |x_i| + O(\epsilon_{\text{machine}}^2)$. That is, show that the worst-case error bound grows *logarithmically* rather than *linearly* with n!

(b) Pete R. Stunt, a Microsoft employee, complains, "While doing this kind of recursion may have nice error characteristics in theory, it is ridiculous in the real world because it will be insanely slow—I'm proud of my efficient software and can't afford to have a function-call overhead for every number I want to add!" Explain to Pete how to implement a slight variation of this algorithm with the same logarithmic error bounds (possibly with a worse constant factor) but roughly the same performance as a simple loop.

(c) In the pset 1 Julia notebook, there is a function "div2sum" that computes $\tilde{f}(x) =$`div2sum(x)` in single precision by the above algorithm. Modify it to not be horrendously slow via your suggestion in (b), and then plot its errors for random inputs as a function of n with the help of the example code in the Julia notebook (but with a larger range of lengths n). Are your results consistent with your error bounds above?