18.335 Problem Set 1

Due Friday, 15 February 2019. You should submit your problem set **electronically** on the 18.335 Stellar 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 1: Floating point

Trefethen, probem 13.2. (For 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 2: 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 3: Newtonish methods

Newton's method for a root of $f(x) = 0$ is to iterate $x_{n+1} = x_n - f(x_n)/f'(x_n)$ starting from some initial guess x_1 (which must be sufficiently close to the root to guarantee convergence in general). Suppose that you are also given $f''(x)$, the second derivative. In this problem, you will propose an iteration scheme that takes advantage of this second-derivative information.

(a) Propose a Newton-like iteration that takes advantage of f, f', and f'' (assuming f is analytic in the neighborhood of the root). (Hint: use a second-order Taylor approximation of f.) If you solve a quadratic equation $ax^2 + bx + c = 0$, you might want to compute the two roots as $\frac{-b-\sqrt{b^2-4ac}}{2a}$ and $\frac{2c}{-b-\sqrt{b^2-4ac}}$, assuming $b > 0$, to avoid disastrous cancellation errors for $|ac| \ll b^2$. (Google "quadratic formula cancellation error." In the event of a disaster, your method can fall back to an ordinary Newton step.)

(b) Analyze its asymptotic convergence rate: if x is an exact root, write $x_n = x(1 + \delta_n)$ as in class, and solve for δ_{n+1} in terms of δ_n assuming you are close to the root ($\delta_n \ll 1$).

(c) Modify the Julia Newton's-method notebook from class to implement your method to compute a root of $f(x) = x^3 - 1$. In particular start with $x_1 = 2$, so that your scheme should(!) converge to $x = 1$, and look at the error $x_n - 1$. Demonstrate that it agrees with your predicted convergence rate from the previous part. [You should use arbitrary precision as in the notebook from class, so that you can watch the convergence rate for many digits. An approximate number of accurate digits is given by $-\log_{10}(x_n - 1)$.]

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?