18.335 Problem Set 2

Due Fri., 27 September 2013.

Problem 1: Floating-point

(a) 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.)

(b) A generalization of Trefethen, problem 14.2: given a function $g(x)$ that is analytic (i.e., has a Taylor series) for $|x|$ sufficiently small, and $g'(0) \neq 0$, show that $g(O(\epsilon)) = g(0) + g'(0)O(\epsilon)$.

(c) Julia provides a function `log1p(x)` that computes $\ln(1+x)$. What is the point of providing such a function, as opposed to let the user compute $\ln(1+x)$ herself? (Hint: not performance.) Propose and write a possible implementation of log1p for double-precision inputs, in the form:

```
function mylog1p(x::Float64)
.......
end
@vectorize_1arg Float64 mylog1p
```

(the second line extends your function to work on arrays) and demonstrate via

```
x = logspace(-100, 100)
exact = float64(log1p(map(big,x)))
max(abs(exact - mylog1p(x)) ./
exact)
```

that your mylog1p function is accurate (the maximum relative error should $\lesssim 10^{-14}$) (here, the `exact` vector is computed in 256-bit arbitrary-precision arithmetic via the `big` function). You *may* use the built-in `log` function inside your `mylog1p` function.

Problem 2: Addition

This problem is about the floating-point error involved in summing n numbers, i.e. in computing the function $f(x) = \sum_{i=1}^{n} x_i$ for $x \in \mathbb{F}^n$ ($\mathbb{F}$ being the set of floating-point numbers), where the sum is done in the most obvious way, in sequence. In pseudocode:

```
sum = 0
for i = 1 to n
    sum = sum + x_i
f(x) = sum
```

For analysis, it is a bit more convenient to define the process inductively:

$$\begin{aligned} s_0 &= 0 \\ s_k &= s_{k-1} + x_k \text{ for } 0 < k \leq n, \end{aligned}$$

with $f(x) = s_n$. When we implement this in floating-point, we get the function $\tilde{f}(x) = \tilde{s}_n$, where $\tilde{s}_k = \tilde{s}_{k-1} \oplus x_k$, with $\oplus$ denoting (correctly rounded) floating-point addition.

(a) Show that $\tilde{f}(x) = (x_1 + x_2)\prod_{k=2}^{n}(1+\epsilon_k) + \sum_{i=3}^{n} x_i \prod_{k=i}^{n}(1+\epsilon_k)$, where the numbers ϵ_k satisfy $|\epsilon_k| \leq \epsilon_{\text{machine}}$. Equivalently, show that

$$\tilde{f}(x) = \sum_{i=1}^{n} x_i \prod_{k=i}^{n}(1+\epsilon_k),$$

where $\epsilon_1 = 0$ (assuming correct rounding).

(b) Show that $\prod_{k=i}^{n}(1+\epsilon_k) = 1+\delta_i$ where $|\delta_i| \leq (n-i+1)\epsilon_{\text{machine}} + O(\epsilon^2_{\text{machine}})$.

(c) Show that the error can be bounded as: $|\tilde{f}(x) - f(x)| \leq n\epsilon_{\text{machine}} \sum_{i=1}^{n} |x_i| + O(\epsilon^2_{\text{machine}})$.

(d) Suppose that the ϵ_k values are uniformly randomly distributed in $[-\epsilon_{\text{machine}}, +\epsilon_{\text{machine}}]$. Explain why the *mean* error can be bounded by $|\tilde{f}(x) - f(x)| = O\left(\sqrt{n}\epsilon_{\text{machine}} \sum_{i=1}^{n} |x_i|\right)$. (Hint: google "random walk"...you can just quote standard statistical results for random walks, no need to copy the proofs.)

(e) Compare your error bounds above to numerical experiments in Julia. Here, we will use an old trick to compute the floating-point errors: compare the results computed in one precision to the "exact" results computed in a higher precision. In particular, we use the `float32` function (which converts a number to single precision) accumulate the sum in single precision, rather than Julia's

default double precision. Plot the error $|\tilde{f}(x) - f(x)| / \sum_i |x_i|$ as a function of n on a log–log scale (Julia's `loglog` command), and explain your observation in terms of your results above.[1]

This is implemented in the example file `loopsum.jl`, posted on the course page, which computes the sum $f(x)$ =`loopsum(x)` via the above algorithm in single precision. For your numerical experiment, compute the sum of n random inputs $x \in [0,1)^n$ via Julia's `rand(n)` function. You can then compute $|\tilde{f}(x) - f(x)| / \sum_i |x_i|$ for a given n via

```
x = rand(n)
err = abs(loopsum(x) - sum(x)) /
sum(abs(x))
```

Problem 3: Addition, another way

Here you will analyze $f(x) = \sum_{i=1}^{n} x_i$ as in problem 2, but this time you will compute $\tilde{f}(x)$ in a different way. 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 problem 2.

(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). With an analysis similar to that of problem 2, prove that $|\tilde{f}(x) - f(x)| \leq \epsilon_{\text{machine}} \log_2(n) \sum_{i=1}^{n} |x_i| + O(\epsilon^2_{\text{machine}})$. That is, show that the worst-case error bound grows *logarithmically* rather than *linearly* with n!

(b) If the floating-point rounding errors are randomly distributed as in problem 2, estimate the average-case error bound.

(c) 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 (hint: look at how I implemented recursive matrix multiplication in my cache-oblivious handout from lecture 3).[2]

(d) On the course web page, I've posted a file `div2sum.jl` 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 (c), and then plot its errors for random inputs as a function of n as in problem 2. Are your results consistent with your error estimates above?

(e) Suppose we now multiply two $m \times m$ random matrices A and B ($\in [0,1)^{m \times m}$, uniformly distributed) to form $C = AB$. If you look at any given entry c_{ij} of C, how quickly do you expect the errors to grow with m if you compute AB via the simple 3-loop row-column algorithm? What if you use the optimal cache-oblivious algorithm from class?

Problem 4: Stability

(a) Trefethen, exercise 15.1. [In parts (e) and (f), assume that $\frac{1}{k!}$ can be computed to $O(\epsilon_{\text{machine}})$ and concentrate on the accumulation of errors in the summations.]

(b) Trefethen, exercise 16.1.

[1] Use enough n values to get a clear result. e.g. the command `n = int(round(logspace(2,6,100)))` gives 100 logarithmically spaced n values from 10^2 to 10^6.

[2] In fact, there is a common real-world algorithm that does summation in precisely this recursive way: the Cooley-Tukey fast Fourier transform.

Problem 5: SVD and low-rank approximations

(a) Trefethen, probem 4.5.

(b) Trefethen, problem 5.2.

(c) Trefethen, problem 5.4.