18.335 Problem Set 2

Due Monday, 16 February 2015.

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: Quadratic blues

Suppose we are solving the quadratic equation $x^2 - 2bx + c = 0$. The familiar quadratic formula gives the two solutions $x_\pm = b \pm \sqrt{b^2 - c}$. Here, we will consider the accuracy of the x_- root for $|c| \ll b^2$.

(a) Using the code from the pset1 Julia notebook posted on the web page, plot the accuracy of $x_- = b - \sqrt{b^2 - c}$ for $b = 1$ and for a range of c values from 10^{-1} to 10^{-20}. Explain the observed inaccuracy.

(b) Propose an alternative way to compute x_- that will yield accurate results (within a factor of 10 of machine precision) in double precision. Implement your method and plot its accuracy in Julia using a modified version of the code in the previous part.

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, make sure you avoid the inaccuracy problems that arose in problem 2 above (you can use your solution from problem 2). (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

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:

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

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

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

where $\epsilon_1 = 0$ and where the other ϵ_k satisfy $|\epsilon_k| \le \epsilon_{\text{machine}}$.

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

(c) 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.) This explains the "my_cumsum" results shown in class.

Problem 5: 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.

(d) 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 (c), 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 estimates above?