APC 523/MAE 507/AST 523
Numerical Algorithms for Scientific Computing
Problem Set 1
(EXTENSION – due Wednesday 13 March 2019, 8:55am)

1 Error in (symmetric) rounding vs. chopping

In class, we discussed two general ways to map a real number x to a nearby machine number in $\mathbb{R}(p, q)$:

- **truncation** of everything after p significant figures in the mantissa (this amounts to always rounding *down*), or
- **symmetric rounding** (round down when the first discarded bit b_{p+1} is 0, round up when b_{p+1} is 1).

We showed that an upper bound on the relative error in replacing x by trunc(x) was

$$\left|\frac{x - \text{tr}(x)}{x}\right| \le 2 \cdot 2^{-p} \quad . \tag{1}$$

We also asserted (but did not prove) that the upper bound in the relative error for symmetric rounding was a factor of 2 smaller:

$$\left|\frac{x - \text{rd}(x)}{x}\right| \le 2^{-p} \quad . \tag{2}$$

Prove equation (2). HINT: you should probably consider separately the cases in which the first discarded bit is 0 and in which it's 1.

Recall, for the remainder of this problem set, that this value eps $\equiv 2^{-p}$ is what we call **machine epsilon**. It's the largest relative error that can be introduced simply by mapping a real number onto an element of $\mathbb{R}(p, q)$. And it's often convenient to denote the result of rounding to an element of $\mathbb{R}(p, q)$ as rd$(x) \equiv x(1 + \varepsilon)$, $|\varepsilon| \le$ eps.

2 An accurate implementation of e^x

The exponential function $f(x) = e^x$ can be represented by a power series

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad , \tag{3}$$

which is uniformly convergent on $\mathbb{R}$. For this problem, suppose that we are working in a floating-point system with five decimal digits in the mantissa (i.e. five significant figures).

(a) Approximate $e^{5.5}$ by working out the terms in this series up to $n = 30$ (so the first 31 terms), assuming a mantissa with five decimal digits.[1] In each term, evaluate the numerator and denominator separately and then divide them. Treat exponentiation as repeated multiplication, and evaluate equal-precedence operations in left-to-right associative order (i.e. we evaluate x^4 as $(((x \cdot x) \cdot x) \cdot x)$).

[1]Writing a bit of code to do the rounding automatically isn't super tough (the %g format specifier in languages/environments that understand C-style `printf` formatting could do this, for example, but you'd need to apply it after each individual operation and do repeat floating pt. to string back to floating pt conversions). Anyway, it may be worth writing a routine to do this out of general interest, but testing it to be sure it's foolproof will likely take you a while longer than you expect, so it's probably quicker just to do this out manually (i.e. do each operation in an interactive MATLAB or Python session and manually round to 5 sigfigs after each operation).

(b) Now work out the partial sums

$$S_k \equiv \sum_{n=0}^{k} \frac{x^n}{n!}$$

up to $k = 30$ (still for $x = 5.5$) by adding from left to right, i.e. compute S_{k+1} as $S_k + (5.5)^{k+1}/(k+1)!$. For what value of k has the value of $e^{5.5}$ converge to five significant figures (meaning that adding the next term no longer changes the 5-digit answer)? Compare this to the value of $e^{5.5}$ computed directly using the built-in exp function of your computing environment (at double precision). What is the magnitude of the relative error incurred in computing $e^{5.5}$ this way using a 5-digit mantissa?

(c) Repeat part (b), but now add the terms within each partial sum from right to left. Did that change the answers to any of the sub-questions in part (b)?

(d) Now approximate $e^{-5.5}$ by working out successive partial sums (you don't need to redo part (a) — just make all the odd-index terms negative). Again, we want the least k for which S_k has converged to five significant figures, and then we want the error relative to computing $e^{-5.5}$ directly using a built-in exponential function. But this time, compute the partial sums in the following ways and compare/contrast the results:

(i) Always add left to right.

(ii) Within each partial sum, add right to left.

(iii) When computing S_k, add all the positive terms contributing to that partial sum left to right, then add all the negative terms left to right, and then combine those results.

(iv) When computing S_k, add all the positive terms contributing to that partial sum right to left, then add all the negative terms right to left, and then combine those results.

Which method converges most quickly? Which has the lowest error? How did this compare to when the argument of the exponential was positive?

(e) Can you think of a way to compute $e^{-5.5}$ more accurately? Validate your proposal.

3 Error propagation in exponentiation

We worked out how (relative) error is propagated by the four basic arithmetic operations $+, -, \times, \div$. But what about exponentiation? It can depend on how exponentiation is implemented…

(a) First let's deal with whole-number exponents. You could imagine computing x^n in a couple of different ways:[2]

(i) Repeatedly multiplying by x

(ii) Computing $e^{n \ln x}$ (in other words, first evaluate the logarithm of x, then repeatedly multiply *that*, and finally raise e to the resulting power).

In each case, derive an upper bound on the relative error resulting from machine arithmetic (this bound may depend on x in either/both scenarios).

In performing your error analysis, recall the ansatz we wrote in class: if "$\circ$" represents one of the four elementary arithmetic operations, and x and y are machine numbers, then we denote the result $\mathrm{fl}(x \circ y)$ of executing that operation with finite precision is a correct machine-number rounding of the exact answer $x * y$, i.e.

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\varepsilon), \quad |\varepsilon| \leq \text{eps} \quad , \tag{4}$$

[2] I'm not saying the computer actually does either of these things exactly — this is just an illustrative exercise — but the machine might do one or the other in different circumstances.

where eps is the machine epsilon. Even though it's not strictly true (and isn't guaranteed by the IEEE standard), assume for simplicity that a single $\ln$ or $\exp$ operation also produces correctly machine-rounded answers when fed a machine number x:

$$\begin{aligned} \text{fl}(\ln x) &= (\ln x)(1+\varepsilon), \quad |\varepsilon| \le \text{eps} \\ \text{fl}(\exp(x)) &= (\exp(x))(1+\varepsilon), \quad |\varepsilon| \le \text{eps} \quad . \end{aligned}$$

Neglect terms of $O(\text{eps}^2)$ or higher in your analysis.

Based on your results, determine a guideline (in terms of x and n) for when exponentiating via repeated multiplication is more accurate than the log-exponential method.

(b) Now let's allow arbitrary exponents. Suppose x is positive and a is nonzero. Determine the propagated relative error ε in x^a when:

(i) x is an exact machine number but a is subject to a relative error ε_a;

(ii) a is an exact machine number but x is subject to a relative error ε_x.

(ε_a and ε_x may be larger than eps in this problem). In each case, express the propagated relative error in the result only in terms of a, x, ε_a and ε_x. Again, neglect quantities that are above first-order in $\varepsilon_a, \varepsilon_x$. When (if ever) could the propagated error in either scenario become substantial?

4 Conditioning

Consider the function

$$f(x) = 1 - e^{-x} \tag{5}$$

on the interval $[0, 1]$.

(a) Determine the condition $(\text{cond}f)(x)$ of this function in terms of x. Show that it is less than 1 everywhere on $[0, 1]$ (in other words, the *problem* of determining $f(x)$ given x is well-conditioned on this interval).

(b) Now suppose x is a machine number (on a computer whose machine epsilon is eps). Let A be the algorithm for computing this function as written. In other words, A consists of the following steps in order:

(i) negate x; then

(ii) compute e^{-x} using the standard math library function for $\exp$; and finally

(iii) subtract that result from 1.

Pretend that step (i) introduces no error (you're just flipping the sign bit) and that each of steps (ii) and (iii) returns a correct rounding of the real answer, just as in Problem 3.

Neglecting quantities of order $O(\text{eps}^2)$ and above, estimate an upper bound on the condition $(\text{cond}A)(x)$ of this algorithm[3] in terms of x. Show that, in contrast to $(\text{cond}f)(x)$, $(\text{cond}A)(x)$ is greater than 1 everywhere on $[0, 1]$.

(c) Plot $(\text{cond}f)(x)$ and $(\text{cond}A)(x)$ vs. x on $[0, 1]$. You should notice that A becomes progressively more ill-conditioned for smaller and smaller x values. What's the root cause of the poor conditioning?

(d) What is the least value of x that ensures at most 1 bit of significance is lost while evaluating $f_A(x)$? What if we are willing to lose 2 bits of significance? 3 bits? 4 bits?

[3]Remember, this involves identifying the result $f_A(x)$ of these operations on x with the result $f(x_A)$ of having done exact arithmetic on some alternate input x_A, and then estimating the relative error between x_A and x as a multiple of eps.

(e) Estimate an upper bound on the relative error in the output (i.e. the forward error) starting from each of the x values computed in part (d) as input.

(f) Since the underlying math problem is not ill-conditioned for small x, an alternate algorithm may give us more accuracy in that regime. Any ideas?

5 Limits in $\mathbb{R}(p,q)$

The base e of the natural logarithm can be defined analytically by the limit

$$e \equiv \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n \quad . \tag{6}$$

This limit defines a sequence, namely the values of $(1 + 1/n)^n$ for $n = 1, 2, 3, \ldots$. Consider the subsequence of entries for which n is an integer power of 10 ($n = 1, 10, 10^2, 10^3, \ldots$). Write a short piece of code that computes entries in this sub-sequence and stops once two successive terms agree to 12 significant figures. Your code should output:

- the value n_{stop} of n for which you've achieved this 12-sigfig convergence;
- the final value to which you've converged;
- a table of all the intermediate values computed for $0 \le n \le n_{\text{stop}}$.

Submit both your code and an explanation that accounts for why your code converged to the value it did.

6 Fun with square roots

Here's a fun and illuminating programming exercise (probably easiest to do in MATLAB or Numpy since elementary functions are broadcast elementwise to array elements, but feel free to code this in C/C++ if that's your preference). Do everything below in double precision.

Make an array x of 1001 floats **from 1.0 to 10.0, inclusive** (easiest to do with the MATLAB/Numpy `linspace` or `logspace` functions – it doesn't matter whether the array entries are linearly or logarithmically spaced on the interval $[0, 1]$). Now write a loop to successively square-root that array 52 times. Then write another loop to successively square that result 52 times. In a perfect world, we'd end up with the same array we started with.

But we don't. Plot the resulting array values, along with a dashed $y = x$ line on the plot for reference. Surprised?

Your task is to account for what's going on. Open-ended freestyle exploration, but you do need to give a substantive account of what's happened, along with any supporting exploratory code, plots, written analysis, and other evidence. You'll notice in particular that there seem to be a couple of values for which the final result *is* pretty much the value you started with. What values are those, and why are they the ones left intact?

If you want to gain some insight into what might be happeneing, I recommend repeating the problem, only instead of 52 iterations of square-rooting and squaring, see what happens for 53 iterations, 54 iterations, 49 iterations, and 50 iterations.

This problem is super fun and enlightening if you've never done it before. I hope you enjoy it.

7 The issue with polynomial roots

I mentioned in class that the problem of finding the roots $(r_1, r_2, \ldots, r_n)$ of a degree-n polynomial given its $n + 1$ coefficients $(a_0, a_1, a_2, \ldots, a_n)$ is, in general, ill-conditioned. This was first realized by Wilkinson in the mid-20th century, somewhat by accident. Let's see the phenomenon firsthand and then try to account for it.

We will work with Wilkinson's polynomial

$$w(x) \equiv \prod_{k=1}^{20}(x-k) = (x-1)(x-2)\cdots(x-20) \quad . \tag{7}$$

(a) Multiply out the factors to get $w(x)$ in the form

$$w(x) = \sum_{n=0}^{20} a_n x^n = x^{20} - 210x^{19} + 20615x^{18}\ldots \quad . \tag{8}$$

Work out all the (integer) coefficients exactly. It's ok to use a computer for this; just make sure you have the coefficients exactly.

(b) Now store the coefficients as (double-precision) floats and generate a function that evaluates $w(x)$ for a a given x. You can code one yourself if you like[4] or use any built-in polynomial library of your choice (I'm not familiar with MATLAB, but Numpy provides a Polynomial class in the `numpy.polynomial` package). Grab any off-the-shelf root-finder based on the Newton-Raphson method (we haven't discussed root-finding yet, but that's ok — the details of the root-finding method aren't the main issue here). Put in an initial guess of 21, and see what you get out as the root. Remember, the roots of this polynomial are just the integers 1 through 20, and 21 is closest to 20, so you might expect the result to converge to 20. Does it?

I believe Numpy's Polynomial objects have a builtin `root()` method. See what that gives you for the largest root and whether it's any better.

(c) Now change the coefficient a_{20} from 1 to $1+\delta$, where $\delta = 10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}$ and repeat part (b) for each of those δ values. What happens to the largest root?

(d) Lest you think that only the highest degree coefficient is compromising us, change a_{20} back to 1 and change a_{19} from -210 to $-210-2^{-23}$. This should be an exact machine number at double precision, so there's no roundoff issue in the coefficient. What has happened to the roots 16 and 17?

(e) Let's get some insight into what we've seen above by doing a restricted-case error analysis. Imagine a general *monic* degree-n polynomial

$$p(x) = \sum_{k=0}^{n} a_k x^k = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1}x^{n-1} + x^n \tag{9}$$

(we've restricted attention to monic polynomials with $a_n \equiv 1$, so we have only n coefficients to specify). Let Ω denote the map from the space of the n free coefficients of $p(x)$ to the space of its n roots:

$$\Omega = \Omega(a_0, a_1, \ldots, a_{n-1}), \quad \Omega_k = \text{the } k\text{th root of } p(x),\ k = 1, 2, \ldots, n \quad . \tag{10}$$

As in class, we can define a condition matrix $\Gamma_{k\ell}$ as the condition number for the kth root given changes in the ℓth coefficient. For some particular root Ω_k, we get a condition vector (fixed k, but ℓ runs from 0 to $n-1$). Let's put an L_1 norm on this vector (sum of the absolute values of the elements) to collapse the condition vector to a single condition number

$$(\operatorname{cond}\Omega_k)(\vec{a}) \equiv \sum_{\ell=0}^{n-1}(\Gamma_{k\ell})(\vec{a}) \quad . \tag{11}$$

(i) Find a concise expression (written as a summation with some pre-factor) for $(\operatorname{cond}\Omega_k)(\vec{a})$ in terms of $\Omega_k, p'(\Omega_k)$, and the coefficients a_ℓ.

[4]But be thoughtful about how you code it. Look up something called *Horner's algorithm.*

(ii) For the Wilkinson polynomial $w(x)$, evaluate the condition numbers for the roots $r = 14, 16, 17, 20$. Discuss your results.

(iii) Could a sufficiently clever algorithm help us here? Why or why not?

8 Recurrence in reverse

In class, we saw that elements in the sequence of integrals

$$y_n \equiv \int_0^1 dx\, x^n e^x \quad (n \geq 0) \tag{12}$$

satisfy a recurrence relation

$$y_{n+1} = e - (n+1)y_n \tag{13}$$

whose terms progressively magnify any relative error present in earlier terms. We suggested that reversing the recurrence relation (i.e. solving it for y_n in terms of y_{n+1}) could avoid this growth in relative error: to compute some particular value y_k, we could begin with some known value $y_N, N > k$ and work *down* to the value of y_k. This raised two questions:

(i) How much bigger than k does N need to be to constrain the relative error in y_k to some desired tolerance ε?

(ii) The reason we introduced the recurrence relation (13) in the first place is that we were pretending that the integral (12) was super difficult or expensive to evaluate directly. So how are we supposed to know the seed value y_N for running the recurrence relation in reverse?

In this problem, we're going to answer question (i) and, in the process, realize that the answer to question (ii) is "We don't need to know y_N at all!"

(a) Reverse the recurrence relation (13). Imagine this as a map g_k from y_N to $y_k, k < N$. Establish an upper limit on $(\text{cond}\, g_k)(y_N))$ in terms of k and N.

(b) The condition number relates the relative error in y_k to the relative error in y_N. Suppose we want a relative error of ε in y_k and just accept a 100% relative error in y_N, which is tantamount to taking $y_N^* = 0$ as the (very wrong) starting value of y_N. In terms of ε and k, determine the minimum value of N need to achieve the target relative error in y_k.

(c) Take $\varepsilon = \text{eps}$ (the machine epsilon) and take $k = 20$. Determine the value of N needed to achieve this error in y_{20}.

(d) Write a short piece of code (a loop) to compute y_{20} using the backward recurrence relation from part (a) starting from a seed value of 0 at the N value you found in part (c). Compare this to what you get by computing y_{20} directly using one or two off the shelf integration routines (and/or Wolfram Alpha).

Analogs of this trick of working with backward recurrence relations to achieve high accuracy features prominently in some methods for numerically evaluating certain special functions from physics, e.g. Bessel functions.