APC 523: Numerical Algorithms for Scientific Computing
Homework 1

Zachary Hervieux-Moore

Wednesday 13th March, 2019

Exercise 1: Error in (symmetric) rounding vs. chopping

Answer: We will show that symmetric rounding satisfies the following bound by considering two cases:

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| \leq 2^{-p}$$

First, we consider the case where the discarded bit is 0. For example, $x = 0.0011\mathbf{01...}$ where we are rounding after the 4th decimal point. We have

$$x - \mathrm{rd}(x) = \pm\left(\sum_{i=p+1}^{\infty} b_i 2^{-i}\right) 2^e$$

But, by assumption, $b_{p+1} = 0$, so

$$x - \mathrm{rd}(x) = \left(\sum_{i=p+2}^{\infty} b_i 2^{-i}\right) 2^e$$

Then, maximizing this expression by picking $b_i = 1$ for the remaining b_i's and minimizing x by picking $x = 0.1000...$ as done in class, yields the bound

$$\left|\frac{x - \mathrm{rd}(x)}{x}\right| \leq 2^{-(p+1)} \cdot 2 = 2^{-p}$$

The second case is to consider the discarded bit is 1. For example, $x = 0.1011\mathbf{10...}$, where again we are rounding after the 4th decimal. In this case, $\mathrm{rd}(x) = 0.1100$ as we are rounding up. Note, that this means that x and $\mathrm{rd}(x)$ will be equal, up to say decimal j, where $\mathrm{rd}(x)$ will be 1 at j, x will be 0, then $\mathrm{rd}(x)$ will be 0 for all decimals greater than j and x will be 1. In the example given, $j = 2$. Notice in the example, $\mathrm{rd}(x) - x = 0.0000\mathbf{01...}$, the carryover from the rounding up causes the first non zero bit to be located at decimal $p + 2$. This carryover phenomena holds in the general case. So we get

$$\mathrm{rd}(x) - x = \left(\sum_{i=p+2}^{\infty} b_i 2^{-i}\right) 2^e$$

Similar to as before, maximizing and picking the worse case for x yields the bound

$$\left|\frac{\mathrm{rd}(x) - x}{x}\right| = \left|\frac{x - \mathrm{rd}(x)}{x}\right| \leq 2^{-(p+1)} \cdot 2 = 2^{-p}$$

Exercise 2: Accurate implementation of e^x

Answer: The associated tables and numbers are in the attached Jupyter notebook titled `question2.ipynb`. The written answers are here.

a) See notebook for table of terms

b) The value of k was 18 to converge to 5 significant figures to 244.71. The relative error with the built-in function was 7.384e-5.

c) The value of k was 17 to converge to 5 significant figures to 244.69. The relative error with the built-in function was 7.897e-6. The answers did change, this method is more accurate by roughly an order of magnitude and converges faster. This is due to the fact that adding the smaller terms first can yield more accurate rounding of the smaller digits since you are rounding even smaller digits first whereas adding small numbers to a large number simply truncates them.

d) See notebook for implementations. The method that converges the quickest with the lowest error is algorithm ii). This is the same as the positive exponent but is 3 orders of magnitude of accuracy worse because subtraction adds more error than simply adding:

   i) Left to Right: The value of k was 26 to converge to 5 significant figures to 3.8363e-3. The relative error with the built-in function was 0.0613.

   ii) Right to Left: The value of k was 20 to converge to 5 significant figures to 4.0000e-3. The relative error with the built-in function was 0.0212. Note that this is better than the previous version which coincides with parts b) and c).

   iii) Left to Right, Positive then Negative: The value of k was 19 to converge to 5 significant figures to 0.0000e0. The relative error with the built-in function was 1.0. Note that this terrible performance is due to the positive and negative term canceling each other out when combined.

   iv) Right to Left, Positive then Negative: The value converges to 5 significant figures to 1.0000e-2. The relative error with the built-in function was 1.447. This is the worst performance and again is caused by the cancellation of the positive and negative part with a slight mismatch due to higher precision from adding right to left.

e) The simplest way is to simply invert our super accurate answer for e from part c). That is $e^{-5.5} = \frac{1}{e^{5.5}}$. This yields a value of 4.0868e-3 and a relative error of 6.989e-6.

Exercise 3: Error propagation in exponentiation

Answer: In everything that follows, I assumed that ϵ for each operation was unique.

a) i) We are simply doing repeated multiplication so we get the following:

$$\begin{aligned} x^n &= x^n(1+\epsilon)\cdot \ldots \cdot (1+\epsilon) = x^n(1+\epsilon)^n \\ &\approx x^n(1+n\epsilon) \end{aligned}$$

ii) This calculation is easy to do in the order of the operation:

$$\begin{aligned} &\ln(x) = \ln(x)(1+\epsilon) \\ \implies &n\ln(x) = n\ln(x)(1+\epsilon)(1+\epsilon) \\ &\approx n\ln(x)(1+2\epsilon) \\ \implies &e^{n\ln(x)} = e^{n\ln(x)(1+2\epsilon)}(1+\epsilon) \\ &\approx e^{n\ln(x)}(1+2\epsilon n\ln(x))(1+\epsilon) \\ &\approx e^{n\ln(x)}(1+2\epsilon n\ln(x)+\epsilon) \end{aligned}$$

We would do option i) if the error is less than option ii) or:

$$\begin{aligned} &n\epsilon \leq 2n\ln(x)\epsilon + \epsilon \\ \implies &e^{\frac{n-1}{2n}} \leq x \end{aligned}$$

b) i) We have the same mechanical steps as before except that $\text{fl}(a) = a(1+\epsilon_a)$. We use the method of $x^a = e^{a\ln(x)}$.

$$\begin{aligned} &\ln(x) = \ln(x)(1+\epsilon) \\ \implies &a\ln(x) = a\ln(x)(1+\epsilon)(1+\epsilon)(1+\epsilon_a) \\ &\approx a\ln(x)(1+2\epsilon+\epsilon_a) \\ \implies &e^{a\ln(x)} = e^{a\ln(x)(1+2\epsilon+\epsilon_a)}(1+\epsilon) \\ &\approx e^{a\ln(x)}(1+2\epsilon a\ln(x)+\epsilon_a a\ln(x)+\epsilon) \end{aligned}$$

ii) This time we have $\text{fl}(x) = x(1+\epsilon_x)$ but the steps are similar.

$$\begin{aligned}
\ln(x) &= \ln(x(1+\epsilon_x))(1+\epsilon) = (\ln(x) + \ln(1+\epsilon_x))(1+\epsilon) \\
&\approx (\ln(x) + \epsilon_x))(1+\epsilon) = \ln(x)(1+\epsilon) + \epsilon_x \\
\implies a\ln(x) &= a\ln(x)(1+\epsilon)(1+\epsilon) + \epsilon_x a(1+\epsilon) \\
&\approx a\ln(x)(1+2\epsilon) + a\epsilon_x \\
\implies e^{a\ln(x)} &= e^{a\ln(x)(1+2\epsilon)+a\epsilon_x}(1+\epsilon) \\
&\approx e^{a\ln(x)}(1 + 2\epsilon a\ln(x) + \epsilon_x a + \epsilon)
\end{aligned}$$

The problems occur when $x \approx 0$ because the logarithmic term will go to negative infinity. Or, in the second case, if a is large, then we don't have the option of $\ln(x)$ canceling it out if $x \approx 1$.

Exercise 4: Conditioning

Answer: We have that $f(x) = 1 - e^{-x}$.

a) The conditioning number on $(0, 1]$ is as follows:

$$(\text{cond}f)(x) = \left|\frac{xf'(x)}{f(x)}\right| = \left|\frac{xe^{-x}}{1-e^{-x}}\right| = \frac{x}{e^x - 1}$$

From this, it is easy to see that $(\text{cond}f)(x) < 1$ on $(0, 1]$ by using standard e^x inequalities or doing the Taylor series for the denominator.

$$(\text{cond}f)(x) = \frac{x}{x + O(x^2)} < 1$$

b) We reproduce similar steps as question 3. We also use the result of error propagation of subtraction from class. We have

$$\begin{aligned}
-x &= -x(1) \\
\implies e^{-x} &= e^{-x}(1+\epsilon) \\
\implies 1 - e^{-x} &= (1 - e^{-x})(1 + \frac{e^{-x}}{1-e^{-x}}\epsilon) \\
&\approx (1 - e^{-x})(1 + \frac{e^{-x}}{1-e^{-x}}\epsilon)
\end{aligned}$$

Thus, we make our ansatz and use standard first order approximations

$$\begin{aligned}
1 - e^{-x_A} &= (1 - e^{-x})(1 + \frac{1}{e^x - 1}\epsilon) \\
\implies \ln(1 - e^{-x_A}) &= \ln(1 - e^{-x}) + \ln(1 + \frac{1}{e^x - 1}\epsilon) \\
\implies -e^{-x_A} &= -e^{-x} + \frac{e^{-x}}{1 - e^{-x}}\epsilon \\
\implies -x_A &= -x + \ln(1 - \frac{1}{1-e^{-x}}\epsilon) \\
\implies \left|\frac{x_A - x}{x}\right| &= \frac{1}{x(1-e^{-x})}\epsilon = \frac{e^x}{x(e^x - 1)}\epsilon
\end{aligned}$$

Thus, our condition number is

$$(\text{cond}A)(x) = \frac{e^x}{x(e^x - 1)}$$

From here, we have a monotonically decreasing function that is greater than 1 when we let $x = 1$, thus it is greater than 1 for $x \in [0, 1]$.

c) The graph is attached below. (cond $A)(x)$ is very poorly conditioned near $x = 0$ because the denominator of the term in front of ϵ_{exp} goes to 0 while the numerator goes to 1, this leads to an exponential rise to infinity. The root cause of this is the error introduced during the subtraction because 1 and e^{-x} become very close to each other when x get closer to 0.

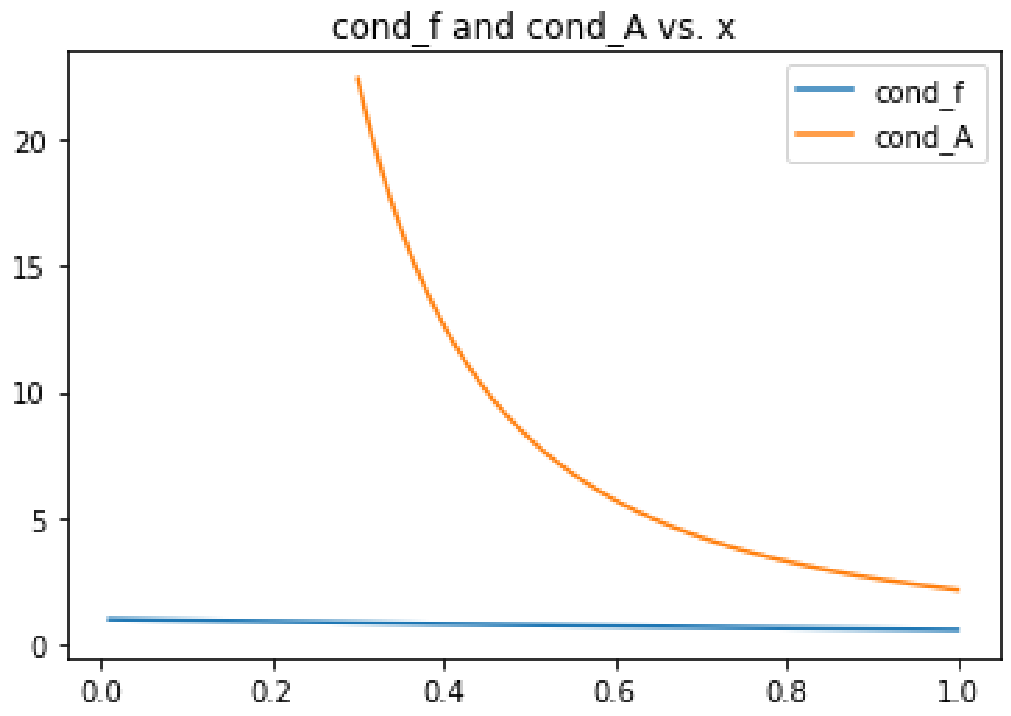

Figure 1: Plot of (cond $f)(x)$ and (cond $A)(x)$

d) Using our previous expression from part b), we are trying to find errors that cause the $\frac{1}{e^x-1} \leq 2^b$ where b is the desired bits. Solving for x yields the following table.

b	x
1	$\ln(3) - \ln(2)$
2	$\ln(5) - \ln(4)$
3	$2\ln(9) - \ln(8)$
4	$\ln(17) - \ln(16)$

e) Now we simply plug this values into the error propagation term

$$\frac{1}{e^x - 1}\epsilon$$

Where $\epsilon = 2^{-16}$, thus we get the following

x	$error$
$\ln(3) - \ln(2)$	3.051e-5
$\ln(5) - \ln(4)$	6.103e-5
$2\ln(9) - \ln(8)$	1.221e-4
$\ln(17) - \ln(16)$	2.441e-4

f) As part a) shows, the underlying problem is not ill-conditioned. We can hope that a better algorithm can give us more accuracy. For this, we consider the Taylor series expansion.

$$1 - e^{-x} = -\sum_{k=1}^{\infty} \frac{-x^k}{k!}$$

We then use Horner's method to reduce the number of multiplications:

$$1 - e^{-x} = x - \frac{x^2}{2} + \frac{x^3}{6} - \frac{x^4}{24} + O(x^5)$$
$$= x\left(1 + x\left(-\frac{1}{2} + x\left(\frac{1}{6} + x\left(-\frac{1}{24} + x\left(O(x^5)\right)\right)\right)\right)\right)$$

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

Answer: The code used to accomplish this question is found in `question5.ipynb`. The answer to the questions are as follows. $n_{\text{stop}} = 10e16$ where it converged onto the value of 1.0 for the remained of n. The table of intermediate values are in the notebook. The explanation for why it converges to 1 is as follows. $10^16\ 2^53$. As we recall, there are 52 bits in the mantissa which means when we do $1/n$ with such a large n, it gets rounded down to 0. Thus, we are left with 1^n which is of course 1.

Exercise 6: Fun with square roots

Answer: Note, there are accompanying plots in the notebook `question6.ipynb`. First, we note the staircase patterns of the numbers. At iteration 54, it becomes a line at 1. Again, this is due to the mantissa having only 52 bits, so doing 54 square roots will force any floating point number to 1. At which point any power returns 1. But why the staircase pattern? If you look through my plots, you can see as we vary the number of iterations from 45 to 54, the staircase gets steeper and the steps are not evenly spaced. That is, the horizontal of the step grows logarithmically. Again, this is due to only having a fixed number of points to represent the real number continuum. Floating point numbers are distributed logarithmically to keep relative error consistent. That is, when we round, we'd prefer the relative error of rounding small numbers to be equal to rounding large numbers.

One thing to notice is that the blue lines are always below the orange (the true) value of doing these successive square roots and squares. It may be tempting to think that some numbers get rounded up after square rooting and so the result squares should be larger than the true value. However, as square roots are irrational numbers, we calculate them using Taylor series approximations. Thus, we can always ensure that are floating point numbers match however many digits we can fit. As a result, we are always subject to truncation error which always results in an underestimate of the true value when we reapply the powers of 2.

The final interesting thing to note is why certain is why certain values are extremely close to their true values. For example, 2.719 and 7.3891. This is because $2.719^{1/2^{52}} = 1.0000000000000002$ after rounding and $7.3891^{1/2^{52}} = 1.0000000000000004$ which are the next two smallest numbers larger than 1 in floating point numbers. To reiterate, because there is no machine number between these numbers, then every value from 2.719 to 7.3891 gets truncated to 1.0000000000000002. This is why the stair case pattern is formed.

Exercise 7: The issue with polynomial roots

Answer: Most of the answers associated with this question can be found in the Jupyter notebook `question7.ipynb`.

a) The integer coefficients are listed in the first cell of the notebook.

b) Code in the notebook. The Netwon-Raphson method returns 19.999995 with an initial guess of 21. I would classify this as convergent. However, Numpy's built in `roots()` function return 20.00054 which is different by a non trivial amount. It is also worse than the other method.

c) Code in the notebook. For $\delta = 10^{-8}$, it converges to 9.585. For $\delta = 10^{-6}$, it converges to 7.752. For $\delta = 10^{-4}$, it converges to 5.969. For $\delta = 10^{-2}$, it converges to 5.470. These are clearly huge deviations from 20 even though the pertubations are small.

d) Code in the notebook. With this change, roots 16 and 17 disappear completely and become complex. In fact, 10 out of the 20 roots become complex with a pertubation of $\delta = 2^{-23}$.

e) To find the conditioning number, we first not that a deviation in a_i of δ_{a_i} will yield a deviation in Ω_k of δ_{Ω_k} that satisfies the following

$$p_{a_i+\delta_{a_i}}(\Omega_k + \delta_{\Omega_k}) = 0$$

Where $p_{a_i+\delta_{a_i}}$ denotes the shifted polynomial. We write out this complete polynomial

$$\begin{aligned} p_{a_i+\delta_{a_i}}(\Omega_k + \delta_{\Omega_k}) &= a_0 + a_1(\Omega_k + \delta_{\Omega_k}) + \dots \\ &+ (a_i + \delta_{a_i})(\Omega_k + \delta_{\Omega_k})^i + \dots + (\Omega_k + \delta_{\Omega_k})^n \end{aligned}$$

Which we simplify and do a first order approximation on

$$= a_0 + a_1\Omega_k(1 + \frac{\delta_{\Omega_k}}{\Omega_k}) + \dots + (a_i + \delta_{a_i})\Omega_k^i(1 + i\frac{\delta_{\Omega_k}}{\Omega_k}) + \dots + \Omega_k^n(1 + n\frac{\delta_{\Omega_k}}{\Omega_k})$$

Now we rearrange this into a different form for clarity

$$\begin{aligned}
&= a_0 + a_1(\Omega_k + \Omega_k\delta_{\Omega_k}) \\
&\quad + a_2(\Omega_k^2 + 2\Omega_k\delta_{\Omega_k}) \\
&\quad + a_3(\Omega_k^3 + 3\Omega_k^2\delta_{\Omega_k}) \\
&\quad + \dots \\
&\quad + (a_i + \delta_{a_i})(\Omega_k^i + i\Omega_k^{i-1}\delta_{\Omega_k}) \\
&\quad + \dots \\
&\quad + (\Omega_k^n + n\Omega_k^{n-1}\delta_{\Omega_k})
\end{aligned}$$

Which, we notice that all the terms infront of δ_{Ω_k} is precisely $p'(\Omega_k$. Thus we have that

$$\begin{aligned}
&\delta_{a_i}\Omega_k^i + p'(\Omega_k)\delta_{\Omega_k} = 0 \\
\implies &\frac{\partial\Omega_k}{\partial a_i} = -\frac{\Omega_k^i}{p'(\Omega_k)}
\end{aligned}$$

So we conclude that the conditioning for the a_i is

$$(\text{cond}\Omega_k)(a_i) = \left|\frac{a_i\frac{\partial\Omega_k}{\partial a_i}}{\Omega_k}\right| = \left|\frac{a_i\Omega_k^i}{\Omega_k p'(\Omega_k)}\right|$$

Which of course is only part of the total condition and so our answer is

i)

$$(\text{cond}\Omega_k)(\overrightarrow{a}) = \frac{1}{|\Omega_k p'(\Omega_k)|}\sum_i |a_i\Omega_k^i|$$

ii) Code in the notebook. But the resulting condition numbers are:

$$\begin{aligned}
(\text{cond}\Omega_{14})(\overrightarrow{a}) &= 7.193916e+13 \\
(\text{cond}\Omega_{16})(\overrightarrow{a}) &= 3.044400e+13 \\
(\text{cond}\Omega_{17})(\overrightarrow{a}) &= 2.593885e+13 \\
(\text{cond}\Omega_{20})(\overrightarrow{a}) &= 1.373204e+11
\end{aligned}$$

iii) The condition number is incredibly high, so any small change in the inputs will cause huge changes to the outputs. This makes a sufficiently clever algorithm non-existent.

Exercise 8: Recurrence in reverse

Answer:

a) The first reversal is

$$y_k = \frac{e}{k+1} - \frac{1}{k+1} y_{k+1}$$

Expanding these yields,

$$\begin{aligned} y_k &= \frac{e}{k+1} - \frac{1}{k+1}(\frac{e}{k+2} - \frac{1}{k+2} y_{k+2}) \\ &= \frac{e}{k+1} - \frac{e}{(k+1)(k+2)} + \frac{1}{(k+1)(k+2)} y_{k+2}) \\ &\vdots \\ &= e \sum_{j=1}^{N-k} \frac{(-1)^{j+1}}{\prod_{i=1}^{j}(k+i)} + (-1)^{N-k} \frac{y_N}{\prod_{i=1}^{N-k}(k+i)} \end{aligned}$$

b) We use the definition of condition,

$$(\text{cond} g_k)(y_N) = \left| \frac{g_k' y_N}{y_k} \right| = g_k' \left| \frac{y_N}{y_k} \right|$$

The sequence is monotonic so we know that $\left| \frac{y_N}{y_k} \right| \leq 1$. ee also sub in the derivative of g_k to get

$$(\text{cond} g_k)(y_N) = \frac{1}{\prod_{i=1}^{N-k}(k+i)} = \frac{k!}{N!}$$

Now, we know that the condition number relates the input error and output error as follows

$$(\text{cond} g_k)(y_N) \epsilon_{y_N} = \epsilon_{y_k}$$

But, by assumption, we have $\epsilon_{y_N} = 1$, So we get

$$\epsilon_{y_k} = \frac{k!}{N!}$$

Solving for N (using Gamma functions) will give the value of N required to get the desired error.

c) We take $\epsilon_{y_k} = 2e^{-16}$ and $k = 20$, our previous part says that we need a value of $N = 32$ to get the error desired.

d) Code attached in notebook `question8.ipynb`. The value we find is within the eps of the machine when compared to Wolfram Alpha.