AMath 584, Autumn 2017

Sample Solutions to Assignment 5.

Reading: Lectures 12-15 in the text.

1. p. 96, Exercise 12.1.

   The 2-norm condition number $\kappa(A)$ is the ratio of the largest to smallest singular value of A. The largest singular value is $\sigma_1 = \|A\|_2 = 100$. The sum of squares of all singular values, $\sum_{i=1}^{202} \sigma_i^2 = 100^2 + \sum_{i=2}^{202} \sigma_i^2$, is the square of the Frobenius norm, namely, $101^2 = 100^2 + 201$. It follows that the smallest singular value σ_{202} must be less than or equal to 1, and it could be equal to 1 if all of the other singular values $\sigma_2, \ldots, \sigma_{201}$ were equal to 1 also. Thus, the best possible lower bound on $\kappa(A)$ is $\kappa(A) \geq 100$.

2. What is the gap between 2 and the next larger double precision number? What is the gap between 201 and the next larger double precision number? How many IEEE double precision numbers are there between an adjacent pair of nonzero IEEE single precision numbers?

   The binary floating point representation of 2 is $1.0_2 \times 2^1$. Therefore the next larger double precision floating point number is $(1 + 2^{-52}) \times 2^1$, and the gap is 2^{-51}. The binary floating point representation of 201 is $1.1001001_2 \times 2^7$. Therefore the gap between this and the next larger double preccision floating point number is $2^{-52} \times 2^7 = 2^{-45}$.

   A single precision word has 23 bits for the mantissa; a double precision word has 52 bits for the mantissa. Between two adjacent single precision numbers, one with a 0 in the 23rd bit and one with a 1 in the 23rd bit, you can take a double precision word that matches the smaller number in the first 23 bits and has either 0 or 1 in each of the reamining 29 bits. If these extra bits are all 0, then this is equal to the single precision number, but otherwise it is strictly between the pair. Thus there are $2^{29} - 1$ double precision numbers strictly between a pair of adjacent single precision numbers.

3. In the 1991 Gulf War, the Patriot missile defense system failed due to roundoff error. The troubles stemmed from a computer that performed the tracking calculations with an internal clock whose integer values in tenths of a second were converted to seconds by multiplying by a 24-bit binary approximation to one tenth:

$$0.1_{10} \approx 0.00011001100110011001100_2. \tag{1}$$

   (a) Convert the binary number in (1) to a fraction. Call it x.

$2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} = 209715/2097152$

(b) What is the absolute error in this number? That is, what is the absolute value of the difference between x and $\frac{1}{10}$?

9.5×10^{-8}.

(c) What is the time error in seconds after 100 hours of operation (i.e., the value of $|360,000 - 3,600,000x|$)?

0.3433.

(d) During the 1991 war, a Scud missile traveled at approximately Mach 5 (3750 mph). Find the distance that a Scud missile would travel during the time error computed in (c).

$(3750/3600) * 0.3433 \approx 0.36$ miles.

On February 25, 1991, a Patriot battery system, which was to protect the Dhahran Air Base, had been operating for over 100 consecutive hours. The roundoff error caused the system not to track an incoming Scud missile, which slipped through the defense system and detonated on US Army barracks, killing 28 American soldiers.

4. (a) Determine the absolute and relative condition numbers for the problem of evaluating $f(x) = e^x$. Would you say that this problem is well-conditioned or ill-conditioned, say, for $x = -20$?

Since $f'(x) = e^x$, the absolute condition number is e^x and the relative condition number is $|x|$. For $x = -20$, this problem is well-conditioned in both the absolute and relative sense.

(b) The following Matlab code uses a Taylor series to approximate e^x:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots .$$

```
oldsum = 0;
newsum = 1;                  % First term in series.
term = 1;
n = 0;
while newsum ~= oldsum       % Iterate until next term is negligible.
  n = n + 1;
  term = term * x/n;         % This is x^n / n!
  oldsum = newsum;
  newsum = newsum + term;
end;
```

This code adds terms in the Taylor series until the next term is so small that adding it to the current sum makes no change in the floating point number that is stored. The code works fine for $x > 0$. Try it for a few positive values of x

and compare your results with `exp(x)` computed in Matlab to convince yourself that the values are accurate. (Use `format long e` to print out the values to 16 decimal places, or look at the difference between the value you computed and that returned by `exp(x)`.) Now try the code for $x = -20$ and compare your result with that returned by `exp(-20)`. You should see a large relative error. Explain this inaccuracy using the fact that floating point arithmetic satisfies $x \oplus y = (x + y)(1 + \epsilon)$, $x \ominus y = (x - y)(1 + \epsilon)$, etc., where $|\epsilon|$ is less than or equal to the machine precision. [Hint: Look at the size of intermediate sums.]

For $x = -20$, the code computed $5.6219e - 09$, while `exp(-20)` returned $2.0612e - 09$. The absolute error is on the order of $3.e - 9$, but the relative error is about 1.7. This is in contrast to, say, $x = 20$, where the numbers returned by this code and by `exp(20)` differed only in the 16th decimal place and the relative error was about $1.2e - 16$. One can see what is happening by removing the semicolon after the expressions for `term` and `newsum` and observing their intermediate values. The size of the terms in the series increases to about $4.3e + 7$ before starting to decrease. This results in double precision rounding errors of size about $4.3 \times 10^7 \times 10^{-16} \approx 4.3e - 9$, which lead to completely wrong answers when the true solution is on the order of 10^{-9} or less.

Would you say that this algorithm is stable but not backward stable, backward stable, or unstable (in a relative sense) when the input x is negative? Explain your answer.

If we fix $x = -20$ and let $\epsilon_{machine}$ get smaller and smaller, we will get an answer that is accurate to $O(\epsilon_{machine})$, but with a large constant of about 10^{16} in front of $\epsilon_{machine}$. But if we let x become more and more negative as $\epsilon_{machine} \to 0$, we will continue to get inaccurate answers. Therefore, the algorithm is unstable.

(c) How could you modify this code to work better for negative values of x.

Evaluate $e^{|x|}$ and take one over this result if x is negative.

5. *Numerical differentiation.* One can approximate the derivative of a function $f(x)$ by

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}. \tag{2}$$

Since, using Taylor's theorem with remainder,

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2} f''(\xi), \quad \xi \in [x, x+h],$$

the *truncation error* in approximation (2) is $O(h)$:

$$\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| = \frac{h}{2} \left| f''(\xi) \right|$$

(a) Let $f(x) = \sin(x)$ and $x = \frac{\pi}{3}$. Is the problem of computing $f'(x) = \cos(x)$ well-conditioned or ill-conditioned?

Since $f''(x) = -\sin(x) = -\sqrt{3}/2$, the absolute condition number is quite moderate, and since $xf''(x)/f'(x) = -\frac{\pi}{3}\tan\left(\frac{\pi}{3}\right) = -\pi\sqrt{3}/3$, the relative condition number is also moderate. Thus the problem is well-conditioned.

(b) Try using (2) in Matlab to approximate $f'(x)$ where $f(x) = \sin(x)$ and $x = \pi/3$. Take $h = 1.e-1, 1.e-2, \ldots, 1.e-16$ and make a table of your results and the difference between the computed results and the true value of $f'(x)$, namely, $\cos(\pi/3) = 0.5$.

Following are the results that I got:

```
     h            (sin(x+h)-sin(x))/h              cos(x) -
                                              (sin(x+h)-sin(x))/h


  1.0e-01      4.559018854107610e-01      4.409811458923896e-02
  1.0e-02      4.956615757736871e-01      4.338424226312920e-03
  1.0e-03      4.995669040007700e-01      4.330959992300265e-04
  1.0e-04      4.999566978958203e-01      4.330210417968772e-05
  1.0e-05      4.999956698670261e-01      4.330132973906498e-06
  1.0e-06      4.999995669718871e-01      4.330281129227842e-07
  1.0e-07      4.999999569932356e-01      4.300676437196671e-08
  1.0e-08      4.999999969612645e-01      3.038735485461075e-09
  1.0e-09      5.000000413701855e-01     -4.137018549954519e-08
  1.0e-10      5.000000413701855e-01     -4.137018549954519e-08
  1.0e-11      5.000000413701855e-01     -4.137018549954519e-08
  1.0e-12      5.000444502911705e-01     -4.445029117050581e-05
  1.0e-13      4.996003610813204e-01      3.996389186795568e-04
  1.0e-14      4.996003610813204e-01      3.996389186795568e-04
  1.0e-15      5.551115123125783e-01     -5.511151231257827e-02
  1.0e-16                          0      5.000000000000000e-01
```

The errors decrease down to about $3.e-9$ as h decreases to $1.0e-8$, but then they remain flat for a while and then start to increase. For $h = 1.0e-16$, the computed result is 0.

(c) Suppose the only rounding errors made are in rounding $f(x+h)$ and $f(x)$, so that the computed values are $f(x+h)(1+\epsilon_1)$ and $f(x)(1+\epsilon_2)$ where $|\epsilon_1|$ and $|\epsilon_2|$ are both less than or equal to the machine precision. By about how much would the computed difference quotient in (2) differ from the exact difference quotient. Use this to explain your results in (b).

$$\frac{f(x+h)(1+\epsilon_1) - f(x)(1+\epsilon_2)}{h} = \frac{f(x+h) - f(x)}{h} + \frac{f(x+h)\epsilon_1 - f(x)\epsilon_2}{h}.$$

Thus, the difference between the actual difference quotient and the computed difference quotient is about $\frac{\epsilon_{machine}}{h}$. Comparing this with the truncation error, which is about h, we see that as h decreases the truncation error decreases but the error due to roundoff increases. We get the smallest total error when

$$\frac{\epsilon_{machine}}{h} \approx h; \quad \text{i.e., } h \approx \sqrt{\epsilon_{machine}},$$

and in this case, the truncation error and the rounding error are each on the order of $\sqrt{\epsilon_{machine}}$. This is the limiting accuracy displayed in the table.

For $h = 1.e - 16$, your computed result was probably 0. Can you explain this?

This is because $1.0e - 16$ is less than the machine precision; hence when you add $\pi/3$ (or the machine representation of this) and $1.0e - 16$, the result is (the machine representation of) $\pi/3$. Hence $\sin(\pi/3 + 1.0e - 16) - \sin(\pi/3)$ evaluates to 0.