AMATH 584 Assignment 5

Tyler Chen

Exercise 12.1

Suppose A is a 202×202 matrix with $\|A\|_2 = 100$ and $\|A\|_F = 101$. Give the sharpest possible lower bound on the 2-norm condition number $\kappa(A)$.

Solution

Write the nonzero singular values of A in descending order as $\sigma_1, \sigma_2, ..., \sigma_{202}$ so that $\sigma_1 \geq \sigma_2 \geq ... \geq \sigma_{202} \geq 0$.

Recall $\|A\|_2 = \sigma_1$ and $\|A\|_F = \sqrt{\sigma_1^2 + ... + \sigma_{202}^2}$.

Then $\sigma_1 = 100$ and $101^2 = \sigma_1^2 + ... + \sigma_m^2 = 100^2 + \sigma_2^2 + ... + \sigma_{202}^2$.

We have $\kappa_2(A) = \sigma_1/\sigma_{202} = 100/\sigma_{202}$. Since singular values are non-negative, $\kappa_2(A)$ is minimized when σ_{202} is maximized. Therefore, we require $\sigma_2, ..., \sigma_{202}$ as large as possible while still satisfying the above conditions.

Obviously this means $\sigma_2 = ... = \sigma_{202}$ so , $201 = 101^2 - 100^2 = \sigma_2^2 + ... + \sigma_{202}^2 = 201\sigma_{202}^2$ and $\sigma_2 = ... = \sigma_{202} = 1$. Therefore the sharpest lower bound for $\kappa_2(A)$ given $\|A\|_2 = 100$ and $\|A\|_F = 101$ is $\kappa_2(A) = \sigma_1/\sigma_{202} = 100$. This bound is attained for any matrix with $\sigma_1 = 100$ and $\sigma_2 = ... = \sigma_{202} = 1$. Clearly many such matrices exist, for instance the diagonal matrix with 100 in first diagonal entry, and 1 in the rest of the diagonal entries.

This proves the lower bound $\kappa_2(A) \geq 100$ is sharp.

Exercise 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?

Solution

For double precision floating points numbers we have 1 sign bit, 52 mantissa bits, and 11 exponent bits.

So $2 = +1.\underbrace{0\ldots0}_{52\text{ zeros}} \times 2^1$ and the next number is $+1.\underbrace{0\ldots0}_{51\text{ zeros}}\, 1 \times 2^1$. So the gap is $2^{-52} \times 2^1 = 2^{-51}$

Similarly, $201 = +1.1001001\ \underbrace{0\ldots0}_{45\text{ zeros}} \times 2^7$ and the next number is $+1.1001001\ \underbrace{0\ldots0}_{44\text{ zeros}}\, 1 \times 2^7$ so the gap is $2^{-52} \times 2^7 = 2^{-45}$.

Single precision floating point numbers have 1 sign bit, 23 mantissa bits, and 8 exponent bits.

This means double precision numbers have an additional 29 bits. Each of these 29 bits could be zero or 1, as long as all aren't zero. So there are $2^{29} - 1$ numbers between consecutive single precision floating point numbers.

Exercise 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.

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

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

(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).

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.

Solution

I guess this calls for spending a couple billion more on defense.

(a) We convert $0.d_1d_2d_3d_4...$ to a fraction as $d_1/2 + d_2/2^2 + d_3/3^2 + ...$ Thus,

$$\begin{aligned} x &= 0.00011001100110011001100_2 \\ &= \frac{1}{2^3} + \frac{1}{2^4} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{1}{2^{11}} + \frac{1}{2^{12}} + \frac{1}{2^{15}} + \frac{1}{2^{16}} + \frac{1}{2^{19}} + \frac{1}{2^{20}} \\ &= \frac{209715}{2097152} \end{aligned}$$

(b) The absolute error in x and $1/10$ is,

$$\left| \frac{1}{10} - x \right| = \frac{1}{10485760}$$

(c) Similarly the time error in seconds after 100 hours of operation is,

$$|360000 - 3600000x| = \frac{5625}{16384}[\text{s}] \approx 0.3433[\text{s}]$$

(d) The distance an object moving 3750 mph travels during the time error computer above is,

$$\left(\frac{5625}{16384}[\text{s}]\right)\left(3750\frac{[\text{mi}]}{[\text{hr}]}\right)\left(\frac{1}{3600}\frac{[\text{s}]}{[\text{hr}]}\right) = \frac{46875}{131072}[\text{mi}] \approx 0.3576[\text{mi}]$$

Exercise 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 $x = -20$?

(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.]

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.

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

Solution

(a) Since $f(x) = e^x$ is differentiable we can calculate the absolute condition number as,

$$\hat{\kappa}(x) = \|J(x)\| = f'(x) = e^x$$

Similarly, we have relative condition number,

$$\kappa(x) = \frac{\|J(x)\|}{\|f(x)\| \,/\, \|x\|} = \frac{e^x}{e^x/|x|} = |x|\text{‘}$$

So at $x = -20$ we have absolute condition number $\hat{\kappa}(-20) = e^{-20} \approx 2.1 \cdot 10^{-9}$ and relative condition number $\kappa = 20$. Since the relative condition number is on the order of 10^1 we say the problem of evaluating $f(x) = e^x$ is well conditioned.

(b) We implement this in python as,

```
def exercise_4():
    def taylor_exp(x):
        oldsum=0
        newsum=1
        term=1
        n=0
        while newsum != oldsum:
            n=n+1
            term = term*x/n
            oldsum=newsum
            newsum=newsum+term
        return newsum
    for x in [1,5,10,20,-20]:
        print([x,taylor_exp(x),np.exp(x),np.abs(np.exp(x)-taylor_exp(x))/np.
            exp(x)])
```

This gives output,

```
[1, 2.7182818284590455, 2.7182818284590451, 1.6337129034990842e-16]
[5, 148.41315910257654, 148.4131591025766, 3.830079435309396e-16]
[10, 22026.46579480671, 22026.465794806718, 3.3032796463874436e-16]
[20, 485165195.4097902, 485165195.40979028, 1.2285432949295985e-16]
[-20, 5.621884472130418e-09, 2.0611536224385579e-09, 1.7275426784924197]
```

So for positive numbers the difference the relative error is small (on the order of machine epsilon), however for $x = -20$ the relative error is quite large.

If $x < 0$, then successive terms in the taylor expansion have different signs, so the terms "cancel" with each other giving a small final answer. However, the inaccuracies from adding and subtracting do not necessarily cancel. So in the end the absolute error of calculating e^x for negative x may not be much smaller than for positive x, meaning the relative error is much larger.

This hypothesis is supported by the fact that the absolute error between the actual and comped values of e^{-20} is in the magnitude of 10^{-9}, roughly the same as the differences for e^{20}, even though the relative error is far larger.

The algorithm is unstable since the problem is well conditioned, but the computed answer is nowhere near the exact answer, in a relative sense. This means we did not solve a nearby problem exactly, or nearly solve a nearby problem.

(c) If $x < 0$ We can simply compute e^{-x} and then return $1/e^{-x}$.

With $x = -20$ this give,

```
[-20, 2.0611536224385583e-09, 2.0611536224385579e-09, 4.1359030627651384e-25]
```

So the result is far better. This can easily be explained as follows. Suppose we compute $e^{20}(1+\epsilon)$ since calculating positive exponents works fine. Then $1/(e^{20}(1+\epsilon)) = e^{-20}(1-\epsilon+\epsilon^2-...)(1+\epsilon_1) \approx e^{-20}(1+\mathcal{O}(\epsilon_{\text{mach}}))$.

Problem 5

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 $O(h)$:

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

(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?

(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, $\cos(\pi/3) = 0.5$.

(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).

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

Solution

(a) We use $f = \cos(x)$ to denote the problem. The input is $x = \pi/3$. Since f is continuous,

$$\kappa(x) = \frac{\|J(x)\|}{\|f(x)\| \,/\, \|x\|} = \frac{\|-\sin(x)\|}{\|\cos(x)\| \,/\, \|x\|} = x|\tan(x)|$$

Therefore, $\kappa(\pi/3) = \pi/\sqrt{3}$, is on the order of 10^0, the problem is well conditioned.

(b) We use the formula listed to calculate the derivative of $\sin(x)$ at $x = \pi/3$ as,

```
def exercise_5_b():
    x=np.pi/3
    h=np.flip(np.logspace(-16,-1,16),0)
    Df = (np.sin(x+h)-np.sin(x))/h
    print(np.transpose([h,Df,0.5-Df]))

exercise_5_b()
```

This gives output,

```
[[  1.0000000000000001e-01   4.5590188541076104e-01    4.4098114589238957e-02]
 [  1.0000000000000000e-02   4.9566157577368708e-01    4.3384242263129202e-03]
 [  1.0000000000000000e-03   4.9956690400076997e-01    4.3309599923002651e-04]
 [  1.0000000000000000e-04   4.9995669789693054e-01    4.3302103069464692e-05]
```

```
[  1.0000000000000001e-05   4.9999566986702598e-01    4.3301329740175198e-06]
[  9.9999999999999995e-07   4.9999956697188708e-01    4.3302811292278420e-07]
[  9.9999999999999995e-08   4.9999995699323563e-01    4.3006764371966710e-08]
[  1.0000000000000000e-08   4.9999999696126451e-01    3.0387354854610749e-09]
[  1.0000000000000001e-09   5.0000004137018550e-01   -4.1370185499545187e-08]
[  1.0000000000000000e-10   5.0000004137018550e-01   -4.1370185499545187e-08]
[  9.9999999999999994e-12   5.0000004137018550e-01   -4.1370185499545187e-08]
[  9.9999999999999998e-13   5.0004445029117051e-01   -4.4452091170505807e-05]
[  1.0000000000000000e-13   4.9960036108132044e-01    3.9963891867955681e-04]
[  1.0000000000000000e-14   4.9960036108132044e-01    3.9963891867955681e-04]
[  1.0000000000000001e-15   5.5511151231257827e-01   -5.5111512312578270e-02]
[  9.9999999999999998e-17   0.0000000000000000e+00    5.0000000000000000e
   -01]]
```

(c) We assume the only rounding errors are made in the rounding of $f(x+h)$ and $f(x)$. In particular, this means we make the assumption x, $x+h$, and h are floating point numbers, and that we can exactly evaluate the subtraction and division exactly,

With these assumptions in mind we calculate,

$$\begin{aligned}\frac{f(x+h)(1+\epsilon_1)-f(x)(1+\epsilon_2)}{h} &= \frac{f(x+h)+f(x+h)\epsilon_1-(f(x)+f(x)\epsilon_2)}{h}\\ &= \frac{f(x+h)-f(x)}{h}+\frac{f(x+h)\epsilon_1-f(x)\epsilon_2}{h}\end{aligned}$$

Therefore, the difference between the exact difference quotient and the computer quotient is,

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

where $|\epsilon_1|, |\epsilon_2| \leq \epsilon_{\text{mach}}$.

In the worse case we have $\epsilon_1 = \epsilon_{\text{mach}}$ and $\epsilon_2 = -\epsilon_{\text{mach}}$ so the error is,

$$\epsilon_{\text{mach}}\frac{f(x+h)+f(x)}{h}$$

Since $f(x+h) \to f(x)$ as $h \to 0$ then the error goes to $\epsilon_{\text{mach}} 2f(x)/h$, which blows up as h goes to zero.

For larger h the exact difference quotient is a secant line between two "not very near" points x and $x+h$, so it is not a very good approximation of the tangent at x. As $h \to 0$, by definition, the exact difference quotient will go to the derivative. This explains the increase in accuracy as we start to decrease h from $1e-1$ to about $1e-12$.

However, around this point our computed results become less accurate as h continues to decrease as the numerical error becomes dominant. In particular, the total error is like,

$$\frac{h}{2}|f''(\xi)| + \frac{\epsilon_{\text{mach}} 2f(x)}{h}$$

Which is minimized at a constant multiple of $\sqrt{\epsilon_{\text{mach}}}$ which is what we observe.

Eventually, when h is on the order of magnitude of machine precision we have $f(x+h)$ and $f(x)$ round to the same floating point number, so that their difference is zero.