This is a working set of notes for in-class discussion (APPM 4600) on algorithms, some concepts and tools useful to talk about their performance, and particularly, on the concepts of stability and conditioning, which are central to numerical methods for scientific computing, as they allow us to understand if precision is lost (and if so, how much) when running an algorithm or evaluating a mathematical function.

1 Algorithms and Pseudocodes

Given a certain computational task we want to perform, an **algorithm** is a procedure that describes a finite number of steps to be performed in a specific order. A **pseudocode** is a way to describe this algorithm in a clear, concise form. Pseudocodes usually include a description of inputs, outputs and distinct, clear steps (as their name suggests, they may resemble code or a flow diagram).

Example 1: If we want to compute the sum of N given values $x_i \in \mathbb{R}$, a proposed algorithm might simply be to add them sequentially from $i = 1$ to $i = N$, that is, $S = \sum_{i=1}^{N} x_i$. A pseudocode for this algorithm might look as follows:

- **Inputs:** $N, \{x_i\}_{i=1}^{N}$
- **Output:** S
- (1) Set $S = 0$ (Initialize)
- (2) for $i = 1 \dots N$:
- (3) $\quad S = S + x_i$
- (4) Return S

Example 2: Say we want to compute the two distinct, real roots of a quadratic equation given a, b, c. A pseudocode for an improved algorithm might involve deciding which formula to use for each root depending on the sign of b:

- **Inputs:** a, b, c
- **Outputs:** r_1, r_2
- (1) Set $r_1 = r_2 = 0$.
- (2) Compute discriminant $disc = b^2 - 4ac$
- (3) If $disc <= 0$
- (4) $\quad$ print("roots are not real and distinct"), return r_1, r_2.
- (5) Else
- (6) $\quad$ If $b == 0$
- (7) $\quad\quad r_1 = \sqrt{-c/a}, r_2 = -r_1$
- (8) $\quad$ Else If $b > 0$

- (9) $r_1 = \frac{2c}{(-b-\sqrt{disc})}$, $r_2 = \frac{-b-\sqrt{disc}}{2a}$
- (10) Else
- (11) $r_1 = \frac{-b+\sqrt{disc}}{2a}$, $r_2 = \frac{2c}{(-b+\sqrt{disc})}$
- (12) Return r_1, r_2

1.1 What are desirable properties in an algorithm?

If we have a mathematical problem or task in mind, we want the algorithm designed to perform it to be

- **Correct:** We want this algorithm to be mathematically correct; it should produce the right answer for the range of inputs considered.
- **Efficient:** We want our algorithm to not require an unreasonable amount of time, memory or other resources to perform the task at hand. Another way to say this is that we want these costs to not grow too quickly as the number of degrees of freedom (variables) in our problem increases.
- **Stable:** If there is a small change in our inputs (which could be an error due to measurement, rounding, etc), we want the corresponding change in the outputs to be small as well. To be more precise, we care that the relative error in the outputs is not unnecessarily amplified by the algorithm chosen. This property, *when it refers to an algorithm*, is known as **stability**.

1.2 Condition number and stability

To understand the concept of stability further, it is important to distinguish it from the related concept of condition number, or conditioning. To make this very general concept a bit more concrete, we consider a **mathematical problem** to be written in terms of: given input x (which could be a real number, or say, a vector in $\mathbb{R}^n$), evaluates $y = f(x)$ where again y could be a real number, a vector, etc.

Definition 1 *We first define relative errors for the inputs and outputs given that we perturb input x by a small amount Δx:*

$$\text{relerr}_x(x, \Delta x) = \frac{||\Delta x||}{||x||}$$
$$\text{relerr}_f(x, \Delta x) = \frac{||f(x + \Delta x) - f(x)||}{||f(x)||}$$

where $||\cdot||$ is absolute value or a given vector norm. We say that the condition number for function f at input x, denoted as $\kappa_f(x)$, is given by

$$\kappa_f(x) = \lim_{\Delta x \to 0} \frac{\text{relerr}_f(x, \Delta x)}{\text{relerr}_x(x, \Delta x)} = \lim_{\Delta x \to 0} \frac{||f(x + \Delta x) - f(x)||||x||}{||\Delta x||||f(x)||}$$

(if this limit exists). If f is differentiable function from $\mathbb{R}$ to $\mathbb{R}$, this limit simplifies to

$$\kappa_f(x) = |f'(x)| \frac{|x|}{|f(x)|}$$

Intuitively, if we start with inputs with a relative accuracy of 10^{-16}, the condition number tells us that the relative accuracy of the output $f(x)$ will be around $10^{-16}\kappa_f(x)$. In other words, the condition number is a "multiplier" inherent in evaluating the mathematical function $f(x)$ if our inputs are not fully precise. We will call a problem or function *well-conditioned* if $\kappa_f(x)$ is close to 1 for all relevant x. We will say it is *ill-conditioned* if it of considerably large magnitude ($\kappa_f(x) \simeq (1/\varepsilon_{mach}$ is particularly bad, as we lose all precision when evaluating $f(x)$ in that case).

Examples

Let's calculate the condition number for some simple functions $f(x)$:

- $f_1(x) = x + a$: Using the formula for differentiable functions, we get that

  $$\kappa_{f_1}(x) = \frac{|x|}{|x+a|}$$

  We see clearly that this condition number, as a function of x, goes to infinity as $x = -a$, and to 1 as $|x| \to \pm\infty$. This matches what we know about addition and subtraction in terms of loss of precision: the only values that have issues are those where $x + a$ is a subtraction of values close to each other.

- $f_2(x) = ax$: Using the formula again, we get:

  $$\kappa_{f_2}(x) = |a|\frac{|x|}{|ax|} = 1$$

  So, multiplication by a scalar a is always well conditioned.

- $f_3(x) = x^2$:

  $$\kappa_{f_3}(x) = 2|x|\frac{|x|}{|x^2|} = 2$$

- $f_4(x) = e^x$:

  $$\kappa_{f_4}(x) = e^x\frac{|x|}{e^x} = |x|$$

Stability: Now, say we come up with an algorithm to evaluate $y = f(x)$, described by a pseudocode involving a number of steps to transform x into an approximation of $y = f(x)$. We will say that this algorithm is *numerically stable* if it involves a series of steps that do not unnecessarily lose precision (are not unnecesarily badly conditioned). Intuitively, there is no way to improve upon the condition number of $f(x)$, but a stable algorithm will not make the resulting errors in our outputs a lot worse than that.

Let's look at some examples. On each one of these, the underlying mathematical problem is well-conditioned, but some algorithms proposed to solve them lose precision (have steps which themselves are badly conditioned, involve things like subtracting two numbers close to each other).

Quadratic formula

To make things simple, let's say we want to find one of the roots of the quadratic $x^2 + bx - 1$. That is, we want to evaluate the function

$$r_1(b) = \frac{-b + \sqrt{b^2 + 4}}{2}$$

We can then ask what the condition number is as a function of the input b. We know from our previous work that when b is relatively large and positive, this formula introduces quite a bit of loss of precision. The formula for condition number tells us that:

$$\begin{aligned}\kappa_{r_1}(b) &= \left|-1 + \frac{b}{\sqrt{b^2+4}}\right| \frac{|b|}{|-b + \sqrt{b^2+4}|} \\ &= \left|\frac{b - \sqrt{b^2+4}}{\sqrt{b^2+4}}\right| \frac{|b|}{|-b + \sqrt{b^2+4}|} = \frac{|b|}{\sqrt{b^2+4}}\end{aligned}$$

This would suggest the mathematical problem itself is well-conditioned for large b, as it goes to 1 as $b \to \pm\infty$. However, when we evaluate it in finite precision, we have serious loss of precision. Why is that?

Evaluating a polynomial

Consider this exercise from homework, involving different methods to evaluate a polynomial $p(x) = (x-2)^9$. Once again, we can compute $\kappa_p(x) = 9|x-2|^8 \frac{|x|}{|x-2|^9} = 9\frac{|x|}{|x-2|}$. This does have some inherent loss of precision when evaluating x near 2, due to the subtraction involved. For example, evaluating $p(2.01)$ has a condition number of $\kappa_p(2.01) = 9\frac{2.01}{0.01} = 1809$.

We have three potential algorithms to evaluate this:

1. **Compact form:** Evaluate $x - 2$, and then raise it to the ninth power.
2. **Expanded form:** Expand the polynomial as $p(x) = x^9 - 18x^8 + 144x^7 - 672x^6 + 2016x^5 - 4032x^4 + 5376x^3 - 4608x^2 + 2304x - 512$. Evaluate it as the sum of these terms in the monomial basis.
3. **Horner's method:** Take the expanded polynomial, and re-write it as $p(x) = -512 + x(2304 + x(\cdots x(-18 + x))\cdots))$. Evaluate this using a for loop going from inside the parenthesis out.

When we use these three algorithms, however, two of them (compact and Horner) give us relative errors of about 10^{-13} (which is about what we expect from the condition number of around 10^3), but the expanded one gives us horrible errors (near 1, which means the plot of this function looks extremely noisy!).

Solving a 2×2 system of linear equations

As we will discuss later in the semester, one of the types of math problems that we tend to encounter the most involve the solution of a system of m linear (or sometimes non-linear) equations in n unknowns $x_1, \ldots, x_n$. As you might know from previous coursework in linear algebra and calculus, in the case of a system of 2 linear equations in 2 unknowns x, y, solving it can be understood in terms of finding the intersection of two lines (for $n = 3$, the intersection of planes, and so on).

Consider the problem:

$$x + 10^6 y = 10^6$$
$$10^6 x - y = 2 \times 10^6$$

This system is equivalent (dividing both equations by 10^6):

$$10^{-6} x + y = 1$$
$$x - 10^{-6} y = 2$$

By computing slopes, we can see that this involves the intersection of a line that is almost horizontal (with very small negative slope) and a line that is almost vertical (with positive slope 10^6). We can compute the exact solution by hand, and find that it is $x = 2.000000999998, y = 0.999997999999$ (that is, very close to $[2, 1]$).

This system can be written as $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{A}$ a 2×2 matrix, and $\mathbf{x}, \mathbf{b}$ vectors of unknowns and right-hand-sides. A condition number κ_A can then be defined for this system; it turns out to be independent of x and it involves properties of the matrix $\mathbf{A}$ as a linear operator. For this system, κ_A is very close to 1, which means our math problem is very well conditioned.

We then propose two algorithms to solve it using the elimination method (which we may have learned in middle school, and re-learned in Linear Algebra):

1. Eliminate x from the second equation by subtracting the right multiple of the first equation from it. Then solve for y. With the value you get for y, recover the value for x from the first equation.

2. *Change the order of the equations first.* Then carry out the same elimination method as in method 1.

Doing math in pen and paper, both methods should return the same solution. However, it turns out that because of finite precision, the first method amplifies the error by about 10^6, while the second method does not. So, the second method is stable (and we will learn more about it when we see Gaussian Elimination); the first is not.

More on condition number for linear systems of equations

Inspired by the previous example, we want to say a bit more about how we can determine what the "condition number" of a linear system $\mathbf{Ax} = \mathbf{b}$ is for a $n \times n$ non-singular (invertible) matrix $\mathbf{A}$. to make things a bit simpler, we will assume there is only a perturbation (small relative error) in the right-hand-side $\mathbf{b}$, and see how that impacts the relative error in our solution $\mathbf{x}$. Know, however, that similar results exist accounting for small errors in $\mathbf{A}$ (and they give the same condition number!).

So, imagine that the right-hand-side we solve for is not $\mathbf{b}$, but $\widetilde{\mathbf{b}} = \mathbf{b} + \mathbf{\Delta b}$. Take note of the fact that this means each entry of our vector is perturbed by a little bit. Now, since our matrix is invertible, we can find the exact solutions for both the original and the perturbed problem:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$$
$$\widetilde{\mathbf{x}} = \mathbf{A}^{-1}\widetilde{\mathbf{b}} = \mathbf{A}^{-1}\mathbf{b} + \mathbf{A}^{-1}\mathbf{\Delta b}$$

So, we have a formula for the absolute error:

$$||\widetilde{\mathbf{x}} - \mathbf{x}|| = ||\mathbf{A}^{-1}\mathbf{\Delta b}||$$

and the relative error is:

$$\frac{||\widetilde{\mathbf{x}} - \mathbf{x}||}{||\mathbf{x}||} = \frac{||\mathbf{A}^{-1}\mathbf{\Delta b}||}{||\mathbf{x}||}$$

Consider the limit definition for the condition number. It would imply that:

$$\kappa_{\mathbf{A}}(\mathbf{b}) = \lim_{||\mathbf{\Delta b}||\to 0} \frac{\frac{||\mathbf{A}^{-1}\mathbf{\Delta b}||}{||\mathbf{x}||}}{\frac{||\mathbf{\Delta b}||}{||\mathbf{b}||}} = \lim_{||\mathbf{\Delta b}||\to 0} \frac{||\mathbf{A}^{-1}\mathbf{\Delta b}||||\mathbf{b}||}{||\mathbf{x}||||\mathbf{\Delta b}||} = \lim_{||\mathbf{\Delta b}||\to 0} \frac{||\mathbf{A}^{-1}\mathbf{\Delta b}||||\mathbf{Ax}||}{||\mathbf{\Delta b}||||\mathbf{x}||}$$

where we have substituted $\mathbf{b} = \mathbf{Ax}$ on the last line. Note that the final expression involves the product of two fractions of the form $||\mathbf{My}||/||\mathbf{y}||$. This can be understood, intuitively, as *how much matrix* $\mathbf{M}$ *stretches the length of a vector* $\mathbf{y}$ *relative to its length* $||\mathbf{y}||$. It's such an important concept that it defines a kind of **matrix norm**. That is:

Definition 2 *We define the matrix operator 2 norm* $||\mathbf{A}||_2$ *as*

$$||\mathbf{A}||_2 = \max_{\mathbf{x}\neq 0} \frac{||\mathbf{Ax}||_2}{||x||_2} = \max_{||\mathbf{x}||_2=1} ||\mathbf{Ax}||_2$$

Given this definition, one can easily derive the property that for all $\mathbf{x}$, $||\mathbf{Ax}||_2 \leq ||\mathbf{A}||_2||\mathbf{x}||_2$.

And so, we can easily bound the condition number above by a *matrix condition number* (independent of $\mathbf{x}$ or $\mathbf{b}$) $\kappa_{\mathbf{A}} = ||\mathbf{A}||||\mathbf{A}||^{-1}$. We had some intuition of how this might be related to how "linearly independent" the columns (or rows) of $\mathbf{A}$ are from the example above. To gain more intuition, here are some useful results:

- If a matrix $\mathbf{U}$ is unitary (its columns / rows are orthogonal and of length 1), $\kappa_{\mathbf{U}} = 1$.
- If a matrix has a basis of orthonormal eigenvectors with eigenvalues λ_i (e.g. it is real and symmetric), then $\kappa_{\mathbf{A}} = \frac{\max|\lambda_i|}{\min|\lambda_i|}$
- If a matrix $\mathbf{A}$ can be written as $\mathbf{A} = \mathbf{U\Sigma V}$ where Σ is diagonal with non-negative entries σ_i (this is known as a Singular Value Decomposition, or SVD), then $\kappa_{\mathbf{A}} = \frac{\max\sigma_i}{\min\sigma_i} = \frac{\sigma_1}{\sigma_n}$

1.3 Big O and little o notation

Very often when discussing the performance of a numerical method, we will want to be precise about the following:

- Say we have an approximation $u(h)$ to the solution to some math problem, depending on a small parameter h. Here h usually is a discretization length, time step, etc for which, in theory, as $h \to 0$, our approximation tends to the true solution u.

  We want to know: how quickly does $u(h)$ go to u? In other words, how fast does the error $e(h) = u(h) - u$ go to zero?

- Say we have an iterative process that, given an initial guess u_0, produces better and better guesses u_n for our solution u. We might have good reason to think that as $n \to \infty$, $u_n \to u$. How fast does this happen? If we have some error tolerance TOL, how many iterations must I take so that $|u_n - u| \leq \text{TOL}$?
- Say I am solving a problem with N variables (unknowns), and it takes $C(N)$ time to run in my laptop. I increase the resolution of my problem, and it now requires 10 times more variables. By how much does the time it takes to run increase? How fast do memory requirements grow?

To be able to say precise things about all of these, we will be using what is known as "big O and little o" notation. We define these first in the context of the first two questions raised above (which have to do with how fast an approximation approaches the ground truth), and then we explain how big O can be used for the third question (which has to do with growth of costs as N grows).

Definition 3 *Let $f(x)$ and $g(x)$ be defined in an interval around 0. Then, we say that*

- *$f(x) = O(g(x))$ ($f(x)$ is big O of $g(x)$) as $x \to 0$ if there exists constants $M, \delta > 0$ such that*

$$\left|\frac{f(x)}{g(x)}\right| \leq M$$

for all $|x| \leq \delta$. In particular, this will be true if $\lim_{x\to 0} \frac{f(x)}{g(x)} = L$ exists. This means $f(x) \to 0$ at least as fast as $g(x) \to 0$.

- *$f(x) = o(g(x))$ ($f(x)$ is little o of $g(x)$) as $x \to 0$ if*

$$\lim_{x\to 0} \left|\frac{f(x)}{g(x)}\right| = 0$$

that is, if $f(x) \to 0$ faster than $g(x) \to 0$.

Using big O and small o to talk about convergence

Definition 4 (Sequences) *Suppose $\{\beta_n\}_{n=1}^{\infty}$ is a sequence known to converge to 0, and $\{\alpha_n\}_{n=1}^{\infty}$ a sequence that converges to α. If there exists constants K, N such that*

$$|\alpha_n - \alpha| \leq K|\beta_n|$$

for all $n \geq N$, then we say that $\{\alpha_n\}$ converges to α at a rate of $O(\beta_n)$. We can write this as $\alpha_n = \alpha + O(\beta_n)$.

Definition 5 (Continuous approximation) *Suppose $G(h) \to 0$ as $h \to 0$, and $F(h) \to L$. If there exist constants $K, \delta > 0$ such that*

$$|F(h) - L| \leq K|G(h)|$$

for all $|h| \leq \delta$, then we write $F(h) = L + O(G(h))$. We will often use $G(h) = h^p$ with $p > 0$.

Examples

- **Using Taylor expansions to approximate functions:** Say we are interested in computing $F(h) = \sin(h^2)$ for small h. We use the Maclaurin expansion of $sin(x)$ using one and two terms:

$$\sin(x) \simeq x \quad F(h) \simeq F_1(h) = h^2$$
$$\sin(x) \simeq x - \frac{x^3}{6} \quad F(h) \simeq F_3(h) = h^2 - \frac{1}{6}h^6$$

  We are interested in knowing how the error $E_1(h) = F(h) - F_1(h)$ and $E_2(h) = F(h) - F_2(h)$ behave as $h \to 0$. Using the Taylor residual theorem, we find that:

$$\sin(h^2) - h^2 = \frac{\sin(\eta)}{6}h^6 \quad \eta \in (0, h^2)$$
$$\sin(h^2) - h^2 + \frac{h^6}{6} = -\frac{\sin(\mu)}{5!}h^{10} \quad \mu \in (0, h^2)$$

  That is, $E_1(h) = O(h^6)$ and $E_2(h) = O(h^{10})$.

- Find the rate of convergence for $\lim_{h\to 0} \frac{\sin(h)}{h} = 1$.

  Answer: We write down the Maclaurin expansion of $\sin h$ up to cubic degree plus the corresponding remainder, and then divide it by h. We subtract 1 and try to find the leading power of h as $h \to 0$:

$$\frac{\sin h}{h} - 1 = \frac{h - h^3/6 + \sin(\eta)h^5/5!}{h} - 1 = -h^2/6 + \sin(\eta)h^4/5!$$

  Since the smallest power is h^2, as $h \to 0$, this quantity will converge to 1 with error behaving like $O(h^2)$.

Using big O to talk about computational costs

As mentioned above, another important use of big O notation will involve tracking various computational costs: number of floating point operations (proportional to time spent), memory storage (e.g. in MBs), energy use, etc. These are often tracked as a function of problem size N (number of degrees of freedom). For example: For a generic matrix $\mathbf{A}$ of size N $\times$ N,

- Its storage in memory is $O(N^2)$
- Multiplying it times a vector of size N $\times$ 1 takes $O(N^2)$ operations.
- Solving a linear system $\mathbf{Ax} = \mathbf{b}$ (using Gaussian elimination) is $O(N^3)$.

What do we mean by these? The big O notation is used similarly, but now we care about what happens to a cost $C(N)$ as $N \to \infty$. We can determine that by looking at the limit of $C(N)/N^p$, and asking whether numerator or denominator go to infinity faster.

Definition 6 (Big O for cost growth) *Assume $C(N), G(N) \to \infty$ as $N \to \infty$. We will say that $C(N) = O(F(N))$ if*

$$\lim_{N \to \infty} \frac{C(N)}{G(N)} = L$$

We will usually set $G(N) = N^p$ for $p > 0$.

For example: if we can write $C(N) = N^3 + 2N^2 - 4N + 100$, we will say $C(N) = O(N^3)$, as that is the leading term as $N \to \infty$.