1 Conditioning of a problem

In most problems of interest, we are interesting in obtaining an output $f(x)$ for a given input x. This problem is said to be *well-conditioned* if "small" perturbations of x results in only "small" changes in $f(x)$. An *ill-conditioned* problem is one where "small" perturbation of x leads to a "large" change in $f(x)$. The notion of "small" and "large" often depends on the problem and application of interest.

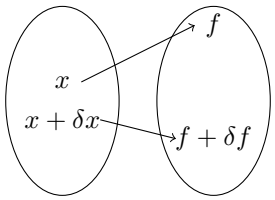

Figure 1: Conditioning quantifies how small/large the change δf in output is for a δx perturbation of the input.

1.1 Absolute condition number

One way to measure "conditioning" of a problem is as follows. Let δx denote a small perturbation of x (the input) and let $\delta f = f(x+\delta x) - f(x)$ be the corresponding change in the output. The *absolute condition number* $\hat{\kappa}(x, f)$ of the problem f at x is defined as

$$\hat{\kappa} = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|}$$

Note that if f is differentiable at x, and $J(x)$ is the Jacobian of $f(x)$ at x, we obtain that

$$\hat{\kappa} = \|J(x)\|$$

1.2 Relative condition number

Note that since the input (x) and the output ($f(x)$) are on different spaces, a more appropriate measure of conditioning is to measure the changes in the input and output in terms of relative changes. The *relative condition number* $\kappa(x, f)$ of the problem f at x is defined as

$$\kappa = \lim_{\delta \to 0} \sup_{\|\delta x\| \leq \delta} \left(\frac{\|\delta f\|}{\|f\|} \Big/ \frac{\|\delta x\|}{\|x\|} \right)$$

As before, if f is differentiable at x, we can express this in terms of the Jacobian $J(x)$ as

$$\kappa = \frac{\|J(x)\|}{\|f(x)\| \,/\, \|x\|}$$

Even though both the above notions have their uses, relative condition number is more appropriate since as we saw earlier, floating point arithmetic introduces only relative errors.

1.3 Conditioning of matrix-vector products

We have $f(x) = Ax$. The Jacobian is nothing but the matrix A. Hence, we have

$$\kappa(x, Ax) = \frac{\|A\| \, \|x\|}{\|Ax\|}$$

Note that

$$\|x\| = \left\|A^{-1}(Ax)\right\| \leq \left\|A^{-1}\right\| \|Ax\|$$

Hence, we obtain that

$$\kappa(x, Ax) = \frac{\|A\| \, \|x\|}{\|Ax\|} \leq \frac{\|A\| \left\|A^{-1}\right\| \|Ax\|}{\|Ax\|} = \|A\| \left\|A^{-1}\right\|$$

where the bound is independent of x. Hence, $\|A\| \left\|A^{-1}\right\|$ is called as the condition number of the matrix A and is denoted as $\kappa(A)$.

1.4 Conditioning of a system of equations

We are interested in solving the linear system $Ax = b$. In this case, we have $f(b) = A^{-1}b$. The Jacobian of $f(b)$ is nothing but the matrix A^{-1}. Hence, we have

$$\kappa(b, x) = \frac{\left\|A^{-1}\right\| \|b\|}{\|A^{-1}b\|} = \frac{\left\|A^{-1}\right\| \left\|A\left(A^{-1}b\right)\right\|}{\|A^{-1}b\|} \leq \frac{\left\|A^{-1}\right\| \|A\| \left\|A^{-1}b\right\|}{\|A^{-1}b\|} = \kappa(A)$$

1.5 Catastrophic cancellation due to finite precision

Consider the recurrence

$$a_{n+2} = 10a_{n+1} - 9a_n$$

where $a_1 = a_2 = 2.95$. If we work in infinite arithmetic, it is easy to see that the solution for the recurrence is $a_n = 2.95$ for all $n \in \mathbb{Z}^+$. Now let us work in finite arithmetic.

```
% Filename: catastrophic_round_off.m
% This solves the following recurrence:
% a(1) = a(2) = 2.95;
% a(n+2) = 10a(n+1) - 9a(n);
% If we had infinite precision, then a(n) would be '2.95' for all n.
% Let us look at how finite precision affect this computation.
clear all;
clc;
N       =   20;                 %       Number of terms
a       =   zeros(N,1);         %       Initialize all a(n)'s to be zero initially
a(1)    =   2.95;               %       a(1) is set to '2.95'
a(2)    =   a(1);               %       a(2) is set to '2.95'
for n=1:N-2
    a(n+2)  =   10*a(n+1)-9*a(n);       %       Recurrence: a(n+2) = 10a(n+1)-9a(n)
end
a                               %       Display the first N values
```

A MATLAB script (catastrophic_round_off.m) implementing the recurrence is shown above. Let us look at the number of digits lost with iteration as we execute the script. Note that we begin with $a_1 = a_2 = 2.95$. Hence, the first iteration is to obtain a_3. The loss of digit is measured as $16 + \lceil \log_{10}(|a_n - a_1|) \rceil$.

Table 1: Digits lost with iteration

Iteration	Value	Digits lost
1	2.9499999999999993	1
2	2.9499999999999922	2
3	2.9499999999999282	3
4	2.9499999999993527	4
5	2.9499999999941728	5
6	2.9499999999475541	6
7	2.9499999995279858	7
8	2.9499999957518703	8
9	2.9499999617668315	9
10	2.9499996559014825	10
11	2.9499969031133411	11
12	2.9499721280200681	12
13	2.9497491521806118	13
14	2.9477423696255052	14
15	2.9296813266295452	15
16	2.7671319396659051	16
17	1.3041874569931444	17
18	-11.862312887061702	18

As seen from Table 1, a digit is lost with each iteration. Instead of starting with 2.95 as the initial value, if we start with $a(1) = a(2) = 2.9375$, there would be no loss of digits. (Why? $2.9375 = 2^1 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4}$, 10×2.9375 and 9×2.9375 has only few significant digits in binary, whereas we can accomodate 52 significant digits in binary in double precision. Hence, all computations including subtraction can be accomodated with 52 significant digits). However, note that the problem is inherently ill-conditioned. This can be seen from the fact that the recurrence can be written as a linear system $Ax = b$, where

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 9 & -10 & 1 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 9 & -10 & 1 & \cdots & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & -10 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 9 & -10 & 1 \end{bmatrix} \in \mathbb{R}^{N \times N}$$

$x \in \begin{bmatrix} a_1 & a_2 & a_3 & a_4 & \cdots & a_{n-2} & a_{n-1} & a_n \end{bmatrix}^T \in \mathbb{R}^N$ and $b \in \begin{bmatrix} 2.95 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \end{bmatrix}^T \in \mathbb{R}^N$. Table 2 provides the condition number of the matrix A, computed using the MATLAB code made available here. By looking at Tables 1 & 2, we see that the number of significant digits we lose is roughly $\log_{10}(\kappa)$. Recall that we can only obtain roughly 16 significant digits of accuracy; hence losing roughly $\log_{10}(\kappa)$ significant digits means the number of accurate significant digits we obtain is roughly $16 - \log_{10}(\kappa)$.

2 Stability of an algorithm

In this section, we will focus on the *Stability of the algorithm*. In an ideal world, we would like to have numerical algorithm that provides exact solution to numerical problems. However, this is not always the case, especially if the underlying problem is continuous. Hence, we need to quantify how good the algorithm we deploy to solve the problem at hand is. Note that when we defined conditioning, the algorithm we adopted to compute $f(x)$ never came into picture. *Conditioning* is purely a property of the underlying problem and has got nothing to do with the algorithm we adopt to solve the problem.

Let $f : X \mapsto Y$, be the mapping we are interested in computing at x. The algorithm can be viewed as another mapping $\tilde{f} : X \to Y$ between the same two spaces. Ideally, we would like $\tilde{f}(x) = f(x)$. However, if this is not the case, we would like to quantify the error due to the algorithm. A natural choice would be either the absolute error $\left(\left\|\tilde{f}(x) - f(x)\right\|\right)$ or the relative error $\left(\dfrac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|}\right)$.

Table 2: Condition number of the matrix A

Size (n)	Condition number (κ)
1	1×10^{0}
2	$2.61803398874989 \times 10^{0}$
3	$1.3796611576979 \times 10^{2}$
4	$1.551097538854 \times 10^{3}$
5	$1.50551034861693 \times 10^{4}$
6	$1.40292391894488 \times 10^{5}$
7	$1.28687924339244 \times 10^{6}$
8	$1.17166090858295 \times 10^{7}$
9	$1.06249523981219 \times 10^{8}$
10	$9.61243482423378 \times 10^{8}$
11	$8.6836691858234 \times 10^{9}$
12	$7.83708938539833 \times 10^{10}$
13	$7.06839188897062 \times 10^{11}$
14	$6.37226330634367 \times 10^{12}$
15	$5.74254032733743 \times 10^{13}$
16	$5.16635097571205 \times 10^{14}$
17	$4.64733612463187 \times 10^{15}$
18	$4.40455740499201 \times 10^{16}$
19	$4.69973619424595 \times 10^{17}$
20	$3.65507320671634 \times 10^{17}$

This is termed as the forward error. An algorithm is said to be forward stable if

$$\frac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|} = \mathcal{O}\left(\epsilon_m\right)$$

where ϵ_m is the machine precision. However, there are couple of issue with the forward stability, *when we are working on a finite precision machine.*

1. The input x might not be exactly represented on the machine and might get represented as $x + \delta x$ on the machine.
2. The function $f(x)$ might involve many operations each of which is subject to its own rounding-off error in finite precision arithmetic.

For instance, lets just consider the first case: x being represented as $x + \delta x$ on the machine. Even if the algorithm, $\tilde{f}$, to compute f is ***exact*** (say no approximation, no rounding off, etc.), the algorithm will output $f(x + \delta x)$. If the problem is ill-conditioned, then the forward error will be large. Hence, if we use forward stability to quantify the goodness of the algorithm, we see that the algorithm is unnecessarily penalized due to the poor conditioning of the problem and the finite precision of the machine. Hence, forward stability is not the right quantity to measure the goodness of the algorithm. To measure the goodness of an algorithm, we need a quantity

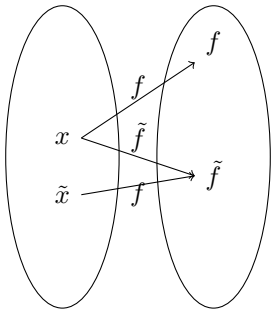

Figure 2:

that removes the effect of conditioning and finite precision of the machine from the forward error. One such quantity that does that is the backward error. We say that an algorithm $\tilde{f}$ is *backward stable*, if for each input x there exists an input $\tilde{x}$ such that

$$f(\tilde{x}) = \tilde{f}(x) \text{ and } \frac{\|\tilde{x} - x\|}{\|x\|} = \mathcal{O}(\epsilon_m)$$

The quantity $\frac{\|\tilde{x} - x\|}{\|x\|}$ is denoted as the backward error. In words,

"A backward stable algorithm gives exactly the right answer to nearly the right question."

Note that $\tilde{x} = f^{-1}\left(\tilde{f}(x)\right)$. Loosely speaking the inverse of the map f removes the ill-conditioning of the forward mapping and the backward error purely characterizes the error due to the algorithm.

3 Backward stability of some basic algorithms

The four simplest computational problems we encounter in almost every algorithm is $+, -, \times$ and $\div$. We will show the backward stability of addition. Proving the backward stability of the rest also follows a similar pattern. Recall that any x_1, x_2 is represented on the machine as $x_1(1+\epsilon_1)$ and $\tilde{x}_2(1+\epsilon_2)$ respectively, where $0 \leq \epsilon_1, \epsilon_2 < \epsilon_m$.

1. **Addition is backward stable**: Consider $f(x_1, x_2) = x_1 + x_2$. The most obvious algorithm gives us

   $$\tilde{f}(x_1, x_2) = x_1(1+\epsilon_1) + x_2(1+\epsilon_2)(1+\epsilon_3)$$

   where $1+\epsilon_3$ takes into account the fact that every operation introduces a rounding off error. This gives us

   $$\tilde{f}(x_1, x_2) = x_1(1+\epsilon_1)(1+\epsilon_3) + x_2(1+\epsilon_2)(1+\epsilon_3) = f(\tilde{x}_1, \tilde{x}_2)$$

   where

   $$\frac{\|\tilde{x}_1 - x_1\|}{\|x\|_1} = \epsilon_1 + \epsilon_3 + \epsilon_1\epsilon_3 = \mathcal{O}(\epsilon_m)$$

   Hence, addition is backward stable.

2. **Inner product is backward stable**: Let $x, y \in \mathbb{R}^n$. Recall that every floating point operation involves a relative error of δ, where $|\delta| < \epsilon_m$ (ϵ_m is the machine precision). We will first prove that the inner product is backward stable. We have

   $$f(x, y) = x^T y = \sum_{k=1}^{n} x_k y_k$$

   When implemented on a finite arithmetic machine, we have

   $$\text{fl}(x^T y) = x_1 y_1 (1+\delta_1) \prod_{j=2}^{n} (1+\delta_j') + \sum_{k=2}^{n} \left(x_k y_k (1+\delta_k) \prod_{j=k}^{n} (1+\delta_j') \right)$$

   where δ_k is the relative error when multiplying $x_k y_k$ and δ_j' is the relative error on adding the term $x_j y_j (1+\delta_j)$.

   Take $\tilde{x}_1 = x_1(1+\delta_1)\prod_{j=2}^{n}(1+\delta_j')$, $\tilde{x}_k = x_k(1+\delta_k)\prod_{j=k}^{n}(1+\delta_j')$ and $\tilde{y} = y$. We have

   $$\tilde{x}_1 - x_1 = x_1 \left((1+\delta_1) \prod_{j=2}^{n} (1+\delta_j') - 1 \right)$$

   $$\tilde{x}_k - x_k = x_k \left((1+\delta_k) \prod_{j=k}^{n} (1+\delta_j') - 1 \right)$$

   Let us use the $\|\cdot\|_{\max}$ norm to compare the relative error between $\tilde{x}$ and x. We have

   $$\|\tilde{x} - x\|_{\max} \leq \|x\|_{\max} \left((1+\epsilon)^n - 1 \right) \implies \frac{\|\tilde{x} - x\|_{\max}}{\|x\|_{\max}} \leq \left((1+\epsilon)^n - 1 \right) \approx n\epsilon + \mathcal{O}((n\epsilon)^2)$$

   and thereby the inner product is backward stable.

3. **Outer product is not backward stable** We will now prove that the outer product is **not necessarily backward stable**. We have

$$\left(xy^T\right)_{ij} = x_i y_j$$

When implemented on a finite arithmetic machine, we have

$$\text{fl}\left(xy^T\right) = \underbrace{\begin{bmatrix} x_1 & 0 & \cdots & 0 \\ 0 & x_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n \end{bmatrix}}_{X} \overbrace{\begin{bmatrix} 1+\delta_{1,1} & 1+\delta_{1,2} & \cdots & 1+\delta_{1,n} \\ 1+\delta_{2,1} & 1+\delta_{2,2} & \cdots & 1+\delta_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 1+\delta_{n,1} & 1+\delta_{n,2} & \cdots & 1+\delta_{n,n} \end{bmatrix}}^{\Delta} \underbrace{\begin{bmatrix} y_1 & 0 & \cdots & 0 \\ 0 & y_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_n \end{bmatrix}}_{Y}$$

Note that in general the matrix X, Δ and Y are full rank matrices (Note that the δ_{ij}'s are not the same). To have backward stability, we first need to find $\tilde{x}$ and $\tilde{y}$ such that $\text{fl}(xy^T) = \tilde{x}\tilde{y}^T$. However, note that the LHS is in general a rank n matrix, whereas the RHS is a rank 1 matrix. Hence, no such $\tilde{x}$ and $\tilde{y}$ exists and thereby the outer product is **not backward stable**.

As we discuss other algorithms, we will state (and in some cases prove) that whether these algorithms are backward stable or not.