What every computational mathematician should know

Dr. Sivaram Ambikasaran*
Stable Accurate Fast Robust Algorithms & Numerics Group
Department of Mathematics
Indian Institute of Technology Madras

July 7, 2020

Conditioning we fail to see,
Will land us in the sea.
Stability is what one can control,
If not, our errors will swole.
Finite precision arithmetic,
You make math on computers frenetic.

1 Abstract

This exposition is a short note on three of the most important concepts in computational mathematics.

1. Floating point arithmetic
2. Conditioning of a problem
3. Stability of an algorithm

The three are independent entities that frequently cross paths in designing numerical algorithms for computational problems.

*Recipient of Saraswathy Srinivasan Prize: Young Scientist Award in Mathematical Sciences (2019); Email ✉: sivaambi@alumni.stanford.edu

2 Floating point arithmetic

Mathematics on computers is a different ball game than pen and paper mathematics. To get rolling, let's look at a simple recurrence relation that can be solved by a high school student.

$$a_1 = a_2 = 2.95$$

$$a_{n+1} = 10a_n - 9a_{n-1}, \quad \forall n \in \mathbb{N} \text{ and } n \geq 2$$

With pen and paper (in fact you do not even need one), it is fairly straight forward to see that $a_n = 2.95$ for all $n \in \mathbb{N}$. Now launch one of the most popular languages used for computing, MATLAB (You can also try this experiment in Python/C++/Fortran/R). Let's try to code up this recurrence and obtain the answers for first few values of n. Below is the little piece of code (Listing 1).

Listing 1: Recurrence

```
% 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) 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
```

The output of the code in Listing 1 is tabulated in Table 1.

As seen from Table 1, a digit is lost with each iteration. Instead of starting with 2.95 as the initial value, if we repeat the experiment with $a(1) = a(2) = 2.9375$, there would be no loss of digits. (Why?). To understand this phenomenon, we need to look at the way numbers are represented on the machine.

Any computer has only finite number of bits to represent real numbers. Hence, we can only represent finitely many real numbers on a computer and will have to deal with approximations of the real number system using finite computer representations. To arrive at a consistent representation of floating point numbers across different computer architecture, the Institute of Electrical and Electronics Engineers proposed a standard for representing real numbers (IEEE-754). Any number on the

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

machine is represented in binary (base 2) format. To be precise, any *normal* number on the machine is represented as

$$x = \pm 1.d_1 d_2 \dots d_s \times 2^e$$

where $1.d_1 d_2 \dots d_s$ is the significand and e is the exponent (both represented using 0's and 1's). For example, consider the number 77 in decimal. We have

$$77_{10} = 2^6 + 2^3 + 2^2 + 2^0 = 1001101_2 = 1.001101_2 \times 2^6 = 1.001101_2 \times 2^{110_2}$$

As indicated in Figure 1, there is 1 sign bit, e bits for the exponent and s bits for

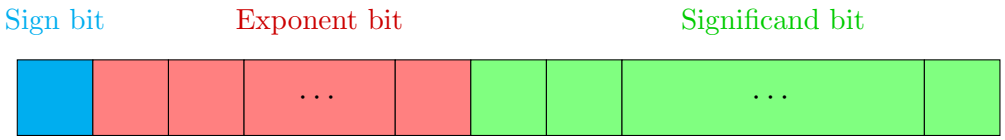

Figure 1: Bits to represent floating point numbers

the significand. We will now list out the general conventions based on IEEE-754 standard.

- **Sign bit**: 0 indicates +, and 1 indicates −.
- **Exponent bit**: Since there are e bits for the exponent, there are a total of 2^e values the exponent can take. To represent negative exponents as well, a bias of $2^{e-1} - 1$ is introduced, i.e., 0 exponent is represented as $0\overbrace{111\dots1}^{e-1}{}_2$.
- **Significand bit**: Stores the leading bits in the mantissa apart from the leading 1.

2.1 Normal floating point number

These are represented as

$$\pm 1.d_1 d_2 \dots d_s \times 2^E$$

where $2 - 2^{e-1} \leq E \leq 2^{e-1} - 1$ (after bias). Note that since normal floating numbers begin with 1, it suffices to store the s bits after the leading 1.

Table 2: Floating point representations

		Significand	
Exponent		All zeros	Non-zero
	All zeros	± 0	Sub-normal numbers
	All ones	$\pm\infty$	Not A Number
	Else	Normal floating point numbers	

Table 3: Positive normal floating point number

	Mantissa	Exponent (without bias)	Number
Smallest	$d_i = 0$ for all $i \leq s$	$00\ldots01$	$2^{2-2^{e-1}}$
Largest	$d_i = 1$ for all $i \leq s$	$11\ldots10$	$(2-2^{-s}) \times 2^{2^{e-1}-1}$

2.2 Sub-normal floating point number

If all the bits in the exponent are zeros, the machine interprets this number either as zero or a sub-normal floating point number. When the significand is non-zero, then such a number is called as a sub-normal number. These numbers need to be interpreted as

$$\pm 0.d_1 d_2 \ldots d_s \times 2^{2-2^{e-1}}$$

The significand stores the s bits after the leading 0. The reason for having sub-

Table 4: Positive sub-normal floating point number

	Representation	Number
Smallest	$d_i = 0$ for all $i < s$ and $d_s = 1$	$2^{2-s-2^{e-1}}$
Largest	$d_i = 1$ for all $i \leq s$	$(1-2^{-s}) \times 2^{2-2^{e-1}}$

normal numbers is to ensure a gradual underflow to zero. For the purposes of analysis, we will assume that all numbers representable on the machine are normal numbers. Note that the number of significand digits in sub-normal numbers is lesser than normal numbers.

2.3 Machine precision

This is defined as the difference between the smallest number exceeding 1 that can be represented on the machine and 1 (There are slightly different definitions depending on the way rounding is done. In this article, we will assume that by rounding we mean truncating or chopping off the digits. Depending on that the definition of machine precision changes). Note that the smallest number exceeding 1 that can be represented on the machine is $1.00\ldots01 = 1 + 2^{-s}$.

$$\boxed{\text{Hence, machine precision is } \epsilon_m = 2^{-s}}$$

Note that if x is any real number (within the representable bounds on the machine) and $\text{fl}(x)$ is the floating point representation of x (i.e., after appropriate chopping x will be represented as $\text{fl}(x)$ on the machine), we then have

$$\left| \frac{x - \text{fl}(x)}{x} \right| \leq \epsilon_m$$

From the above, observe that floating point representation introduces relative errors and not absolute errors.

- **Single precision** Of the total of 32 bits, the first one is allotted for sign, the next 8 for exponent and the remaining 23 for significand.
- **Double precision** Of the total of 64 bits, the first one is allotted for sign, the next 11 for exponent and 52 for significand.

Table 5: Floating point numbers on single and double precision

		32 bit machine	64 bit machine
Sub-normal	Smallest positive	$2^{-149} \approx 1.4 \times 10^{-45}$	$2^{-1074} \approx 4.94 \times 10^{-324}$
	Largest positive	$(1 - 2^{-23}) \times 2^{-126} \approx 1.18 \times 10^{-38}$	$(1 - 2^{-52}) \times 2^{-1022} \approx 2.23 \times 10^{-308}$
Normal	Smallest positive	$2^{-126} \approx 1.18 \times 10^{-38}$	$2^{-1022} \approx 2.23 \times 10^{-308}$
	Largest positive	$(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$	$(2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}$
Machine precision		$2^{-23} \approx 1.2 \times 10^{-7}$	$2^{-52} \approx 2.2 \times 10^{-16}$

Note that because of error in representation, any basic operation (including $+, -, \times, \div$) is subjected to a relative error bounded by the machine precision, i.e., if x and y are floating point numbers (i.e., they are represented exactly on the machine), then we have

$$\text{fl}\left(x \oplus y\right) = \left(x \oplus y\right)\left(1 + \delta\right)$$

where $\oplus \in \{+, 0, \times, \div\}$ and $\mathrm{fl}\,(\cdot)$ is the floating point representation of the resulting quantity and $|\delta| \leq \epsilon_m =$ machine precision.

For more details on IEEE floating point arithmetic, the readers may refer to [1].

3 Conditioning of a problem

In almost all applications, we are interested in obtaining an output $f(x)$ for a given input x. However, there are inherent uncertainties in the input data x. These uncertainties could arise not only because of our inability to measure the input precisely but also due to the fact that numbers need not be represented exactly on the machine (as seen in the previous section). Hence, it is vital to understand how the solution to a problem gets affected by small perturbations in the input data. A given problem is said to be *well-conditioned* if "small" perturbations in x result in "small" changes in $f(x)$. An *ill-conditioned* problem is one where "small" perturbations in x lead to a "large" change in $f(x)$. The notion of "small" and "large" often depends on the problem and application of interest.

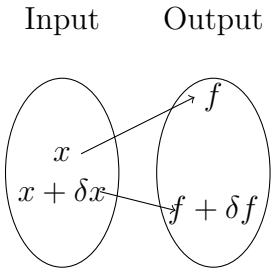

Figure 2: Conditioning quantifies how small/large the change δf in output is for a δx perturbation in the input.

3.1 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

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

where $\|\cdot\|$ denotes an appropriate norm. Note that if f is differentiable at x, and $J(x)$ is the Jacobian of $f(x)$ at x, we obtain that

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

3.2 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

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

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.

To gain a better understanding of the notion of conditioning, we will look at conditioning of some basic problems in the next couple of sub-sections.

3.3 Conditioning of subtraction

Consider **subtracting two positive numbers**, i.e., $f(a, b) = a - b$. If we perturb a by $a + \delta_a$ and b by $b + \delta_b$, we have the condition number in 2-norm to be

$$\kappa(f; a, b) = \lim_{r \to 0} \sup_{\|\delta\|_2 = r} \frac{|\delta_a - \delta_b| \, / \, |a - b|}{\sqrt{\delta_a^2 + \delta_b^2} / \sqrt{a^2 + b^2}} = \lim_{r \to 0} \sup_{\|\delta\|_2 = r} \frac{|\delta_a - \delta_b| \, / \, |a - b|}{r / \sqrt{a^2 + b^2}} = \frac{\sqrt{2}\sqrt{a^2 + b^2}}{|a - b|}$$

Hence, we see that for large values of a and b such that $a - b$ is small (i.e., a is close to b), the problem is ill-conditioned.

3.4 Conditioning of solving for roots of polynomials

Consider finding the **roots of the polynomial** $ax^2 + bx + c$. Here the function $f : \mathbb{R}^3 \mapsto \mathbb{R}^2$, where $f(a, b, c) = \begin{bmatrix} r_1 & r_2 \end{bmatrix}$, where r_1, r_2 are the roots of $ax^2 + bx + c$. Now let's look at the condition at $(a, b, c) = (1, -2, 1)$. The roots are $1, 1$. Let's

perturb the 2 by δ. We have

$$\begin{aligned}\kappa &= \lim_{\delta\to 0} \frac{\|f(1,-(2+\delta),1) - f(1,-2,1)\| \,/\, \|f(1,-2,1)\|}{\|(1,-(2+\delta),1) - (1,-2,1)\| \,/\, \|(1,-2,1)\|} \\ &= \frac{\|(1,-2,1)\|}{\|f(1,-2,1)\|} \lim_{\delta\to 0} \frac{\left\| \frac{\delta+\sqrt{\delta^2+4\delta}}{2}, \frac{\delta-\sqrt{\delta^2+4\delta}}{2} \right\|}{\delta} \\ &= \frac{\sqrt{6}}{2\sqrt{2}} \lim_{\delta\to 0} \frac{\sqrt{2\delta^2+2\delta^2+8\delta}}{\delta} = \sqrt{3} \lim_{\delta\to 0} \frac{\sqrt{\delta^2+2\delta}}{\delta} = \sqrt{3} \lim_{\delta\to 0} \sqrt{1+2/\delta} = \infty\end{aligned}$$

Hence, obtaining roots of a polynomial is ill-conditioned (especially when we want to obtain a double root).

3.5 Conditioning of matrix-vector products

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

$$\kappa\left(f; A, x\right) = \frac{\|A\|\,\|x\|}{\|Ax\|}$$

Note that

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

Hence, we obtain that

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

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

3.6 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

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

For a bit more elaborate discussion on conditioning of a problem, the readers may refer to Lecture 12 in [2].

4 Stability of an algorithm

In an ideal world, where numerical algorithms provide exact solution to the underlying problem, condition number of the problem would be the only quantity that would affect the accuracy of the solution. However, it turns out that the algorithm (or more precisely the sequence of steps) we deploy to solve our problem also affects the accuracy of our solution. Hence, we need to quantify how good our algorithm 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. To understand the notion of stability of an algorithm, lets take a step back and try to abstract what we are after.

Let $f : X \mapsto Y$ be the mapping we are interested in computing. The algorithm can be viewed as another mapping $\tilde{f} : X \mapsto Y$ 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 $\left(\frac{\left\|\tilde{f}(x) - f(x)\right\|}{\|f(x)\|}\right)$. 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 issues 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 that removes the effect of conditioning and finite precision of the machine from the forward error.

This motivates the definition of backward error and backward stability. We say that an algorithm $\tilde{f}$ is *backward stable*, if for each input x there exists an input $\tilde{x}$ such that

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

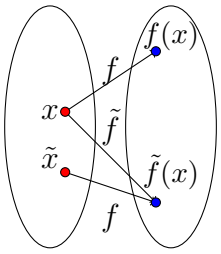

Figure 3: Diagram for understanding forward error and backward error

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

"A backward stable algorithm gives exactly the right output to *nearly* the right input."

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

4.1 Backward stability of some basic algorithms

The four simplest computational steps we encounter in almost every algorithm ($+, -, \times$ and $\div$) are all backward stable.

1. **Subtraction is backward stable**: We will show the backward stability of subtraction. Proving the backward stability of the remaining $(+, \times, \div)$ also follows a similar pattern. Recall that any x_1, x_2 is represented on the machine as $x_1(1 + \epsilon_1)$ and $x_2(1 + \epsilon_2)$ respectively, where $|\epsilon_1|, |\epsilon_2| \leq \epsilon_m$.
   Consider $f(x_1, x_2) = x_1 - x_2$. We have

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

   where $1 + \epsilon_3$ takes into account the fact that every operation introduces a rounding off error ($|\epsilon_3| < \epsilon_m$). 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 $\tilde{x}_1 = x_1 (1+\epsilon_1)(1+\epsilon_3)$ and $\tilde{x}_2 = x_2 (1+\epsilon_2)(1+\epsilon_3)$. We have

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

and

$$\frac{|\tilde{x}_2 - x_2|}{|x_2|} = \epsilon_2 + \epsilon_3 + \epsilon_2\epsilon_3 = \mathcal{O}(\epsilon_m)$$

Hence, addition is backward stable.

2. **Inner product is backward stable**: Let $x, y \in \mathbb{R}^n$. We have

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

Note that each component of x and y have an error in representation on the machine, i.e., $\text{fl}(x_i) = x_i(1+\delta_{x,i})$ and $\text{fl}(y_i) = y_i(1+\delta_{y,i})$. Recall that every floating point operation involves a relative error of δ, where $|\delta| \leq \epsilon_m$ (ϵ_m is the machine precision). Hence, note that

$$\text{fl}(x_i y_i) = \text{fl}(x_i)\,\text{fl}(y_i)(1+\delta_i) = x_i(1+\delta_{x,i})\, y_i(1+\delta_{y,i})(1+\delta_i)$$

where δ_i is the relative error when multiplying $\text{fl}(x_i)$ and $\text{fl}(y_i)$.
When implemented on a finite arithmetic machine, we have

$$\begin{aligned}
\text{fl}(x^T y) &= \text{fl}\left(\sum_{i=1}^{n} x_i y_i\right) = \left(\text{fl}\left(\sum_{i=1}^{n-1} x_i y_i\right) + \text{fl}(x_n y_n)\right)(1+\delta_n') \\
&= \text{fl}\left(\sum_{i=1}^{n-1} x_i y_i\right)(1+\delta_n') + (\text{fl}(x_n y_n))(1+\delta_n') \\
&= \text{fl}\left(\sum_{i=1}^{n-1} x_i y_i\right)(1+\delta_n') + x_n y_n (1+\delta_{x,n})(1+\delta_{y,n})(1+\delta_n)(1+\delta_n')
\end{aligned}$$

where δ_j' is the relative error on adding the term $\text{fl}(x_j)\,\text{fl}(y_j)(1+\delta_j)$.
Take

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

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

and

$$\tilde{y}_k = y_k(1+\delta_{y,k})$$

We have

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

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

$$\tilde{y}_k - y_k = y_k \delta_{y,k}$$

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

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

$$\frac{\|\tilde{y} - y\|_{\max}}{\|y\|_{\max}} \leq \epsilon_m$$

and thereby for each fixed ("small") n, 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 $\text{fl}\left(xy^T\right)$ to be

$$\underbrace{\begin{pmatrix} x_1(1+\delta_{x,1}) & 0 & \cdots & 0 \\ 0 & x_2(1+\delta_{x,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & x_n(1+\delta_{x,n}) \end{pmatrix}}_{X} \overbrace{\begin{pmatrix} 1+\delta_{11} & 1+\delta_{12} & \cdots & 1+\delta_{1n} \\ 1+\delta_{21} & 1+\delta_{22} & \cdots & 1+\delta_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1+\delta_{n1} & 1+\delta_{n2} & \cdots & 1+\delta_{nn} \end{pmatrix}}^{\Delta} \underbrace{\begin{pmatrix} y_1(1+\delta_{y,1}) & 0 & \cdots & 0 \\ 0 & y_2(1+\delta_{y,2}) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & y_n(1+\delta_{y,n}) \end{pmatrix}}_{Y}$$

Note that the matrices X, Δ and Y are in general 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, i.e., $\text{fl}(xy^T)$, is in general a rank n matrix, whereas the RHS, ie.., $\tilde{x}\tilde{y}^T$, is a rank 1 matrix. Hence, no such $\tilde{x}$ and $\tilde{y}$ exists and thereby the outer product is **not backward stable**.

For a bit more elaborate discussion on the different notions of stability, the readers may refer to Lectures 14 & 15 in [2].

5 Revisiting the example

Now let's try to understand the reason why the output of Listing 1 doesn't match with the pen and paper calculation. First observe that 2.95 in binary is $10.11\overline{1100}$ and hence has a non-terminating representation in binary. In double precision representation, 2.95_{10} gets represented on the machine (we will assume that the machine rounds the number to the nearest number representable on the machine) as

$$\begin{aligned}10.11\underbrace{1100\cdots1100}_{\text{repeats 11 times}}1101_2 &= \left(2+\frac{1}{2}+\frac{1}{2^2}+\frac{1}{2^3}+\frac{1}{2^4}+\frac{1}{2^7}+\frac{1}{2^8}+\cdots+\frac{1}{2^{47}}+\frac{1}{2^{48}}+\frac{1}{2^{50}}\right)_{10}\\ &= \frac{3321404725185741_{10}}{11258999068426 24_{10}}\\ &= 2.95000000000000017763568394002504646778106689453125_{10}\end{aligned}$$

Hence, we see that the relative-error in representation of 2.95_{10} on the machine is

$$\begin{aligned}\frac{2.95000000000000017763568394002504646778106689453125_{10}-2.95_{10}}{2.95_{10}} &\approx \frac{1.7764_{10}\times10^{-16}}{2.95_{10}}\\ &\approx 6.02155\times10^{-17}\end{aligned}$$

Now let's quantify the error as the recurrence progresses till we obtain the twentieth term. Note that the recurrence can be written as a linear system

$$\begin{aligned}a_1 &= 2.95\\ a_2 &= 2.95\\ a_3-10a_2+9a_1 &= 0\\ a_4-10a_3+9a_2 &= 0\\ \vdots &= \vdots\\ a_n-10a_{n-1}+9a_{n-2} &= 0\end{aligned}$$

Recasting this in matrix form, we have

$$A\vec{a}=\vec{r}$$

where

$$A=\begin{bmatrix}1&0&0&0&\cdots&0&0&0\\0&1&0&0&\cdots&0&0&0\\9&-10&1&0&\cdots&0&0&0\\0&9&-10&1&\cdots&0&0&0\\0&0&9&-10&\cdots&0&0&0\\\vdots&\vdots&\vdots&\vdots&\ddots&\vdots&\vdots&0\\0&0&0&0&\cdots&-10&1&0\\0&0&0&0&\cdots&9&-10&1\end{bmatrix},\ \vec{a}=\begin{bmatrix}a_1\\a_2\\a_3\\a_4\\a_5\\\vdots\\a_{n-1}\\a_n\end{bmatrix},\vec{r}=\begin{bmatrix}2.95\\2.95\\0\\0\\0\\\vdots\\0\\0\end{bmatrix}.$$

The input to the problem is the matrix A and the right hand side vector $\vec{r}$. The output is the vector $\vec{a}$. Note that the matrix A is exactly represented on the machine. The only error in the input is in representing the vector $\vec{r}$ on the machine and is given by

$$\delta\vec{r} = \vec{r}_{\text{represented}} - \vec{r} \approx \begin{bmatrix} 1.7764 \times 10^{-16} \\ 1.7764 \times 10^{-16} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \in \mathbb{R}^{n \times 1}$$

The solution computed is denoted as $\vec{a}_{\text{computed}}$ and in MATLAB notation $\vec{a}_{\text{computed}} = A \backslash \vec{r}_{\text{represented}}$.
The input error is

$$\frac{\left\|\delta\vec{r}\right\|_2}{\|\vec{r}\|_2} = \frac{\|\vec{r}_{\text{represented}} - \vec{r}\|_2}{\|\vec{r}\|_2} \approx \frac{1.7764 \times 10^{-16}\sqrt{2}}{2.95\sqrt{2}} \approx 6.02155 \times 10^{-17}$$

The observed output error (i.e., the observed forward error) is

$$\frac{\left\|\delta\vec{a}\right\|_2}{\|\vec{a}\|_2} = \frac{\|\vec{a}_{\text{computed}} - \vec{a}\|_2}{\|\vec{a}\|_2}$$

The predicted maximum output error (i.e., the predicted maximum forward error) is

$$\kappa \times \frac{\left\|\delta\vec{r}\right\|_2}{\|\vec{r}\|_2}$$

where κ is the condition number of the problem given by

$$\kappa = \frac{\|A^{-1}\|_2 \|\vec{r}\|_2}{\|\vec{a}\|_2}$$

The backward error is

$$\frac{\|A\vec{a}_{\text{computed}} - \vec{r}\|_2}{\|\vec{r}\|_2}$$

Note that we are using the $\|\cdot\|_2$ to obtain the relative errors and relative condition number. All these have been tabulated in Table 5.

Table 6: Relative forward error and Relative backward error

Matrix size	Condition number	Predicted maximum forward error	Observed forward error	Backward error
n	$\kappa = \frac{\left\|A^{-1}\right\|_2 \left\|\vec{r}\right\|_2}{\left\|\vec{a}\right\|_2}$	$\kappa \times \frac{\left\|\vec{\delta r}\right\|_2}{\left\|\vec{r}\right\|_2}$	$\frac{\left\|\vec{\delta a}\right\|_2}{\left\|\vec{a}\right\|_2}$	$\frac{\left\|A\vec{a}_{\text{computed}} - \vec{r}\right\|_2}{\left\|\vec{r}\right\|_2}$
3	1.105e+01	6.651e-16	1.738e-16	4.258e-16
4	9.128e+01	5.497e-15	1.663e-15	4.258e-16
5	7.398e+02	4.455e-14	1.341e-14	4.258e-16
6	6.083e+03	3.663e-13	1.102e-13	4.258e-16
7	5.069e+04	3.052e-12	9.182e-13	4.258e-16
8	4.267e+05	2.570e-11	7.730e-12	4.258e-16
9	3.621e+06	2.180e-10	6.559e-11	4.258e-16
10	3.092e+07	1.862e-09	5.600e-10	4.258e-16
11	2.653e+08	1.597e-08	4.806e-09	4.258e-16
12	2.286e+09	1.376e-07	4.141e-08	4.258e-16
13	1.977e+10	1.190e-06	3.581e-07	4.258e-16
14	1.714e+11	1.032e-05	3.105e-06	4.258e-16
15	1.491e+12	8.975e-05	2.700e-05	4.258e-16
16	1.299e+13	7.821e-04	2.353e-04	4.258e-16
17	1.134e+14	6.829e-03	2.054e-03	4.258e-16
18	9.919e+14	5.973e-02	1.797e-02	4.258e-16
19	8.689e+15	5.232e-01	1.574e-01	4.258e-16
20	7.622e+16	4.590e+00	1.381e+00	4.258e-16

From Table 5, we can make the following inferences:

- The error in the input data due to finite precision representation of the number 2.95 is of the order of machine precision. Note that if we had started the recurrence with $a_1 = a_2 = 2.9375$ instead of $a_1 = a_2 = 2.95$, then there is no input error since the input $2.9375 = 2 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16}$ is exactly represented in double precision on the machine.
- The condition number of the problem scales exponentially with the problem size, which in turn results in the *forward error being large even though the input error is small.*
- The *backward error is small and is of the order of machine precision.* This indicates that the algorithm (i.e., in this case, forward substitution) is backward stable.
- The predicted maximum forward error is very close to the observed forward error and clearly acts as an upper bound.

6 Conclusion

We have looked at three of the most important aspects to be kept in mind when designing numerical algorithms. Any computational scientist worth his salt would have to pay deep attention to each of these facets when designing algorithms to solve computational problems.

References

[1] Michael L Overton. *Numerical computing with IEEE floating point arithmetic.* SIAM, 2001.

[2] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. SIAM, 1997.