

# Math 6610: Analysis of Numerical Methods, I

## Floating-point arithmetic and computer algorithms

Department of Mathematics, University of Utah

Fall 2025

Resources: Trefethen and Bau 1997, Lectures 12-15  
Atkinson 1989, Chapter 1  
Salgado and Wise 2022, Sections 4.2, 4.3

# Finite representation of numbers

Numbers in **decimal** are represented as

34.1503,

# Finite representation of numbers

Numbers in **decimal** are represented as

34.1503,

This actually means the more complicated expression

$$3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} + 0 \times 10^{-3} + 3 \times 10^{-4}$$

# Finite representation of numbers

Numbers in decimal are represented as

34.1503,

This actually means the more complicated expression

$$3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} + 0 \times 10^{-3} + 3 \times 10^{-4}$$

There's an anatomy to this breakdown:

- The portion 34 is the “integer” part.
- 1503 is the “fractional” part.
- The integer and fractional part are separated by the **radix** (point).
- The base 10 is to be understood (implied) by context.

(34.1503  $\neq$  3.41503)

# Finite representation of numbers

Numbers in **decimal** are represented as

34.1503,

This actually means the more complicated expression

$$3 \times 10^1 + 4 \times 10^0 + 1 \times 10^{-1} + 5 \times 10^{-2} + 0 \times 10^{-3} + 3 \times 10^{-4}$$

There's an anatomy to this breakdown:

- The portion 34 is the “integer” part.
- 1503 is the “fractional” part.
- The integer and fractional part are separated by the **radix** (point).
- The **base 10** is to be understood (implied) by context.

Without context, the **base**  $b$  could be other positive integers:

$$b = 6 \Rightarrow 3 \times 6^1 + 4 \times 6^0 + 1 \times 6^{-1} + 5 \times 6^{-2} + 0 \times 6^{-3} + 3 \times 6^{-4}$$

Note that by convention  $b$  must be strictly larger than all the digits.  
(So 34.1503 could be base 6, or 10, or 978, but could not be base 5.)

## Computer representations

Circuits typically detect the presence (1) or absence (0) of an electrical signal.

For this reason, base  $b = 2$  is the standard computer format, e.g.,:

100010.0010011001111

is approximately equal to the decimal 34.1503.

# Computer representations

Circuits typically detect the presence (1) or absence (0) of an electrical signal.

For this reason, base  $b = 2$  is the standard computer format, e.g.,:

100010.0010011001111

is approximately equal to the decimal 34.1503.

Computers store this *binary* representation of these numbers, and each digit is a “bit”.

8 bits = 1 “byte”

# Computer representations

Circuits typically detect the presence (1) or absence (0) of an electrical signal.

For this reason, base  $b = 2$  is the standard computer format, e.g.,:

100010.0010011001111

is approximately equal to the decimal 34.1503.

Computers store this *binary* representation of these numbers, and each digit is a “bit”.

8 bits = 1 “byte”

Binary representations must store the integer part, the fractional part, and typically also a sign.



# Computer representations

Circuits typically detect the presence (1) or absence (0) of an electrical signal.

For this reason, base  $b = 2$  is the standard computer format, e.g.,:

100010.0010011001111

is approximately equal to the decimal 34.1503.

Computers store this *binary* representation of these numbers, and each digit is a “bit”.

8 bits = 1 “byte”

Binary representations must store the integer part, the fractional part, and typically also a sign.

Q: Why do we always write numbers in base 10?

## Fixed point vs floating point

Fixed-point representations have a fixed radix point, with the size of the fractional part predetermined:

$$100010. \underbrace{0010011001111}_{\text{fixed number of entries}}$$

Then the fractional precision of this representation is fixed for any number.

This *truncation* of finite representations is one of the main challenges to address in numerical computations.

## Fixed point vs floating point

Fixed-point representations have a fixed radix point, with the size of the fractional part predetermined:

$$100010. \underbrace{0010011001111}_{\text{fixed number of entries}}$$

Then the fractional precision of this representation is fixed for any number.

This *truncation* of finite representations is one of the main challenges to address in numerical computations.

Fixed-point representation has a ~~restricted~~ range of precision (pre)defined by the size of the fractional part.

*restricted*

## Fixed point vs floating point

Fixed-point representations have a fixed radix point, with the size of the fractional part predetermined:

$$100010. \underbrace{0010011001111}_{\text{fixed number of entries}}$$

Then the fractional precision of this representation is fixed for any number.

This *truncation* of finite representations is one of the main challenges to address in numerical computations.

Fixed-point representation has a restricted range of precision (pre)defined by the size of the fractional part.

Floating-point representations allow the radix to float:

$$1. \underbrace{000100010011001111}_{\text{fixed number of entries}} + \text{exponent}$$

The **exponent** encodes which exponent the radix is aligned with. (Above: 5)

# Floating-point representations

Generally speaking, floating-point representations store:

significand + exponent + sign

The significand combines the integer and fractional parts of the number.

The exponent encodes the location of the radix.

Floating-point representations allows for a (much) larger operating range than fixed-point representations.

# Floating-point representations

Generally speaking, floating-point representations store:

significand + exponent + sign

The significand combines the integer and fractional parts of the number.

The exponent encodes the location of the radix.

Floating-point representations allows for a (much) larger operating range than fixed-point representations.

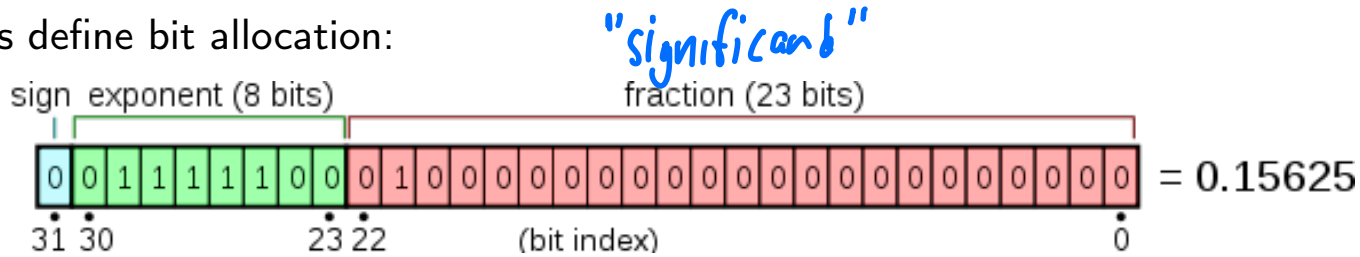
The most popular representation is the IEEE 754 standard, defining various formats:

- Binary 16
- Binary 32 (“single precision”)
- Binary 64 (“double precision”)
- ⋮
- Complex numbers are typically stored as two floats, e.g., Complex 128 (64 + 64)

Most high-level scientific computing languages use double precision as default.

# Floating-point details

These standards define bit allocation:

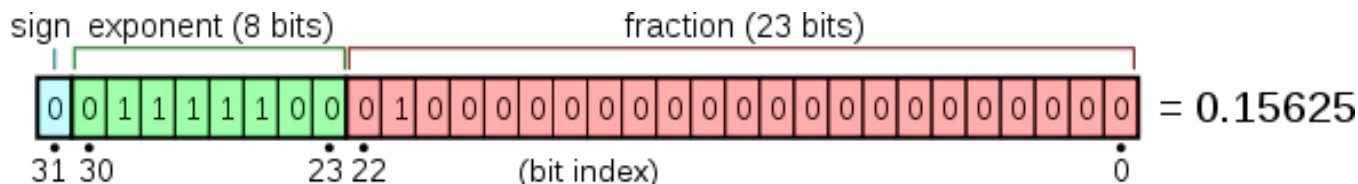


(single precision)

Source: [https://en.wikipedia.org/wiki/File:Float\\_example.svg](https://en.wikipedia.org/wiki/File:Float_example.svg)

# Floating-point details

These standards define bit allocation:



Source: [https://en.wikipedia.org/wiki/File:Float\\_example.svg](https://en.wikipedia.org/wiki/File:Float_example.svg)

The number of bits allocated to the exponent indicates the rounding precision of the format.

Machine precision or machine epsilon is the maximum relative rounding error due to finite representation. If  $\text{fl}(x)$  is the floating-point representation of a number  $x$ , then machine precision is the maximum value of

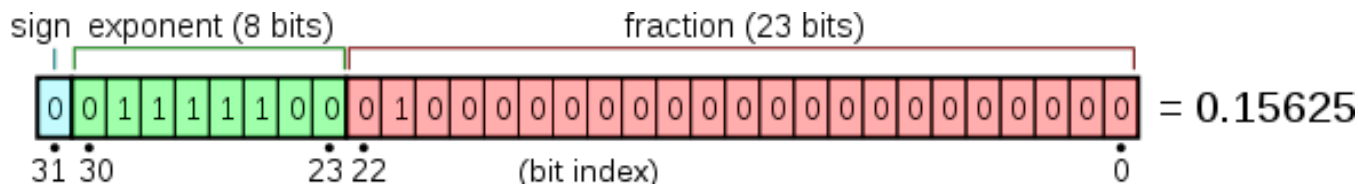
$$\left| \frac{x - \text{fl}(x)}{x} \right|.$$

Roughly speaking, machine epsilon is also the largest  $\epsilon$  such that  $1 + \epsilon$  is rounded to 1.



# Floating-point details

These standards define bit allocation:



Source: [https://en.wikipedia.org/wiki/File:Float\\_example.svg](https://en.wikipedia.org/wiki/File:Float_example.svg)

The number of bits allocated to the exponent indicates the rounding precision of the format.

Machine precision or  <sup>$\epsilon_{mach}$</sup> machine epsilon is the maximum relative rounding error due to finite representation. If  $\text{fl}(x)$  is the floating-point representation of a number  $x$ , then machine precision is the maximum value of

$$|x - \text{fl}(x)| \lesssim |x| \cdot \epsilon_{mach}$$

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

$$2^{-52} \approx 2.22 \cdot 10^{-16}$$

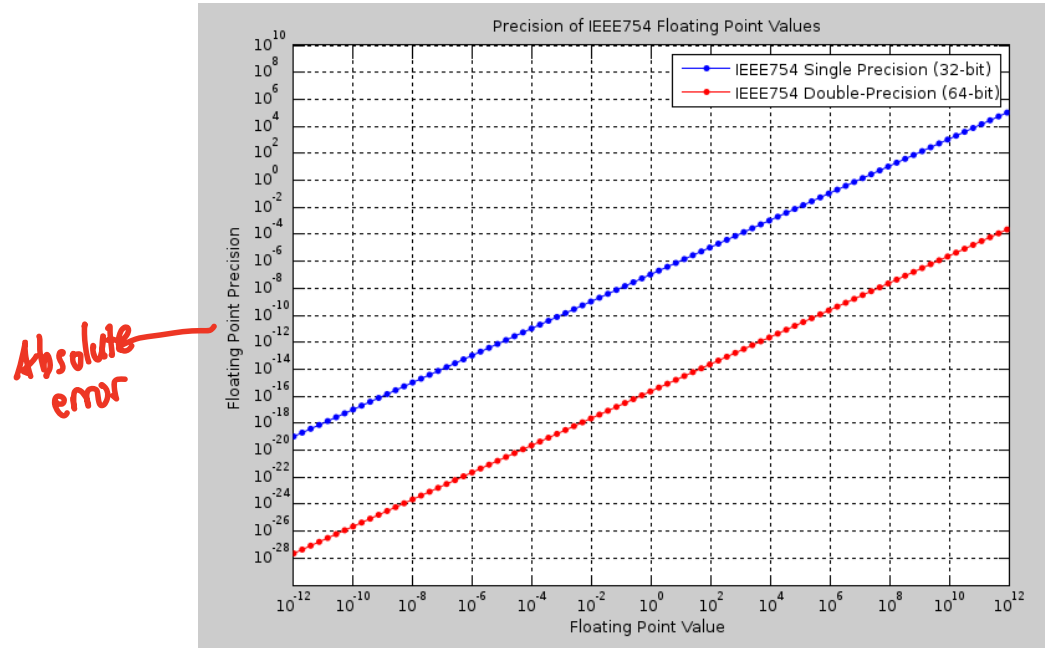
Roughly speaking, machine epsilon is also the largest  $\epsilon$  such that  $1 + \epsilon$  is rounded to 1.

The roundoff or truncation error associated with a floating-point format is essentially equal to  $\text{base}^{-(\# \text{significant digits})}$ .

E.g.: double precision: base  $b=2$ , # significant digits = 52

# Machine precision

Floating-point representation ensures that there is not really an absolute error committed by computer representations, there is only a relative error.



Source: <https://en.wikipedia.org/wiki/File:IEEE754.svg>

## Roundoff and computing

The rounding truncation imparted by finite representations requires attention to how algorithms are implemented. E.g.,

- Implementation of the formula  $\sqrt{1+x^4} - 1$  for small positive  $x$ .

## Roundoff and computing

The rounding truncation imparted by finite representations requires attention to how algorithms are implemented. E.g.,

- Implementation of the formula  $\sqrt{1+x^4} - 1$  for small positive  $x$ .
- Evaluation of  $e^x$  for  $x < 0$ .

## Roundoff and computing

The rounding truncation imparted by finite representations requires attention to how algorithms are implemented. E.g.,

- Implementation of the formula  $\sqrt{1+x^4} - 1$  for small positive  $x$ .
- Evaluation of  $e^x$  for  $x < 0$ .  $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$
- Evaluation of  $\frac{f(x+h)-f(x)}{h}$  for small  $h$ .

# Solution sensitivity

Goal: understand sensitivity of numerical algorithms to roundoff errors. (“stability”)

A first step: understand sensitivity of solutions of *mathematical* problems. (“conditioning”)

# Solution sensitivity

Goal: understand sensitivity of numerical algorithms to roundoff errors. (“stability”)

A first step: understand sensitivity of solutions of *mathematical* problems. (“conditioning”)

## Example

Let  $f(x) = ax$  for scalars  $a, x$ .

The sensitivity of the map  $x \mapsto f(x)$  depends on the value of  $a$ .

## Absolute sensitivity measures

Let  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ .

A sensible measure of sensitivity of  $f$  at  $x$  is perturbation-based:

$$\frac{\|f(x + \delta x) - f(x)\|}{\|\delta x\|}.$$

(some choice of norms)



## Absolute sensitivity measures

Let  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ .

A sensible measure of sensitivity of  $f$  at  $x$  is perturbation-based:

$$\frac{\|f(x + \delta x) - f(x)\|}{\|\delta x\|}.$$

As with derivatives, we can measure the sensitivity at  $x$  by taking limits:

$$\hat{\kappa}_f(x) := \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|}, \quad \delta f := f(x + \delta x) - f(x)$$

$\hat{\kappa}_f(x)$  is called the absolute condition number of  $f$  at  $x$ .

## Absolute sensitivity measures

Let  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ .

A sensible measure of sensitivity of  $f$  at  $x$  is perturbation-based:

$$\frac{\|f(x + \delta x) - f(x)\|}{\|\delta x\|}.$$

As with derivatives, we can measure the sensitivity at  $x$  by taking limits:

$$\hat{\kappa}_f(x) := \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\|}{\|\delta x\|}, \quad \delta f := f(x + \delta x) - f(x)$$

$\hat{\kappa}_f(x)$  is called the absolute condition number of  $f$  at  $x$ .

Note that condition numbers are properties of the map  $f$  and *not* of an algorithmic or finite-precision implementation.

## Relative sensitivity measures

Recall: floating-point arithmetic makes relative errors, not absolute ones.

Thus, absolute condition numbers have limited utility for understanding numerical stability.

## Relative sensitivity measures

Recall: floating-point arithmetic makes relative errors, not absolute ones.

Thus, absolute condition numbers have limited utility for understanding numerical stability.

The relative condition number of  $f$  at  $x$  is defined as

$$\begin{aligned}\kappa_f(x) &:= \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\frac{\|\delta f\|}{\|f(x)\|}}{\frac{\|\delta x\|}{\|x\|}} \\ &= \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\| \|x\|}{\|\delta x\| \|f(x)\|}\end{aligned}$$

Again, this is a (mathematical) property of  $f$ , and not of an algorithmic implementation.

## Relative sensitivity measures

Recall: floating-point arithmetic makes relative errors, not absolute ones.

Thus, absolute condition numbers have limited utility for understanding numerical stability.

The relative condition number of  $f$  at  $x$  is defined as

$$\begin{aligned}\kappa_f(x) &:= \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\frac{\|\delta f\|}{\|f(x)\|}}{\frac{\|\delta x\|}{\|x\|}} \\ &= \lim_{\delta \rightarrow 0} \sup_{\|\delta x\| \leq \delta} \frac{\|\delta f\| \|x\|}{\|\delta x\| \|f(x)\|}\end{aligned}$$

Again, this is a (mathematical) property of  $f$ , and not of an algorithmic implementation.

Problems (functions  $f$ ) with “small” condition numbers are *well-conditioned*.

Problems (functions  $f$ ) with “large” condition numbers are *ill-conditioned*.

Just like derivatives and differentials, the condition number is a sensitivity of relative errors:

$$\frac{\|\delta f\|}{\|f(x)\|} \lesssim \kappa_f(x) \frac{\|\delta x\|}{\|x\|}.$$

# Examples

## Example

If  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$  is smooth, then condition numbers are norms of Jacobians.

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

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

$$J(x) := \frac{\partial f}{\partial x}(x),$$

$$(\underline{J(x)})_{ij} = \frac{\partial f_i}{\partial x_j}, \quad \begin{matrix} i \in [m] \\ j \in [n] \end{matrix}$$

absolute cond. # :  $\frac{\|f\|}{\|x\|} \lesssim \|J(x)\|$  induced norm

# Examples

## Example

$f : \mathbb{C} \rightarrow \mathbb{C}$  defined by  $f(x) = ax$ .

Compute  $\hat{\kappa}_f(x)$  and  $\kappa_f(x)$ .

$$\hat{\kappa}_f(x) = |f'(x)| = |a|$$

$$\kappa_f(x) = |f'(x)| \cdot \frac{|x|}{|f(x)|} = |a| \cdot \frac{|x|}{|ax|} = 1 \quad (\text{for any } a)$$

# Examples

## Example

$f : \mathbb{C} \rightarrow \mathbb{C}$  defined by  $f(x) = x^p$  for arbitrary  $p > 0$ .

Compute  $\hat{\kappa}_f(x)$  and  $\kappa_f(x)$ .

$$\hat{\kappa}_f(x) = |f'(x)| = p|x^{p-1}|$$

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

$p \gg 1$ : worse conditioned function  
(compared to  $p \ll 1$ )



## Examples

### Example

Compute condition numbers for  $f: \mathbb{C}^n \rightarrow \mathbb{C}^n$  defined by  $f(x) = Ax$  for invertible  $A \in \mathbb{C}^{n \times n}$ .

$$\frac{\partial f}{\partial x} = \underline{A}, \quad \hat{K}_f(x) = \|\underline{A}\| \quad (\text{say } \|\underline{A}\|_2 \text{ if } \|\cdot\| \text{ on } \mathbb{C}^n \text{ and } \ell^2 \text{ norm})$$

$$K_f(x) = \|A\| \cdot \frac{\|x\|}{\|Ax\|}$$

$$\stackrel{y=Ax}{\leq} \|A\| \cdot \frac{\|A^{-1}y\|}{\|y\|} \leq \|A\| \cdot \|A^{-1}\|$$

$$\text{E.g., for } \ell^2 \text{ norms: } K_f(x) \leq \|A\|_2 \cdot \|A^{-1}\|_2$$

## Linear problems

The (relative) condition number of the linear map  $x \mapsto Ax$ , for invertible  $A$ , is bounded by

$$\sup_{x \in \mathbb{C}^n} \kappa(x) = \|A\| \|A^{-1}\|.$$

## Linear problems

The (relative) condition number of the linear map  $x \mapsto Ax$ , for invertible  $A$ , is bounded by

$$\sup_{x \in \mathbb{C}^n} \kappa(x) = \|A\| \|A^{-1}\|.$$

By a similar argument, given  $A$  and  $b$ , the condition number of the problem that finds the solution  $x$  to

$$Ax = b, \rightarrow x = A^{-1}b$$

is (bounded by)  $\|A\| \|A^{-1}\|$ .

## Linear problems

The (relative) condition number of the linear map  $x \mapsto Ax$ , for invertible  $A$ , is bounded by

$$\sup_{x \in \mathbb{C}^n} \kappa(x) = \|A\| \|A^{-1}\|.$$

By a similar argument, given  $A$  and  $b$ , the condition number of the problem that finds the solution  $x$  to

$$Ax = b,$$

is (bounded by)  $\|A\| \|A^{-1}\|$ .

More generally, given invertible  $A$ , the (matrix) condition number of  $A$  is defined as

$$\kappa(A) = \|A\| \|A^{-1}\|.$$

if using  $\ell^2$  norms:  $\kappa(\underline{A}) = \frac{\sigma_{\max}(\underline{A})}{\sigma_{\min}(\underline{A})} \geq 1$

# Numerical algorithms

Given  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ , we wish to understand how roundoff errors affect evaluation of  $f$ .

This should depend on

- Loss of accuracy due to finite precision
- Conditioning of  $f$

We need condition numbers here since we'll make a small relative mistake in  $x$  (roundoff/truncation), so we must understand what kind of relative mistake we make in  $f(x)$ .

# Numerical algorithms

Given  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ , we wish to understand how roundoff errors affect evaluation of  $f$ .

This should depend on

- Loss of accuracy due to finite precision
- Conditioning of  $f$

We need condition numbers here since we'll make a small relative mistake in  $x$  (roundoff/truncation), so we must understand what kind of relative mistake we make in  $f(x)$ .

However, things are somewhat more complicated because our implementation is never exactly  $f$ .

Let  $\tilde{f} : \mathbb{C}^n \rightarrow \mathbb{C}^m$  denote the actual algorithmic implementation of  $f$ .

For example, the numerical implementation of  $f(x) = 1 + x$  might be  $\tilde{f}(x) = \text{fl}(1) \oplus \text{fl}(x)$ , where  $\oplus$  is some implementation of the addition of two floating point numbers.

# Numerical algorithms

Given  $f : \mathbb{C}^n \rightarrow \mathbb{C}^m$ , we wish to understand how roundoff errors affect evaluation of  $f$ .

This should depend on

- Loss of accuracy due to finite precision
- Conditioning of  $f$

We need condition numbers here since we'll make a small relative mistake in  $x$  (roundoff/truncation), so we must understand what kind of relative mistake we make in  $f(x)$ .

However, things are somewhat more complicated because our implementation is never exactly  $f$ .

Let  $\tilde{f} : \mathbb{C}^n \rightarrow \mathbb{C}^m$  denote the actual algorithmic implementation of  $f$ .

For example, the numerical implementation of  $f(x) = 1 + x$  might be  $\tilde{f}(x) = \text{fl}(1) \oplus \text{fl}(x)$ , where  $\oplus$  is some implementation of the addition of two floating point numbers.

We might hope that

$$\frac{\|f(x) - \tilde{f}(x)\|}{\|f(x)\|} = \mathcal{O}(\epsilon_{\text{mach}}),$$

and one suspects that this condition might be tied to conditioning of  $f$ .

## Forward error

Suppose  $f$  is the “exact” procedure, and  $\tilde{f}$  is the “approximate” (computational) procedure.

The **forward error** at  $x$  is the relative error committed by  $\tilde{f}$ :

$$\text{FE}(x) = \frac{\|f(x) - \tilde{f}(x)\|}{\|f(x)\|}$$



## Forward error

Suppose  $\mathbf{f}$  is the “exact” procedure, and  $\tilde{\mathbf{f}}$  is the “approximate” (computational) procedure.

The **forward error** at  $\mathbf{x}$  is the relative error committed by  $\tilde{\mathbf{f}}$ :

$$\text{FE}(\mathbf{x}) = \frac{\|\mathbf{f}(\mathbf{x}) - \tilde{\mathbf{f}}(\mathbf{x})\|}{\|\mathbf{f}(\mathbf{x})\|}$$

This notion of error is typically used to understand stability as follows:

$$\text{FE}(\mathbf{x}) \leq \frac{\|\tilde{\mathbf{f}}(\mathbf{x}) - \mathbf{f}(\tilde{\mathbf{x}})\|}{\|\mathbf{f}(\mathbf{x})\|} + \frac{\|\mathbf{f}(\tilde{\mathbf{x}}) - \mathbf{f}(\mathbf{x})\|}{\|\mathbf{f}(\mathbf{x})\|},$$

where we’ve introduced an intermediate quantity  $\tilde{\mathbf{x}}$ . Ideally, this quantity is a “small” perturbation of  $\mathbf{x}$ .

## Forward error

Suppose  $f$  is the “exact” procedure, and  $\tilde{f}$  is the “approximate” (computational) procedure.

The **forward error** at  $x$  is the relative error committed by  $\tilde{f}$ :

$$\text{FE}(x) = \frac{\|f(x) - \tilde{f}(x)\|}{\|f(x)\|}$$

This notion of error is typically used to understand stability as follows:

$$\text{FE}(x) \leq \frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} + \frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|},$$

where we’ve introduced an intermediate quantity  $\tilde{x}$ . Ideally, this quantity is a “small” perturbation of  $x$ . In particular, if  $\tilde{x}$  is “close” to  $x$ , then the latter term is

$$\frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|} \lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|}.$$

All of this motivates a requirement on the behavior first term, which is purely an algorithmic concern.

## Forward stability

### Definition

An algorithm  $\tilde{f}$  is *forward stable* at  $x$  if there is some  $\epsilon > 0$  so that,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} \lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|},$$

whenever  $\|x - \tilde{x}\|/\|x\| \leq \epsilon$ .

(Typically, this definition is applied/useful for  $\epsilon = \mathcal{O}(\epsilon_{\text{mach}})$ .)

A forward stable algorithm provides “approximately the right answer to a closely related question”.

## Forward stability

### Definition

An algorithm  $\tilde{f}$  is *forward stable* at  $x$  if there is some  $\epsilon > 0$  so that,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} \lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|},$$

whenever  $\|x - \tilde{x}\|/\|x\| \leq \epsilon$ .

(Typically, this definition is applied/useful for  $\epsilon = \mathcal{O}(\epsilon_{\text{mach}})$ .)

A forward stable algorithm provides “approximately the right answer to a closely related question”.

If an algorithm is forward stable and the procedure is well-conditioned, then we have our desired error:

$$\begin{aligned} \text{FE}(x) &\leq \frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} + \frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|} \\ &\lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|} + \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|} \stackrel{\epsilon \sim \epsilon_{\text{mach}}}{\lesssim} \kappa(x) \epsilon_{\text{mach}}. \end{aligned}$$

## Forward stability

### Definition

An algorithm  $\tilde{f}$  is *forward stable* at  $x$  if there is some  $\epsilon > 0$  so that,

$$\frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} \lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|},$$

whenever  $\|x - \tilde{x}\|/\|x\| \leq \epsilon$ .

(Typically, this definition is applied/useful for  $\epsilon = \mathcal{O}(\epsilon_{\text{mach}})$ .)

A forward stable algorithm provides “approximately the right answer to a closely related question”.

If an algorithm is forward stable and the procedure is well-conditioned, then we have our desired error:

$$\begin{aligned} \text{FE}(x) &\leq \frac{\|\tilde{f}(x) - f(\tilde{x})\|}{\|f(x)\|} + \frac{\|f(\tilde{x}) - f(x)\|}{\|f(x)\|} \\ &\lesssim \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|} + \kappa(x) \frac{\|x - \tilde{x}\|}{\|x\|} \stackrel{\epsilon \sim \epsilon_{\text{mach}}}{\lesssim} \kappa(x) \epsilon_{\text{mach}}. \end{aligned}$$

This required error estimation procedure, establishment of forward stability of  $f$ , is *forward error analysis*. Showing forward stability is frequently technical and difficult.

## Backward stability

A dual, but somewhat stranger+stronger notion of stability is backward stability.

From this point of view, we essentially assume that the algorithm produces “exactly the right answer to a closely related question”.

### Definition

An algorithm  $\tilde{f}$  is backward stable at  $x$  if we have

$$\tilde{f}(x) = f(\tilde{x}), \tag{1}$$

for some  $\tilde{x}$  satisfying  $\|x - \tilde{x}\| = \|x\|\mathcal{O}(\epsilon)$ .

(Again, to make this practical, one requires  $\epsilon = \epsilon_{\text{mach}}$ .)

An ancillary definition: the corresponding notion of backward error of  $\tilde{f}$  at  $x$  is,

$$\text{BE}(x) = \inf \left\{ \frac{\|x - \tilde{x}\|}{\|x\|} \mid \tilde{f}(x) = f(\tilde{x}) \right\}.$$

## Backward stability

A dual, but somewhat stranger+stronger notion of stability is backward stability.

From this point of view, we essentially assume that the algorithm produces “exactly the right answer to a closely related question”.

### Definition

An algorithm  $\tilde{f}$  is backward stable at  $x$  if we have

$$\tilde{f}(x) = f(\tilde{x}), \quad (1)$$

for some  $\tilde{x}$  satisfying  $\|x - \tilde{x}\| = \|x\|\mathcal{O}(\epsilon)$ .

(Again, to make this practical, one requires  $\epsilon = \epsilon_{\text{mach}}$ .)

An ancillary definition: the corresponding notion of backward error of  $\tilde{f}$  at  $x$  is,

$$\text{BE}(x) = \inf \left\{ \frac{\|x - \tilde{x}\|}{\|x\|} \mid \tilde{f}(x) = f(\tilde{x}) \right\}.$$

If  $\tilde{f}$  is backward stable, then showing accuracy of the algorithm  $\tilde{f}$  is easier.

Estimating error by establishing backward stability is *backward error analysis*. Backward stability is frequently easier to show than forward stability, but forward stable algorithms need not be backward stable.

# Examples

Assume some floating-point axioms:

1. For each  $x \in \mathbb{R}$ , then  $fl(x) = x(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .
2. For floating-points numbers  $x, y \in \mathbb{R}$ , then  $x \odot y = (x \cdot y)(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .    ( $\cdot = +, -, \times$ )



# Examples

Assume some floating-point axioms:

1. For each  $x \in \mathbb{R}$ , then  $fl(x) = x(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .
2. For floating-points numbers  $x, y \in \mathbb{R}$ , then  $x \odot y = (x \cdot y)(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ . ( $\cdot = +, -, \times$ )

## Example

Given  $x, y \in \mathbb{R}$ , is the implementation  $\tilde{f}(x, y) := fl(x) \ominus fl(y)$  of  $f(x, y) = x - y$  backward stable?

# Examples

Assume some floating-point axioms:

1. For each  $x \in \mathbb{R}$ , then  $fl(x) = x(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .
2. For floating-points numbers  $x, y \in \mathbb{R}$ , then  $x \odot y = (x \cdot y)(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ . ( $\cdot = +, -, \times$ )

## Example

Given  $x \in \mathbb{R}$ , is the implementation  $\tilde{f}(x) := 1 \oplus fl(x)$  of  $f(x) = 1 + x$  backward stable?

# Examples

Assume some floating-point axioms:

1. For each  $x \in \mathbb{R}$ , then  $fl(x) = x(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .
2. For floating-points numbers  $x, y \in \mathbb{R}$ , then  $x \odot y = (x \cdot y)(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ . ( $\cdot = +, -, \times$ )

## Example

Given  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ , the floating-point implementation of  $f(\mathbf{x}, \mathbf{y}) = \mathbf{y}^T \mathbf{x}$  is backward stable.

# Examples

Assume some floating-point axioms:

1. For each  $x \in \mathbb{R}$ , then  $fl(x) = x(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ .
2. For floating-points numbers  $x, y \in \mathbb{R}$ , then  $x \odot y = (x \cdot y)(1 + \epsilon)$  for  $|\epsilon| \leq \mathcal{O}(\epsilon_{\text{mach}})$ . ( $\odot = +, -, \times$ )

## Example

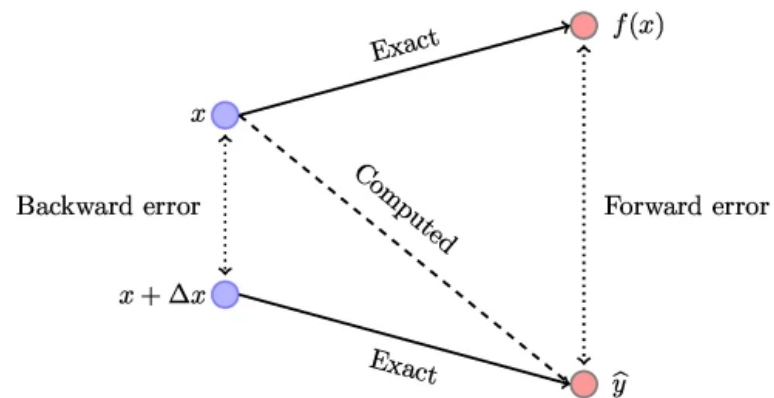
The floating-point implementation of  $f(x, y) = xy^T$  is *not* backward stable.

## Some punch lines

*Conditioning* is a property of mathematical maps/functions.

*Stability* is a property of an algorithmic implementation.

Well-conditioned functions *and* stable implementations yield reliable accuracy.



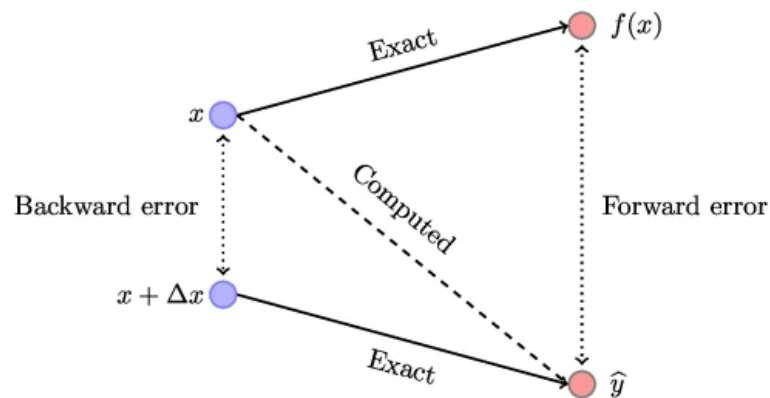
Source: <https://nhigham.com/2020/03/25/what-is-backward-error>

## Some punch lines

*Conditioning* is a property of mathematical maps/functions.

*Stability* is a property of an algorithmic implementation.

Well-conditioned functions *and* stable implementations yield reliable accuracy.



Source: <https://nhigham.com/2020/03/25/what-is-backward-error>

Studying conditioning of maps is essential to understanding best-possible behavior of algorithms. An ill-conditioned operation with a stable implementation wouldn't yield reliably accurate results.

# Eigenvalue conditioning

Weren't we interested in eigenvalues....?

Consider  $\mathbf{A} \in \mathbb{C}^{n \times n}$ , along with the map  $\lambda : \mathbb{C}^{n \times n} \rightarrow \mathbb{C}$  that computes an eigenvalue of  $A$ :

$$\mathbf{A}\mathbf{v} = \lambda(\mathbf{A})\mathbf{v}, \quad \text{for some } \mathbf{v} \neq \mathbf{0}$$

What is the conditioning of this operation?

# Eigenvalue conditioning

Weren't we interested in eigenvalues....?

Consider  $\mathbf{A} \in \mathbb{C}^{n \times n}$ , along with the map  $\lambda : \mathbb{C}^{n \times n} \rightarrow \mathbb{C}$  that computes an eigenvalue of  $\mathbf{A}$ :

$$\mathbf{A}\mathbf{v} = \lambda(\mathbf{A})\mathbf{v}, \quad \text{for some } \mathbf{v} \neq \mathbf{0}$$




What is the conditioning of this operation?

## Theorem (Bauer-Fike)

*Assume  $\mathbf{A} \in \mathbb{C}^{n \times n}$  is diagonalizable with eigenvector matrix  $\mathbf{V} \in \mathbb{C}^{n \times n}$ . Then, using the 2-norm, the absolute condition number of computing eigenvalues is  $\tilde{\kappa}_\lambda(\mathbf{A}) = \kappa(\mathbf{V})$ .*



## References I

-  Atkinson, Kendall (1989). *An Introduction to Numerical Analysis*. New York: Wiley. ISBN: 978-0-471-62489-9.
-  Salgado, Abner J. and Steven M. Wise (2022). *Classical Numerical Analysis: A Comprehensive Course*. Cambridge: Cambridge University Press. ISBN: 978-1-108-83770-5. DOI: 10.1017/9781108942607.
-  Trefethen, Lloyd N. and David Bau (1997). *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics. ISBN: 0-89871-361-7.