Advanced Numerical Algorithms (MU4IN920)

Lecture 4: Floating-point summation algorithms

Stef Graillat

Sorbonne Université

Summary of the previous lecture

Fast Fourier Transform (FFT):

- A very efficient algorithm (in $\mathcal{O}(n \log n)$)
- It has applications in many fields (imaging, signal processing, computer algebra, number theory, etc.)
- There are other algorithms than the Cooley-Tukey one
- If the data are real, a discrete cosine transform is used instead

Outline

Outline

Representation

A floating-point number x is represented in base B by:

- its sign s_x (0 for x positive, 1 for x negative)
- its mantissa (significand) m_x of $n+1$ digits
- its exponent e_x, an integer of k digits between $e_{\min}$ and $e_{\max}$

such that

$$x = (-1)^{s_x} \times m_x \times B^{e_x}$$

with $m_x = x_0 . x_1 x_2 \ldots x_n$ where $x_i \in \{0, 1, \ldots, B-1\}$.

To obtain the best possible accuracy with a fixed-length mantissa n, the mantissa is normalized, that is to say, $x_0 \neq 0$, $m_x \in [1, B[$.

A special code is necessary for 0.

Before 1985

The result of an operation may differ depending on the language, the compiler and the computer architecture.

computer	B	n	$e_{\min}$	$e_{\max}$
Cray 1	2	48	−8192	8191
	2	96	−8192	8191
DEC VAX	2	53	−1023	1023
	2	56	−127	127
HP 28 and 48G	10	12	−499	499
IBM 3090	16	6	−64	63
	16	14	−64	63
	16	28	−64	63

Impossible to write portable numerical code.

Revolution in 1985: IEEE-754 standard

A standard defines the representation of data and the behavior of basic operations in in floating-point arithmetic

This standard defines:

- formats for data
- special values
- rounding modes
- accuracy of basic operations
- rules for conversion

754 *ANSI/IEEE Standard for Binary Floating-Point Arithmetic* in 1985
854 *ANSI/IEEE Standard for Radix-Independent Floating-Point Arithmetic* in 1987 (where $B = 2$ or 10)

IEEE 754-2008, *IEEE Standard for Binary Floating-Point Arithmetic* in 2008
IEEE 754-2019, *IEEE Standard for Binary Floating-Point Arithmetic* in 2019

Advantages of IEEE-754 standard

- make possible to write portable programs
- make programs deterministic from one computer to another
- keep mathematical properties
- perform correctly rounded operations
- perform reliable conversions
- facilitate the construction of mathematical proof
- facilitate the use of exceptions
- facilitate comparisons between numbers
- support directed roundings useful for interval arithmetic

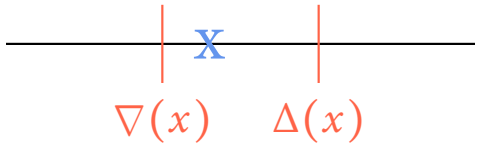
IEEE-754 standard: need for rounding

If x and y are 2 representable numbers, then the result of an operation $res = x \odot y$ is not, in general, a representable number.

For example, in base $B = 10$, the number $1/3$ is not representable with a finite number of digits.

It is necessary to round the result, that is to say to return one of the closest representable numbers.

IEEE-754 standard: rounding modes



The norm proposes 4 rounding modes:

- rounding toward $+\infty$ denoted $\Delta(x)$: return the smallest floating-point number greater or equal the exact result x
- rounding toward $-\infty$ denoted $\nabla(x)$: return the largest floating-point number less or equal the exact result x
- rounding toward 0, denoted $\mathcal{Z}(x)$: return $\Delta(x)$ for negative numbers and $\nabla(x)$ for positive numbers
- rounding to the nearest, denoted $\circ(x)$: return the nearest floating-point number of the exact result x (breaks ties by rounding to the nearest even floating-point number)

The 3 first rounding modes are called directed rounding modes.

IEEE-754 standard: correctly rounding

Let x and y be two representable number, $\odot$ be on operations $+$, $-$, $\times$, $/$ and $\diamond$ a rounding mode.

The standard requires that the result of the computation $x \odot y$ be equal to $\diamond(x \odot_{exact} y)$. The result must be similar to the one obtain by computing with infinite precision and then rounding this result.

Similar for square root.

This property is called correctly rounding.

The standard describes an algorithm for addition, subtraction, multiplication, division and square root and requires that the implementation produces the same result as those algorithms.

Accuracy of computations

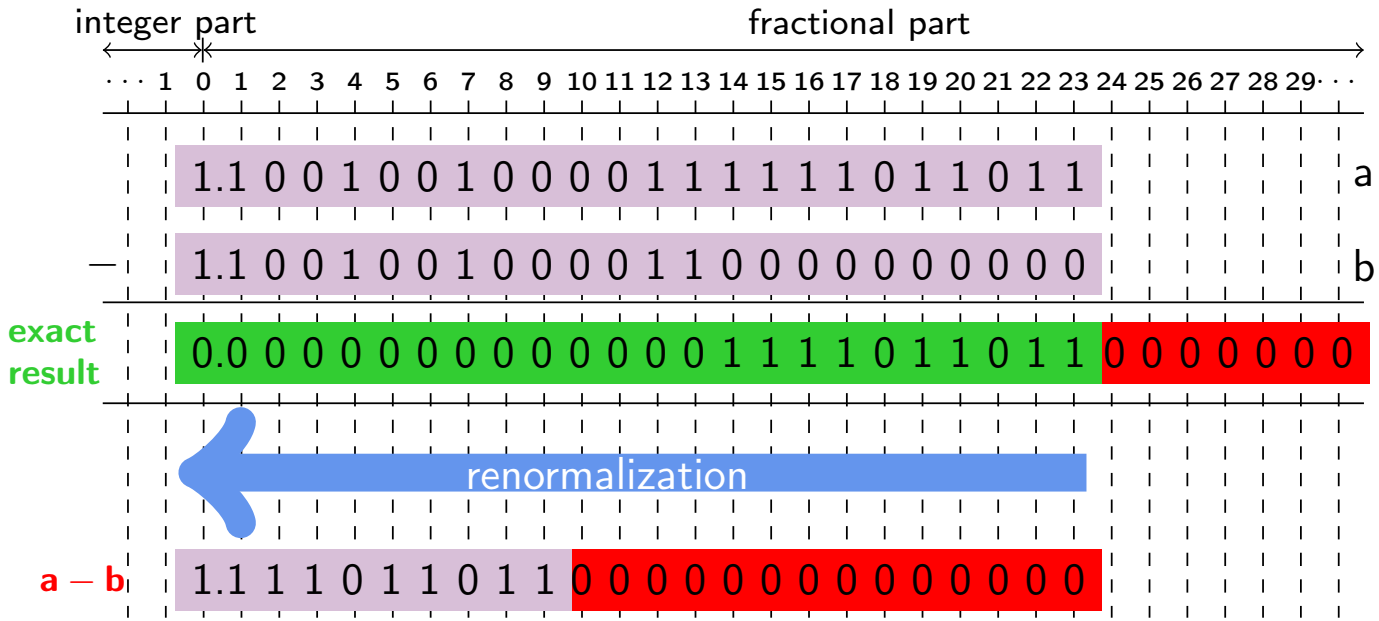
At each rounding, we loose a bit of accuracy, we call this rounding errors.

Even if an operation returns the best possible result (correctly rounded result), a sequence of computation can lead to huge errors due to the accumulation of rounding errors.

The two main sources of rounding errors during a computation are the *cancellation* and the absorption.

Cancellation

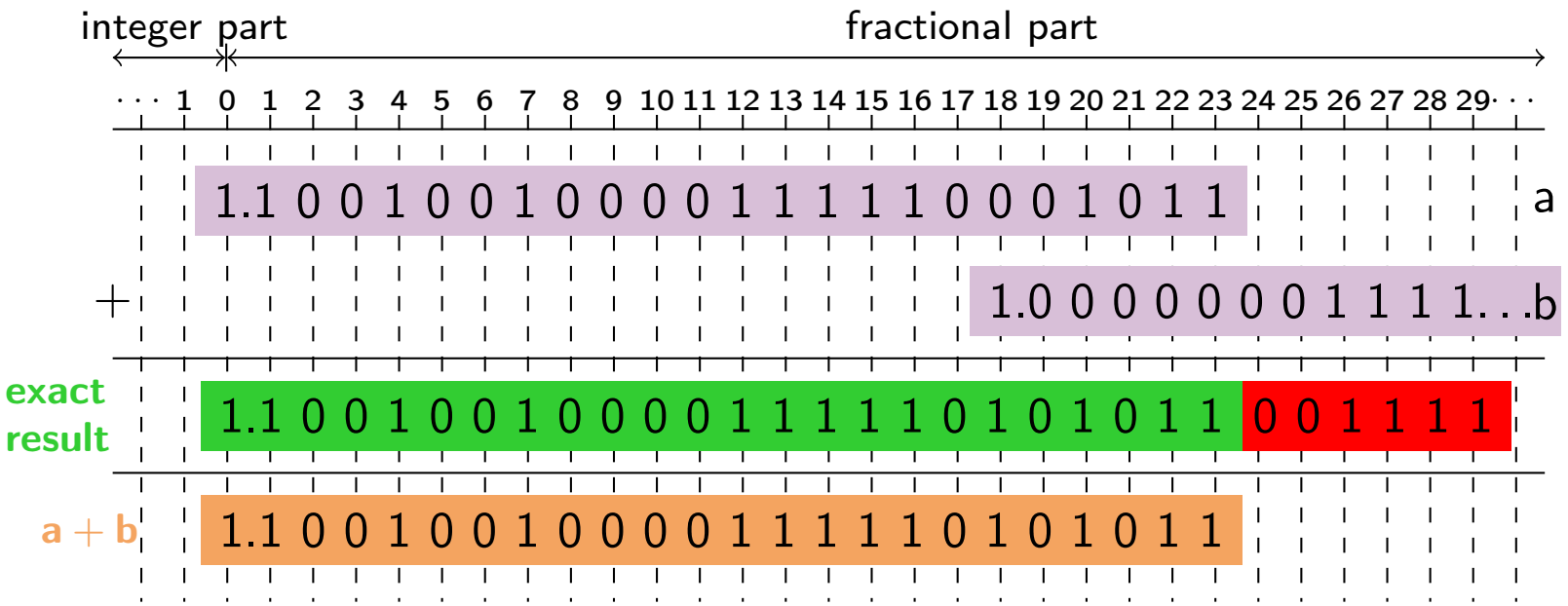
When we subtract two numbers very close.



If operands are results of previous computations with rounding errors, the 0's add on the right (red part) are wrong. The cancellation is said to be catastrophic when there is nearly no significant digits

Absorption

When we add 2 numbers of different order of magnitude we can ”loose” all the information about the smallest one.



Combine with a *cancellation* it can be catastrophic. for example in single precision, if $a = 1$ and $b = 2^{-30}$ then: $a \oplus b = 1$ and so $(a \oplus b) \ominus a = 0$.

Standard model of floating-point arithmetic

IEEE 754 standard (1985,2008,2019)

- The arithmetic operations ops $(+, -, \times, /, \sqrt{})$ are performed as if they were calculated in infinite precision and then rounded off
- Default: rounded to nearest

Type	Size	Mantissa	Exponent	Unit roundoff	Interval
binary32 (simple)	32 bits	23+1 bits	8 bits	$\mathbf{u} = 2^{-24} \approx 5,96 \times 10^{-8}$	$\approx 10^{\pm 38}$
binary64 (double)	64 bits	52+1 bits	11 bits	$\mathbf{u} = 2^{-53} \approx 1,11 \times 10^{-16}$	$\approx 10^{\pm 308}$

Let $x, y \in \mathbb{F}$,

$$\mathrm{fl}(x \circ y) = (x \circ y)(1+\delta), \quad |\delta| \leq \mathbf{u}, \quad \circ \in \{+, -, \cdot, /\}$$

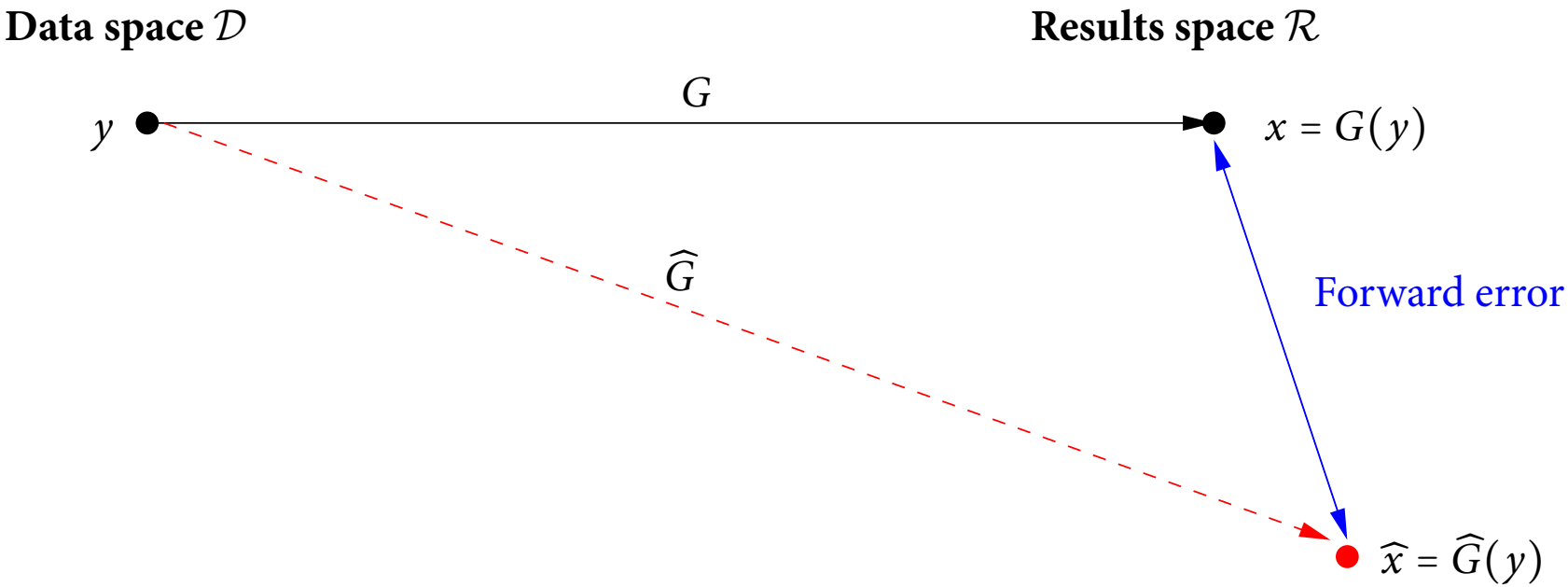
Understanding the difficulties when computing with finite precision

- Controlling the effects of finite precision:
  - How to measure the difficulty of solving the problem?
  - How to characterize the reliability of the algorithm?
  - How to estimate the accuracy of the computed solution?
- Limiting the effects of finite precision
  - How to improve the accuracy of the solution?

How to answer these questions?

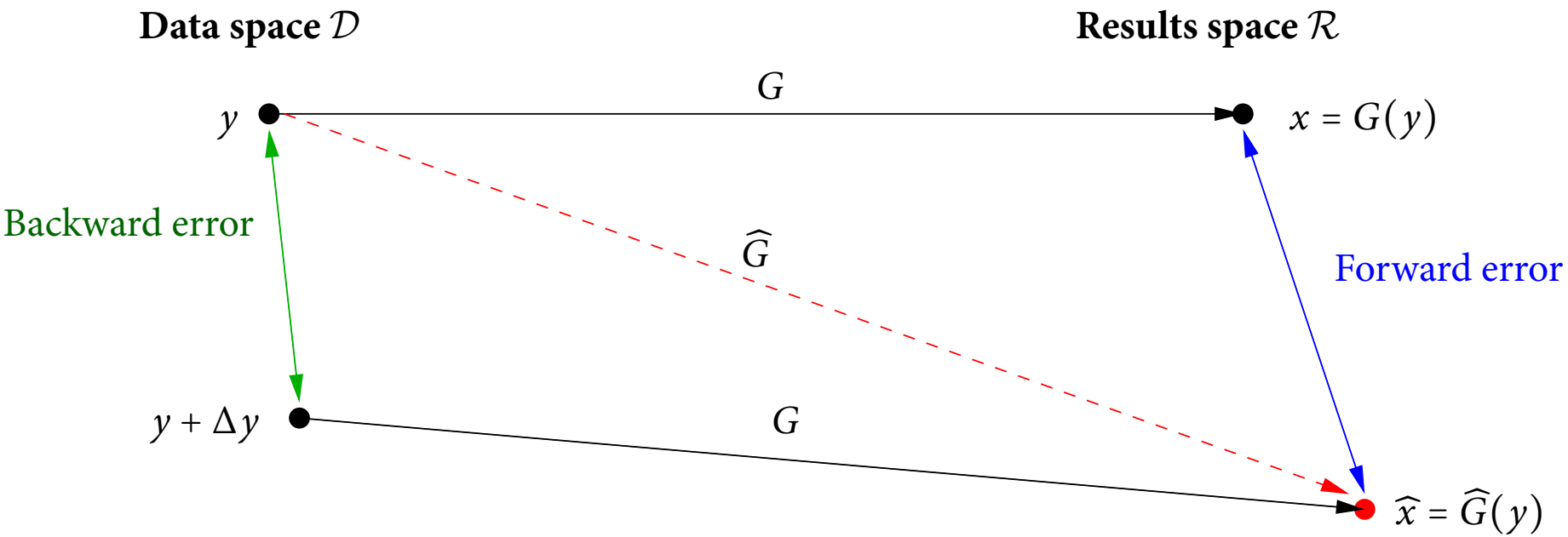
Outline

Error analysis (Wilkinson, Higham)



- Forward error analysis

Error analysis (Wilkinson, Higham)



- Forward error analysis
- Backward error analysis
  Identify $\widehat{x}$ as the solution of a perturbed problem: $\widehat{x} = G(y + \Delta y)$.

Advantages of backward error analysis

- **How to measure the difficulty of solving the problem ?**
  Condition number measures the sensitivity of the solution to perturbation in the data

  Condition number : $\mathrm{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\|\Delta y\| \leq \varepsilon} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}$

Advantages of backward error analysis

- **How to measure the difficulty of solving the problem ?**
  Condition number measures the sensitivity of the solution to perturbation in the data

  Condition number : $\mathrm{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\|\Delta y\| \leq \varepsilon} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}$

- **How to appreciate the reliability of the algorithm?**
  Backward error measures the distance between the problem we solved and the initial problem.

  Backward error : $\eta(\widehat{x}) = \min_{\Delta y \in \mathcal{D}} \{ \|\Delta y\|_{\mathcal{D}} : \widehat{x} = G(y + \Delta y) \}$

Advantages of backward error analysis

- **How to measure the difficulty of solving the problem ?**
  Condition number measures the sensitivity of the solution to perturbation in the data

  Condition number : $\mathrm{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\|\Delta y\| \leq \varepsilon} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}$

- **How to appreciate the reliability of the algorithm?**
  Backward error measures the distance between the problem we solved and the initial problem.

  Backward error : $\eta(\widehat{x}) = \min_{\Delta y \in \mathcal{D}} \{ \|\Delta y\|_{\mathcal{D}} : \widehat{x} = G(y + \Delta y) \}$

- **How to estimate the accuracy of the computed solution?**
  At first order, the rule of thumb:

  $$\text{forward error} \lesssim \text{condition number} \times \text{backward error.}$$

Advantages of backward error analysis

- **How to measure the difficulty of solving the problem ?**
  Condition number measures the sensitivity of the solution to perturbation in the data

  Condition number : $\mathrm{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\|\Delta y\| \leq \varepsilon} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}$

- **How to appreciate the reliability of the algorithm?**
  Backward error measures the distance between the problem we solved and the initial problem.

  Backward error : $\eta(\widehat{x}) = \mathbf{u} \longrightarrow$ backward stable

- **How to estimate the accuracy of the computed solution?**
  At first order, the rule of thumb:

  $$\text{forward error} \lesssim \text{condition number} \times \text{backward error.}$$

Advantages of backward error analysis

- **How to measure the difficulty of solving the problem ?**
  Condition number measures the sensitivity of the solution to perturbation in the data

  Condition number : $\mathrm{cond}(P, y) := \lim_{\varepsilon \to 0} \sup_{\|\Delta y\| \leq \varepsilon} \left\{ \frac{\|\Delta x\|_{\mathcal{R}}}{\|\Delta y\|_{\mathcal{D}}} \right\}$

- **How to appreciate the reliability of the algorithm?**
  Backward error measures the distance between the problem we solved and the initial problem.

  Backward error : $\eta(\widehat{x}) = \mathbf{u} \longrightarrow$ backward stable

- **How to estimate the accuracy of the computed solution?**
  At first order, the rule of thumb:

  $$\text{forward error} \lesssim \text{condition number} \times \mathbf{u}.$$

Achieving more accuracy with compensated algorithms

Key tools for accurate computation

- fixed length expansions libraries: double-double (Briggs, Bailey, Hida, Li), quad-double (Bailey, Hida, Li)
- arbitrary length expansions libraries: Priest, Shewchuk
- arbitrary multiprecision libraries: MP, MPFUN/ARPREC, MPFR
- compensated algorithms (e.g. Kahan, Priest, Ogita-Rump-Oishi)

Error-free transformations (EFT) (Dekker, Knuth) are properties and algorithms to compute the elementary rounding errors,

$$a, b \in \mathbb{F}, \quad a \circ b = \mathrm{fl}(a \circ b) + e, \text{ and } e \in \mathbb{F}$$

EFT for the summation

$$x = a \oplus b \quad \Rightarrow \quad a + b = x + y \quad \text{with } y \in \mathbb{F},$$

Algorithms of Dekker (1971) and Knuth (1974)

Algorithm (EFT of the sum of 2 floating-point numbers with $|a| \geq |b|$)

function $[x, y] = \texttt{FastTwoSum}(a, b)$

$x = a \oplus b$

$y = (a \ominus x) \oplus b$

Algorithm (EFT of the sum of 2 floating-point numbers)

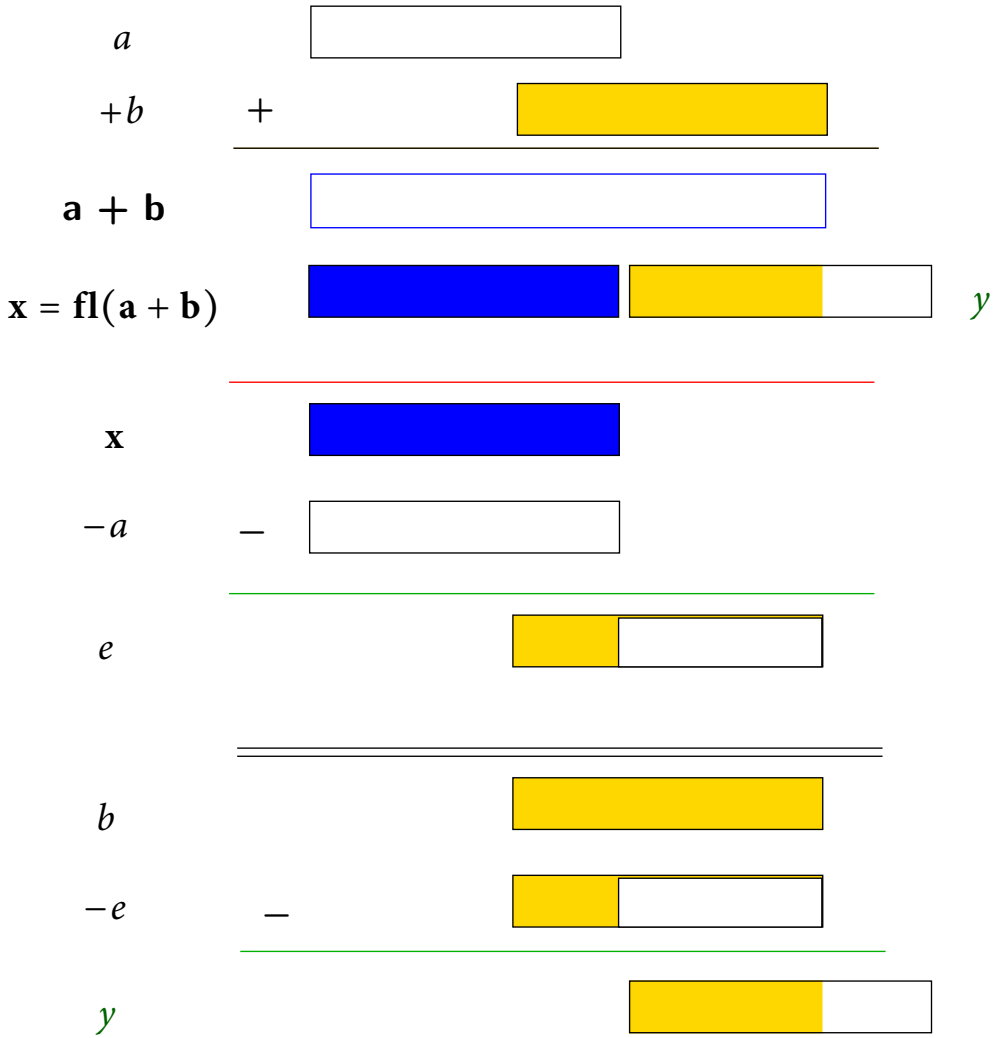
function $[x, y] = \texttt{TwoSum}(a, b)$

$x = a \oplus b$

$z = x \ominus a$

$y = (a \ominus (x \ominus z)) \oplus (b \ominus z)$

EFT for the summation



Error bound for EFT of the sum

Theorem

Let $a, b \in \mathbb{F}$ *and let* $x, y \in \mathbb{F}$ *such that* $[x, y] = \texttt{TwoSum}(a, b)$. *Then,*

$$a + b = x + y, \quad x = a \oplus b, \quad |y| \leq \mathbf{u}|x|, \quad |y| \leq \mathbf{u}|a + b|.$$

The algorithm `TwoSum` *requires* 6 *flops.*

EFT for the product (1/3)

$$x = a \otimes b \quad \Rightarrow \quad a \cdot b = x + y \quad \text{with } y \in \mathbb{F},$$

Algorithm `TwoProduct` by Veltkamp and Dekker (1971)

$$a = x + y \quad \text{and} \quad x \text{ and } y \text{ non overlapping with } |y| \leq |x|.$$

Algorithm (Error-free split of a floating-point number into two parts)

function $[x, y] = \texttt{Split}(a)$

factor $= 2^s + 1$ % $\mathbf{u} = 2^{-p}$, $s = \lceil p/2 \rceil$

$c = \texttt{factor} \otimes a$

$x = c \ominus (c \ominus a)$

$y = a \ominus x$

EFT for the product (2/3)

Algorithm (EFT of the product of 2 floating-point numbers)

function $[x, y] = \texttt{TwoProduct}(a, b)$

$x = a \otimes b$

$[a_1, a_2] = \texttt{Split}(a)$

$[b_1, b_2] = \texttt{Split}(b)$

$y = a_2 \otimes b_2 \ominus (((x \ominus a_1 \otimes b_1) \ominus a_2 \otimes b_1) \ominus a_1 \otimes b_2)$

Theorem

Let $a, b \in \mathbb{F}$ and let $x, y \in \mathbb{F}$ such that $[x, y] = \texttt{TwoProduct}(a, b)$. Then,

$$a \cdot b = x + y, \quad x = a \otimes b, \quad |y| \leq \mathbf{u}|x|, \quad |y| \leq \mathbf{u}|a \cdot b|,$$

The algorithm `TwoProduct` *requires* 17 *flops.*

EFT for the product (3/3)

$$x = a \otimes b \quad \Rightarrow \quad a \times b = x + y \quad \text{with } y \in \mathbb{F},$$

Given $a, b, c \in \mathbb{F}$,

- $\texttt{FMA}(a, b, c)$ is the nearest floating-point number $a \cdot b + c \in \mathbb{F}$

Algorithm (EFT of the product of 2 floating-point numbers)

function $[x, y] = \texttt{TwoProduct}(a, b)$

$x = a \otimes b$

$y = \texttt{FMA}(a, b, -x)$

The FMA is available for example on PowerPC, Itanium, Cell, Xeon Phi, Haswell processors.

Outline

Recursive summmation algorithm

Computation of $s = \sum_{i=1}^{n} p_i$

Algorithm (Classic summation algorithm)

function `res` = `Sum`(p)

$\sigma = 0$;

for $i = 1 : n$

$\sigma = \sigma \oplus p_i$

`res` = σ

Rounding error analysis(1/2)

Lemma 1

If $|\delta_i| \leq \mathbf{u}$, $\rho_i = \pm 1$ *for* $i = 1 : n$ *and* $n\mathbf{u} < 1$ *then*

$$\prod_{i=1}^{n} (1 + \delta_i)^{\rho_i} = 1 + \delta_n,$$

where

$$|\delta_n| \leq \frac{n\mathbf{u}}{1 - n\mathbf{u}} =: \gamma_n.$$

Rounding error analysis (2/2)

Theorem

With the previous notations, we have

$$|\texttt{res} - s| \leq \gamma_{n-1} \sum_{i=1}^{n} |p_i|.$$

Kahan's compensated summation algorithm

Algorithm (Kahan's algorithm)

function `res` = `SCompSum`(p)

$\sigma = 0$

$e = 0$

for $i = 1 : n$

$y = p_i \oplus e$

$[\sigma, e] =$ `FastTwoSum`(σ, y)

`res` $= \sigma$

Rounding error analysis

Theorem

With the previous notations, we have

$$|\mathtt{res} - s| \leq (2\mathbf{u} + \mathcal{O}(n\mathbf{u}^2)) \sum_{i=1}^{n} |p_i|.$$

Priest's doubly compensated summation algorithm

Algorithm (Priest's algorithm)

function res = DCompSum(p)
  sort the p_i such that $|p_1| \geq |p_2| \geq \cdots \geq |p_n|$
  $s = 0$
  $c = 0$
  for $i = 1 : n$
    $[y, u] =$ FastTwoSum(c, p_i)
    $[t, v] =$ FastTwoSum(y, s)
    $z = u \oplus v$
    $[s, c] =$ FastTwoSum(t, z)
  res = s

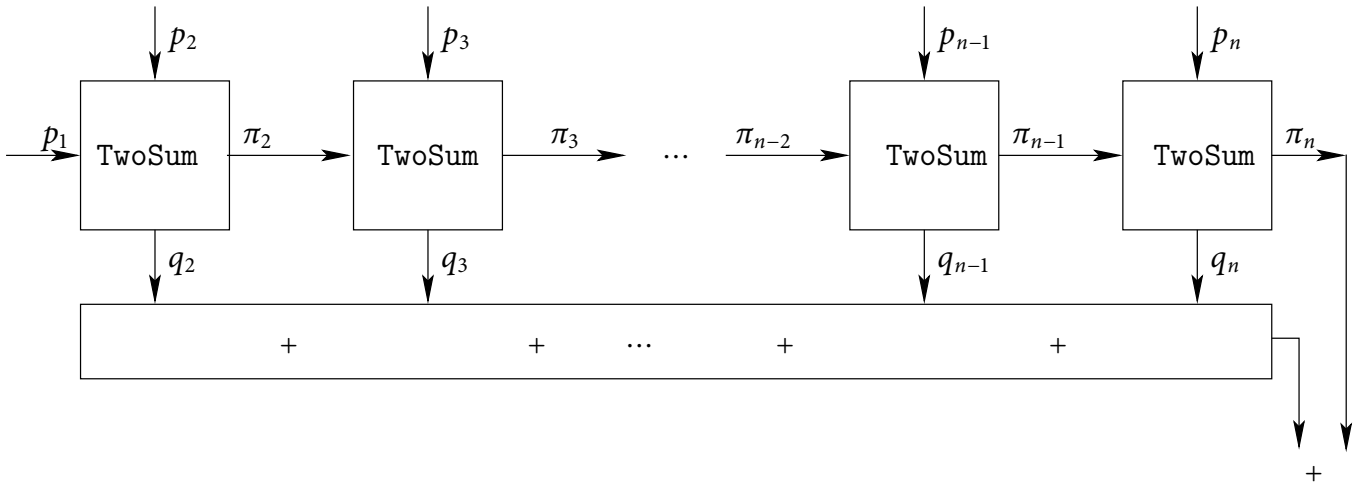
Rounding error analysis

Theorem

With the previous notations, we have

$$|\mathtt{res} - s| \leq 2\mathbf{u}|s|$$

Compensated algorithm of Ogita, Rump and Oishi



Compensated algorithm of Ogita, Rump and Oishi

Algorithm (Compensated algorithm)

function $\texttt{res} = \texttt{CompSum}(p)$

$\pi_1 = p_1$; $\sigma_1 = 0$;

for $i = 2 : n$

$[\pi_i, q_i] = \texttt{TwoSum}(\pi_{i-1}, p_i)$

$\sigma_i = \sigma_{i-1} \oplus q_i$

$\texttt{res} = \pi_n \oplus \sigma_n$

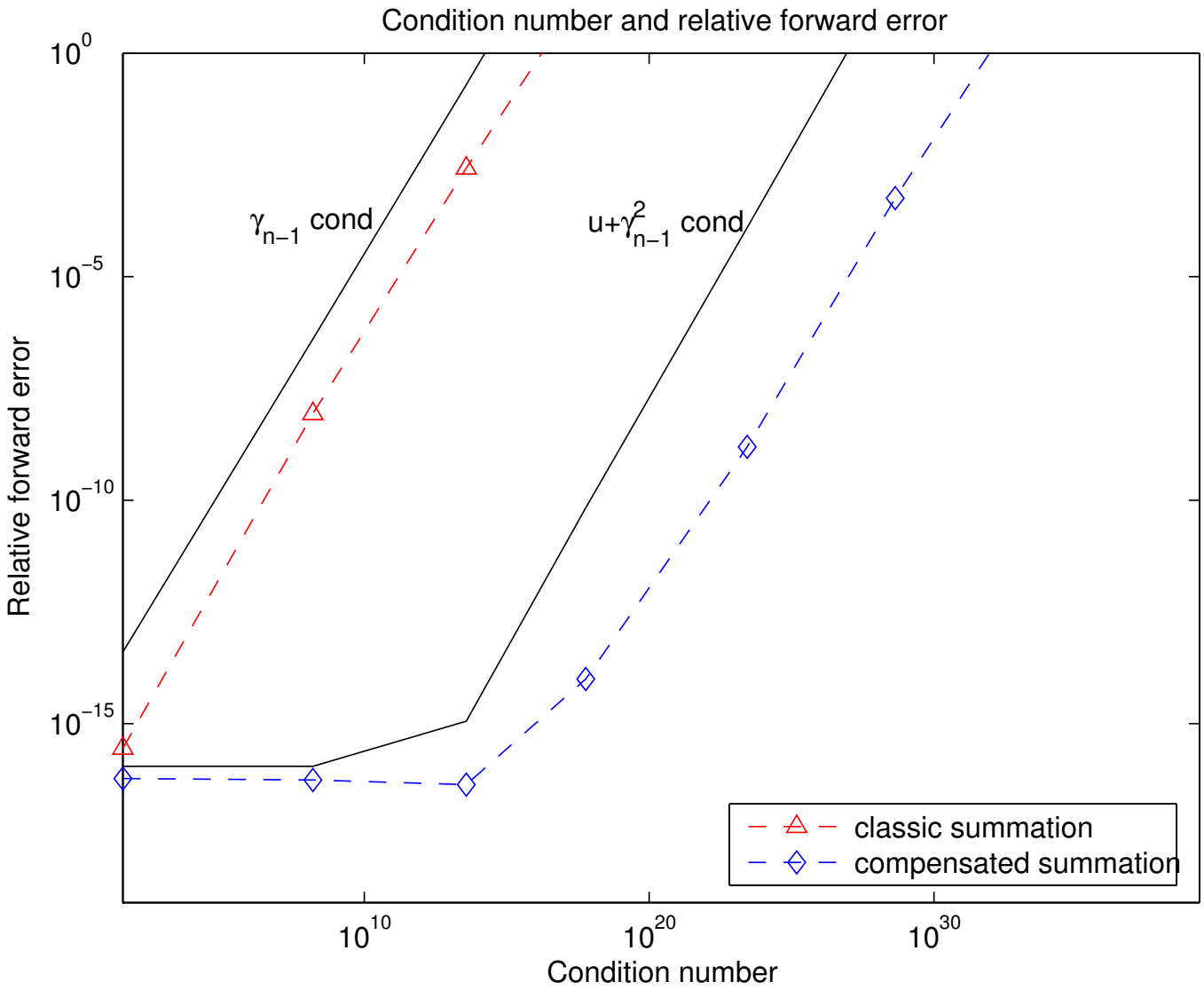
Compensated algorithm of Ogita, Rump and Oishi

Proposition

Let us apply `CompSum` *Algorithm to* $p_i \in \mathbb{F}$, $1 \leq i \leq n$. *Let* $s := \sum p_i$, $S := \sum |p_i|$ *and* $n\mathbf{u} < 1$. *Then, we have*

$$|\mathtt{res} - s| \leq \mathbf{u}|s| + \gamma_{n-1}^2 S. \tag{1}$$

Compensated algorithm of Ogita, Rump and Oishi



Outline

Compensated dot product (1/2)

Algorithm (Compensated dot product algorithm)

function $\texttt{res} = \texttt{CompDot}(x, y)$
  for $i = 1 : n$
    $[r_i, r_{n+i}] = \texttt{TwoProduct}(x_i, y_i)$
  $\texttt{res} = \texttt{CompSum}(r)$

Compensated dot product (2/2)

Algorithm (Compensated dot product algorithm)

function res = CompDot2(x, y)
  $[p, s]$ = TwoProduct(x_1, y_1)
  for $i = 2 : n$
    $[h, r]$ = TwoProduct(x_i, y_i)
    $[p, q]$ = TwoSum(p, h)
    $s = s \oplus (q \oplus r)$
  end
  res $= p \oplus s$

Proposition

Let floating point numbers $x_i, y_i \in \mathbb{F}, 1 \leq i \leq n$, be given and denote by res $\in \mathbb{F}$ *the result computed by Algorithm* CompDot2. *Then*

$$|\texttt{res} - x^T y| \leq \mathbf{u}|x^T y| + \gamma_n^2 |x^T||y|.$$