Arithmétique flottante et analyse d'erreur (AFAE)

Lecture 1: summation

Theo Mary (CNRS)
theo.mary@lip6.fr
https://perso.lip6.fr/Theo.Mary/

M2 course at Sorbonne Université, 2025–2026

$$\sum_{i=1}^{n} x_i$$

Why summation is important

$$y = \sum_{i=1}^{n} x_i \qquad \text{. . . an ubiquitous and fundamental task!}$$

- **Dot products**:

$$a, b \in \mathbb{R}^n \Rightarrow a^T b = \sum_{i=1}^{n} a_i b_i$$

- **Matrix–vector products**:

$$A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^n \Rightarrow (Ab)_j = \sum_{i=1}^{n} a_{ji} b_i, \quad j = 1\colon m$$

- **Matrix–matrix products**:

$$A \in \mathbb{R}^{m \times n},\ B \in \mathbb{R}^{n \times p} \Rightarrow (AB)_{jk} = \sum_{i=1}^{n} a_{ji} b_{ik}, \quad j = 1\colon m, k = 1\colon p$$

- **Gaussian elimination (LU factorization)**:

$$A \in \mathbb{R}^{n \times n},\ A = LU \Rightarrow \begin{cases} \ell_{jk} & = \left(a_{jk} - \sum_{i=1}^{k-1} \ell_{ji} u_{ik} \right) / u_{kk} \\ u_{kj} & = a_{kj} - \sum_{i=1}^{k-1} \ell_{ki} u_{ij} \end{cases}$$

Accumulation of rounding errors

Summation suffers from the accumulation of rounding errors

Standard model of FP arithmetic:

$$\boxed{\mathrm{fl}(x \mathbin{\mathrm{op}} y) = (x \mathbin{\mathrm{op}} y)(1+\delta), \quad |\delta| \le u, \text{ for } \mathrm{op} \in \{+,-,\times,\div\}}$$

Consider the computation of $y = \sum_{i=1}^{n} x_i$ by recursive summation:

$$\begin{aligned}
y_2 &= x_1 + x_2 &\Rightarrow \quad \widehat{y}_2 &= (x_1+x_2)(1+\delta_1)\\
y_3 &= \widehat{y}_2 + x_3 &\Rightarrow \quad \widehat{y}_3 &= (\widehat{y}_2+x_3)(1+\delta_2)\\
& & &= (x_1+x_2)\underbrace{(1+\delta_1)(1+\delta_2)}_{\delta_1 \text{ and } \delta_2 \text{ accumulate!}} + x_3(1+\delta_2)\\
y_4 &= \ldots \text{etc.}
\end{aligned}$$

How can we measure the accumulated effect of all rounding errors?

Forward and backward errors

- Let $y = f(x)$ be computed in finite precision and let $\widehat{y}$ be the computed result
- **Forward error** analysis measures

$$|\widehat{y} - y| \text{ (absolute)} \quad \text{or} \quad \frac{|\widehat{y} - y|}{|y|} \text{ (relative)}$$

- **Backward error** analysis computes the smallest perturbation Δx such that

$$\widehat{y} = f(x + \Delta x)$$

  and measures $|\Delta x|$ (absolute) or $|\Delta x|/|x|$ (relative).
- Backward error analysis recasts the rounding errors as perturbations of the input data
- An algorithm is **backward stable** if it yields a small backward error, where "small" usually means $O(u)$

Forward and backward errors for summation

- **Forward error**

$$\eta_{\mathrm{fwd}} = \frac{|\widehat{y} - y|}{|y|}$$

- **Backward error**

$$\eta_{\mathrm{bwd}} = \min \left\{ \varepsilon > 0 : \exists \delta x_i,\ \widehat{y} = \sum_{i=1}^{n} x_i + \delta x_i,\ |\delta x_i| \le \varepsilon |x_i| \right\}.$$

Two questions:

- Find a **formula** for η_{bwd}
- Find **bounds** for η_{bwd} and η_{fwd} when $\widehat{y}$ is computed in floating-point arithmetic

Formula for backward error

$$\eta_{\mathrm{bwd}} = \min \left\{ \varepsilon > 0 : \exists \delta x_i ,\ \widehat{y} = \sum_{i=1}^{n} x_i + \delta x_i ,\ |\delta x_i| \leq \varepsilon |x_i| \right\}.$$

We have the formula

$$\eta_{\mathrm{bwd}} = \frac{|\widehat{y} - y|}{\sum_{i=1}^{n} |x_i|}.$$

Proof:

- $\frac{|\widehat{y}-y|}{\sum_{i=1}^{n} |x_i|} \leq \eta_{\mathrm{bwd}}$
- $\eta_{\mathrm{bwd}} \leq \frac{|\widehat{y}-y|}{\sum_{i=1}^{n} |x_i|}$ (using $\delta x_i = (\widehat{y} - y)\frac{|x_i|}{\sum_{i=1}^{n} |x_i|}$)

Formula for conditioning

As a result we also obtain the formula

$$\kappa = \frac{\eta_{\mathrm{fwd}}}{\eta_{\mathrm{bwd}}} = \frac{\sum_{i=1}^{n} |x_i|}{\left|\sum_{i=1}^{n} x_i\right|}.$$

- κ is large if $\sum |x_i| \gg |\sum x_i|$ $\Rightarrow$ **cancellation**

Backward error analysis

$$\begin{aligned}
& y_2 &&= x_1 + x_2 \\
\Rightarrow\quad & \widehat{y}_2 &&= (x_1 + x_2)(1+\delta_1) = x_1(1+\delta_1) + x_2(1+\delta_1) \\
& y_3 &&= \widehat{y}_2 + x_3 \\
\Rightarrow\quad & \widehat{y}_3 &&= (\widehat{y}_2 + x_3)(1+\delta_2) \\
& &&= x_1(1+\delta_1)(1+\delta_2) + x_2(1+\delta_1)(1+\delta_2) + x_3(1+\delta_2) \\
\ldots & && \\
\Rightarrow\quad & \widehat{y}_n &&= \textstyle\sum_{i=1}^{n} \left[x_i \prod_{k=k_i}^{n}(1+\delta_k)\right]
\end{aligned}$$

Backward error analysis

$$\begin{aligned}
& y_2 &&= x_1 + x_2 \\
\Rightarrow\quad & \widehat{y}_2 &&= (x_1 + x_2)(1+\delta_1) = x_1(1+\delta_1) + x_2(1+\delta_1) \\
& y_3 &&= \widehat{y}_2 + x_3 \\
\Rightarrow\quad & \widehat{y}_3 &&= (\widehat{y}_2 + x_3)(1+\delta_2) \\
& &&= x_1(1+\delta_1)(1+\delta_2) + x_2(1+\delta_1)(1+\delta_2) + x_3(1+\delta_2) \\
\ldots & && \\
\Rightarrow\quad & \widehat{y}_n &&= \textstyle\sum_{i=1}^{n} \left[x_i \prod_{k=k_i}^{n} (1+\delta_k) \right]
\end{aligned}$$

Worst-case fundamental lemma

Let δ_k, $k = 1 : n$, such that $|\delta_k| \leq u$ and $nu < 1$. Then

$$\prod_{k=1}^{n} (1+\delta_k) = 1 + \theta_n, \quad |\theta_n| \leq \gamma_n := \frac{nu}{1-nu}.$$

Proof: by induction.

General worst-case bound

General algorithm

```
$\mathbb{S} = \{x_1, \ldots, x_n\}$
Repeat
    Choose any pair $(x_i, x_j) \in \mathbb{S}^2$ $(i \neq j)$
    $\mathbb{S} \leftarrow \mathbb{S} \setminus \{x_i, x_j\}$
    $\mathbb{S} \leftarrow \mathbb{S} \cup \{x_i + x_j\}$
until $\mathbb{S} = \{y\}$
```

No matter the summation order we have the bound

$$\eta_{\mathrm{bwd}} \leq \gamma_{n-1} = (n-1)u + O(u^2)$$

Summary

Consider the computation

$$y = \sum_{i=1}^{n} x_i$$

In floating-point arithmetic, the forward error η_{fwd} is bounded by

$$\eta_{\text{fwd}} \leq \eta_{\text{bwd}}\kappa, \qquad \eta_{\text{bwd}} \leq \gamma_{n-1} = (n-1)u + O(u^2), \qquad \kappa = \frac{\sum |x_i|}{|\sum x_i|}$$

Thus η_{fwd} can be large when

- The unit roundoff u is large **(low precision)**
- The dimension n is large **(accumulation)**
- The condition number κ is large **(cancellation)**

No matter the summation order we have the bound

$$\eta_{\mathrm{bwd}} \leq \gamma_{n-1} = (n-1)u + O(u^2)$$

$\Rightarrow$ However, for specific orders, we can get much better bounds, and much smaller errors!

Summation tree

Given a summation order to compute $y = \sum_{i=1}^{n} x_i$, we define its associated summation tree as a **binary tree** such that:

- the n leaf nodes are the n summands x_i
- any inner node is equal to the sum of its two children
- the root node is the final sum y

Example: recursive summation is a **comb tree**

Tree-based summation

- For any summation tree, we have the bound:

$$\eta_{\mathrm{bwd}} \leq \gamma_h = hu + O(u^2)$$

  where h is the height of the tree
- The minimal bound is therefore attained for a **balanced binary tree**, for which $h = \lceil \log_2 n \rceil$. This is called pairwise summation.
- While it achieves the minimal bound, pairwise summation is not efficient on modern computers.

Blocked summation

Blocked summation algorithm:

```
for $i = 1\colon n/b$ do
    Compute $y_i = \sum_{j=(i-1)b+1}^{ib} x_j$.
end for
Compute $y = \sum_{i=1}^{n/b} y_i$.
```

$$\underbrace{\underbrace{x_1 \cdots x_b}_{y_1} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{x_{n-b+1} \cdots x_n}_{y_{n/b}}}_{y}$$

- Widely used in NLA libraries (BLAS, LAPACK, ...)
- $\eta_{\mathrm{bwd}} \le \gamma_h$ with $h = b + n/b - 2$
- With optimal $b = \sqrt{n}$: $h = 2(\sqrt{n} - 1)$

Blocked summation

```
for $i = 1: n/b$ do
    Compute $y_i = \sum_{j=(i-1)b+1}^{ib} x_j$.
end for
Compute $y = \sum_{i=1}^{n/b} y_i$.
```

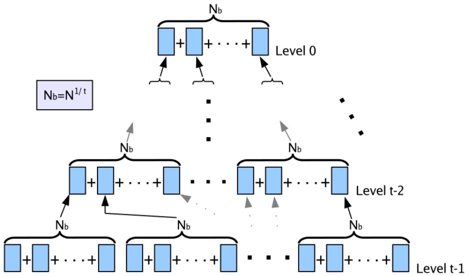
$$\widehat{y}_i = \sum_{j=(i-1)b+1}^{ib} \left[x_j \underbrace{\prod_{k=k_j}^{b} (1+\delta_k^{(i)})}_{\text{at most } b-1 \text{ terms}} \right]$$

$$\widehat{y} = \sum_{i=1}^{n/b} \left[\widehat{y}_i \underbrace{\prod_{k=k_i'}^{n/b} (1+\delta_k')}_{\text{at most } n/b-1 \text{ terms}} \right]$$

$$= \sum_{j=1}^{n} \left[x_j \underbrace{\prod_{k=k_j}^{b} (1+\delta_k^{(i)}) \prod_{k=k_j''}^{n/b} (1+\delta_k')}_{\text{at most } b+n/b-2 \text{ terms}} \right]$$

Superblock summation

- Superblocked summation: tree summation with t levels, block size at level t : $b_t = n^{1/t}$
  - $t = 1 \Rightarrow$ standard recursive summation
  - $t = 2 \Rightarrow$ optimal blocked summation
  - $t = \log_2 n \Rightarrow$ pairwise summation
  - $\eta_{\text{bwd}} \leq \gamma_h$ with $h = t(n^{1/t} - 1)$
  - Castaldo et al. (2009)



FABsum

Fast Accurate Blocked summation algorithm (FABsum) Blanchard, Higham, M. (2020)

```
for $i = 1 \colon n/b$ do
    Compute $y_i = \sum_{j=(i-1)b+1}^{ib} x_j$ with FastSum.
end for
Compute $y = \sum_{i=1}^{n/b} y_i$ with AccurateSum.
```

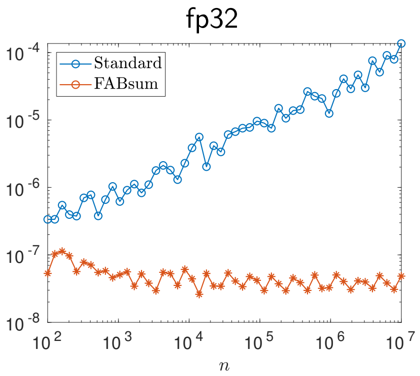
$$\underbrace{\underbrace{x_1 \cdots x_b}_{y_1} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{\cdots\cdots\cdots}_{\cdots} \quad \underbrace{x_{n-b+1} \cdots x_n}_{y_{n/b}}}_{y}$$

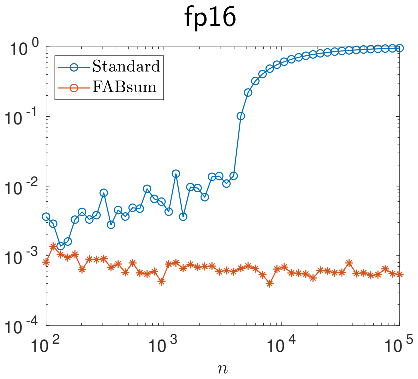
- Cost: $C(n,b) = \frac{n}{b} C_f(b) + C_a(\frac{n}{b}) \approx C_f(n) + \frac{1}{b} C_a(n)$
- Error: $\epsilon(n,b) = \epsilon_f(b) + \epsilon_a(n/b) + \epsilon_f(b)\epsilon_a(n/b)$

  $\Rightarrow$ If $\epsilon_a(p) = pu^2$ (recursive summation in precision u^2), then $\epsilon(n,b) = bu + O(u^2)$ is independent of n to first order

FABsum: numerical results

Backward error for summing random uniform $[0, 1]$ data





Blocked summation: implementation remark

```
for $i = 1: n/b$ do
    Compute $y_i = \sum_{j=(i-1)b+1}^{ib} x_j$.
end for
Compute $y = \sum_{i=1}^{n/b} y_i$.
```

- If implemented as is, requires storing n/b intermediate y_i values, which requires extra memory and is likely to slow down computation
- Better to implement as follows:

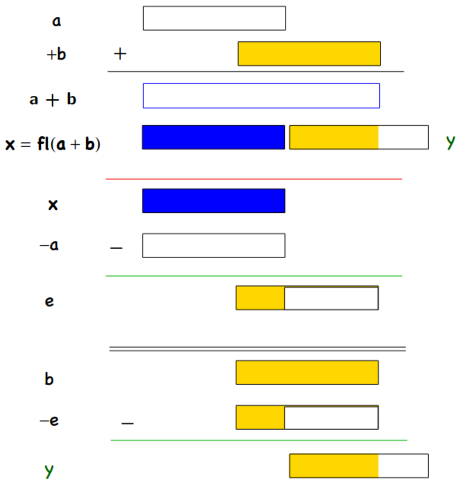
```
$y = 0$
for $i = 1: n/b$ do
    Compute $z = \sum_{j=(i-1)b+1}^{ib} x_j$.
    Compute $y = y + z$
end for
```

Fast2Sum

$$[x, y] = \texttt{Fast2Sum}(a, b)$$

Input: $a, b \in \mathbb{F}$ such that $|a| \geq |b|$
Output: $x = \mathrm{fl}(a + b), y \in \mathbb{F}$ such that
$x + y = a + b$

$x = a + b$
$e = x - a$
$y = b - e$



Fast2Sum

$$[x, y] = \mathtt{2Sum}(a, b)$$

Input: $a, b \in \mathbb{F}$ ~~such that $|a| \geq |b|$~~

Output: $x = \mathrm{fl}(a+b), y \in \mathbb{F}$ such that

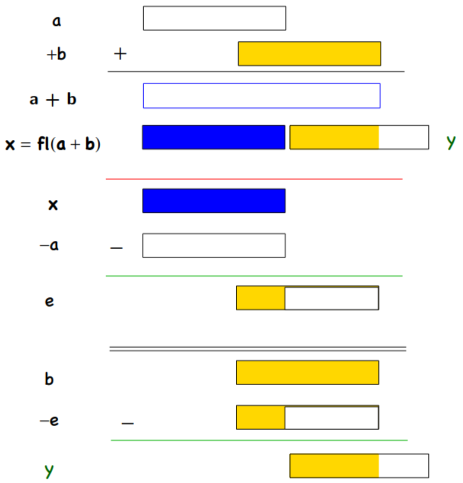
$x + y = a + b$

$x = a + b$

$e = x - a$

$y = (a - (x - e)) + (b - e)$

Can remove $|a| \geq |b|$ restriction at the cost of 3 extra flops



Kahan's summation (compensated summation)

```
Input: $x_i \in \mathbb{F}$, $i = 1\colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

$y = 0$, $z = 0$
for $i = 1\colon n$ do
    $t = x_i + z$
    $[y, z] = \texttt{Fast2Sum}(y, t)$
end for
```

- Kahan's summation reinjects the errors at each step in the sum
- It satisfies the bound $\eta_{\mathrm{bwd}} \leq 2u + O(nu^2)$ (proof is quite complicated)

Kahan–Babuška's variant

```
Input: $x_i \in \mathbb{F}$, $i = 1\colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

$y = 0$, $e = 0$
for $i = 1\colon n$ do
    $[y, z] = \texttt{Fast2Sum}(y, x_i)$
    $e = e + z$
end for
$y = y + e$
```

- Kahan–Babuška's variant accumulates the errors separately and adds them to the sum at the end
- It satisfies the bound $\eta_{\mathrm{bwd}} \le 2u + n^2 u^2$ (proof is still quite complicated)

Kahan–Babuška's variant with branching: Neumaier's variant

```
Input: $x_i \in \mathbb{F}$, $i = 1\colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

$y = 0$, $e = 0$
for $i = 1\colon n$ do
    if $|y| \geq |x_i|$ then
        $[y, z] = \texttt{Fast2Sum}(y, x_i)$
    else
        $[y, z] = \texttt{Fast2Sum}(x_i, y)$
    end if
    $e = e + z$
end for
$y = y + e$
```

- Remember that $\texttt{Fast2Sum}(a, b)$ is only exact if $|a| \geq |b|$
- It satisfies the bound $\eta_{\mathrm{fwd}} \leq u + n^2 \kappa u^2$

Kahan–Babuška's variant with branching: Neumaier's variant

```
Input: $x_i \in \mathbb{F}$, $i = 1 \colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

$y = 0$, $e = 0$
for $i = 1 \colon n$ do
    if $|y| \geq |x_i|$ then
        $[y, z] = \texttt{Fast2Sum}(y, x_i)$
    else
        $[y, z] = \texttt{Fast2Sum}(x_i, y)$
    end if
    $e = e + z$
end for
$y = y + e$
```

Proof: thanks to the branching, `Fast2Sum` is now error-free so before the final $y + e$ addition, we have

$$\sum_{i=1}^{n} x_i = y + \sum_{i=1}^{n} z_i$$

and after it we thus obtain

$$\widehat{y} = y + \widehat{e} + \delta, \quad |\delta| \leq u|y + \widehat{e}|$$

$$\widehat{e} = \sum_{i=1}^{n} z_i + \Delta e, \quad |\Delta e| \leq \gamma_{n-1} \sum_{i=1}^{n} |z_i|$$

$$\leq \gamma_{n-1}^2 \sum_{i=1}^{n} |x_i|$$

- Remember that `Fast2Sum`(a, b) is only exact if $|a| \geq |b|$
- It satisfies the bound $\eta_{\mathrm{fwd}} \leq u + O(n^2)\kappa u^2$

Avoiding branching: `Sum2`

```
**Input:** $x_i \in \mathbb{F}$, $i = 1\colon n$
**Output:** $y \approx \sum_{i=1}^{n} x_i$

$y = 0$, $e = 0$
**for** $i = 1\colon n$ **do**
    $[y, z] = \texttt{2Sum}(y, x_i)$
    $e = e + z$
**end for**
$y = y + e$
```

- Replacing `Fast2Sum` by `2Sum` avoids branching but requires more flops

Sum2: variant with vector overwriting

```
Input: $x_i \in \mathbb{F}$, $i = 1\colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

for $i = 2\colon n$ do
    $[x_i, x_{i-1}] = \texttt{2Sum}(x_i, x_{i-1})$
end for
$y = \left(\sum_{i=1}^{n-1} x_i\right) + x_n$
```

- Let x' be the overwritten vector (for clarity of notation)
- $x'_1, \dots, x'_{n-1}$ are overwritten by the errors and x'_n by the floating-point evaluation of the sum $\sum_{i=1}^{n} x_i$
- Hence $|x'_i| \le O(n)u\sum_{i=1}^{n}|x_i|$ and $|x'_n| \le |\sum_{i=1}^{n} x_i| + O(n)u\sum_{i=1}^{n}|x_i|$
- Therefore the condition number of x' has been reduced by a factor $O(n^2)u$:

$$\frac{\sum_{i=1}^{n}|x'_i|}{|\sum_{i=1}^{n} x'_i|} = \frac{\sum_{i=1}^{n}|x'_i|}{|\sum_{i=1}^{n} x_i|} \le \frac{O(n^2)u\sum_{i=1}^{n}|x_i| + |\sum_{i=1}^{n} x_i|}{|\sum_{i=1}^{n} x_i|} = O(n^2)u\kappa + 1$$

`SumK`: K-fold compensated summation

```
Input: $x_i \in \mathbb{F}$, $i = 1\colon n$
Output: $y \approx \sum_{i=1}^{n} x_i$

for $k = 1\colon K-1$ do
    for $i = 2\colon n$ do
        $[x_i, x_{i-1}] = \mathtt{2Sum}(x_i, x_{i-1})$
    end for
end for
$y = \left(\sum_{i=1}^{n-1} x_i\right) + x_n$
```

- After K iterations, the condition number of x' is $O((nu)^K)\kappa$
- Hence we have the bound $\eta_{\mathrm{fwd}} \le u + O(n^k)\kappa u^k$

  Ogita, Rump, Oishi (2005)
- However, we do not know κ, so how do we know when to stop? $\Rightarrow$ `AccSum`

  Rump, Ogita, Oishi (2008)

Distillation methods

$$\sum_{i=1}^{n} x_i \xrightarrow[\textit{distillation}]{} \sum_{i=1}^{n} d_i, \qquad \text{where } \kappa(d_i) \ll \kappa(x_i)$$

- Goal: distill until $\sum_i d_i$ can be accurately evaluated in floating-point arithmetic
- Higher $\kappa \Rightarrow$ more iterations!

Condensation methods

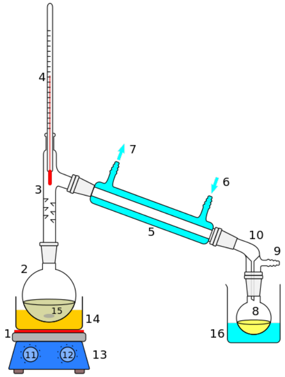
$$\sum_{i=1}^{n} x_i \xrightarrow[\text{condensation}]{} \sum_{i=1}^{m} c_i \xrightarrow[\text{distillation}]{} \sum_{i=1}^{m} d_i, \qquad \text{where } m \ll n \text{ and } \kappa(d_i) \ll \kappa(x_i)$$

- Goal: reduce the number of summands before applying the costly distillation

Condensation methods

$$\sum_{i=1}^{n} x_i \xrightarrow[\text{condensation}]{} \sum_{i=1}^{m} c_i \xrightarrow[\text{distillation}]{} \sum_{i=1}^{m} d_i, \qquad \text{where } m \ll n \text{ and } \kappa(d_i) \ll \kappa(x_i)$$

- Goal: reduce the number of summands before applying the costly distillation



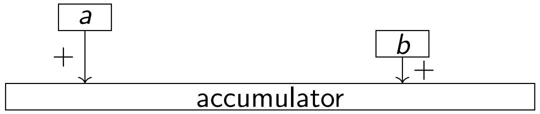
Condense & Distill, conceptually

Conceptual algorithm

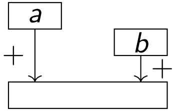
```
$\mathbb{S} = \{x_1, \ldots, x_n\}$
Repeat for all pairs $(x_i, x_j) \in \mathbb{S}^2$ $(i \neq j)$ such that $x_i + x_j$ is exact
    $\mathbb{S} \leftarrow \mathbb{S} \setminus \{x_i, x_j\}$
    $\mathbb{S} \leftarrow \mathbb{S} \cup \{x_i + x_j\}$
until no such pair remains
Distill $\mathbb{S}$
```

- Can we easily determine when $x_i + x_j$ is exact?
- Can we bound the maximum number of leftover summands?

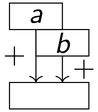
Demmel–Hida method



Demmel–Hida method

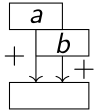


Demmel–Hida method



Consider arithmetic with f-bit mantissa and e-bit exponent ($e = 11$ for fp64).

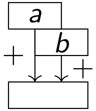
Demmel–Hida method



Consider arithmetic with f-bit mantissa and e-bit exponent ($e = 11$ for fp64).

- One big accumulator: Kulisch method
  ... need one accumulator of $2^e + \log_2 n$ bits

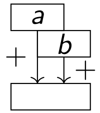
Demmel–Hida method



Consider arithmetic with f-bit mantissa and e-bit exponent ($e = 11$ for fp64).

- One big accumulator: Kulisch method . . . need one accumulator of $2^e + \log_2 n$ bits
- One accumulator per exponent: Malcolm method . . . need 2^e accumulators of $f + \log_2 n$ bits

Demmel–Hida method



Consider arithmetic with f-bit mantissa and e-bit exponent ($e = 11$ for fp64).

- One big accumulator: Kulisch method ... need one accumulator of $2^e + \log_2 n$ bits
- One accumulator per exponent: Malcolm method ... need 2^e accumulators of $f + \log_2 n$ bits
- Demmel–Hida: general method, balance the number and size of accumulators.

Input: n summands x_i, number of exponent bits m to extract
Output: $y = \sum_{j=1}^{2^m} A_j$

```
Initialize $A_j = 0$ for $j = 1, \dots, 2^m$
for $i = 1 : n$ do
    $j \leftarrow m$ leading bits of exponent($x_i$)
    $A_j \leftarrow A_j + x_i$
end for
```

With 2^m accumulators, need F-bit mantissa with

$$F \geq f + \lceil \log_2 n \rceil + 2^{e-m} - 1$$

Demmel–Hida method

		number of bits			
		signif. (t)	exp.	range	$u = 2^{-t}$
fp128	quadruple	113	15	$10^{\pm 4932}$	1×10^{-34}
fp64	double	53	11	$10^{\pm 308}$	1×10^{-16}

Numerical example with fp64 and fp128 arithmetics:

- Assume $\log_2 n \le 29$ ($n \lesssim 0.5 \times 10^9$)
- $F = 113$, $f = 53$, $e = 11 \Rightarrow m$ must thus satisfy

$$F \ge f + \lceil \log_2 n \rceil + 2^{e-m} - 1$$
$$\Rightarrow 2^{11-m} \le 32$$
$$\Rightarrow 6 \le m$$

Distillation vs condensation

Distillation methods (AccSum, etc.)

- ☺ Entirely in the working precision
- ☺ Only uses standard arithmetic operations
- ☹ Strongly dependent on the conditioning
- ☹ Limited parallelism

Condensation methods (Demmel–Hida, etc.)

- ☺ Independent on the conditioning
- ☺ High level of parallelism
- ☹ Requires access to the exponent
- ☹ Requires extended precision arithmetic

Distillation vs condensation

Distillation methods (AccSum, etc.)

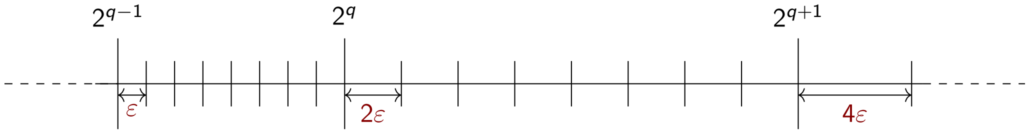
- ☺ Entirely in the working precision
- ☺ Only uses standard arithmetic operations
- ☹ Strongly dependent on the conditioning
- ☹ Limited parallelism

Condensation methods (Demmel–Hida, etc.)

- ☺ Independent on the conditioning
- ☺ High level of parallelism
- ☹ Requires access to the exponent
- ☹ Requires extended precision arithmetic

Can we avoid the use of extended precision arithmetic?

When is $x + y$ exact? Intuition 1



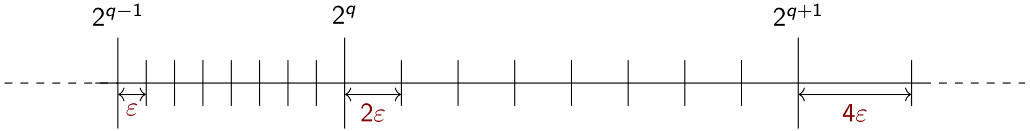
Let $x, y \in \mathbb{F} \cap [2^{q-1}, 2^q]$ such that

$$x = 2^{q-1} + k_x\varepsilon$$
$$y = 2^{q-1} + k_y\varepsilon$$

Then

$$\begin{aligned} x + y &= 2^{q-1} + k_x\varepsilon + 2^{q-1} + k_y\varepsilon \\ &= 2^q + (k_x + k_y)\varepsilon \in \mathbb{F} \text{ iff } k_x + k_y \equiv 0 \bmod 2 \end{aligned}$$

When is $x + y$ exact? Intuition 1



Similarly if

$$x = 2^{q-1} + k_x \varepsilon$$
$$y = 2^q + k_y 2\varepsilon$$

then $x + y \in \mathbb{F}$ iff

$$\begin{cases} x + y \leq 2^{q+1} \text{ and } k_x \equiv 0 \bmod 2 \\ x + y > 2^{q+1} \text{ and } k_x + 2k_y \equiv 0 \bmod 4 \end{cases}$$

When is $x + y$ exact? Intuition 2

$$2^q \times 101 + 2^q \times 111 = 2^q \times 1100 = 2^{q+1} \times 110.0 \in \mathbb{F}$$
$$2^q \times 101 + 2^q \times 110 = 2^q \times 1011 = 2^{q+1} \times 101.1 \notin \mathbb{F}$$

$$2^q \times 101 + 2^{q-1} \times 111 = 2^{q+1} \times 100.01 \notin \mathbb{F}$$
$$2^q \times 101 + 2^{q-1} \times 110 = 2^{q+1} \times 100.00 \in \mathbb{F}$$

When is $x + y$ exact? Theorem

Theorem (Graillat and M.)

Let $x, y \in \mathbb{F}$ of the same sign $\sigma = \pm 1$ such that

$$x = \sigma(\beta^{e_x} + k_x \varepsilon_{e_x}),$$
$$y = \sigma(\beta^{e_y} + k_y \varepsilon_{e_y}).$$

Assuming (without loss of generality) that $|x| \le |y|$, then $x + y \in \mathbb{F}$, and thus the addition is exact, iff one of the following conditions is met:

(i) $x = 0$;

(ii) $|x + y| < \beta^{e_y + 1}$, $e_y - e_x \le t - 1$, and $k_x \equiv 0 \bmod \beta^{e_y - e_x}$;

(iii) $|x + y| = \beta^{e_y + 1}$, $e_y + 1 \le e_{\max}$, $e_y - e_x \le t - 1$, and $k_x \equiv 0 \bmod \beta^{e_y - e_x}$;

(iv) $|x + y| > \beta^{e_y + 1}$, $e_y + 1 \le e_{\max}$, $e_y - e_x \le t - 2$, and $k_x + k_y \beta^{e_y - e_x} \equiv 0 \bmod \beta^{e_y - e_x + 1}$.

When is $x + y$ exact? Corollary

$$k_x + k_y \beta^{e_y - e_x} \equiv 0 \bmod \beta^{e_y - e_x + 1} \quad \xrightarrow[\beta=2,\ e_x=e_y]{} \quad k_x + k_y \equiv 0 \bmod 2$$

Corollary

If $x, y \in \mathbb{F}$ with $\beta = 2$ have the same sign, exponent, and least significant bit, then barring overflow their addition is exact.

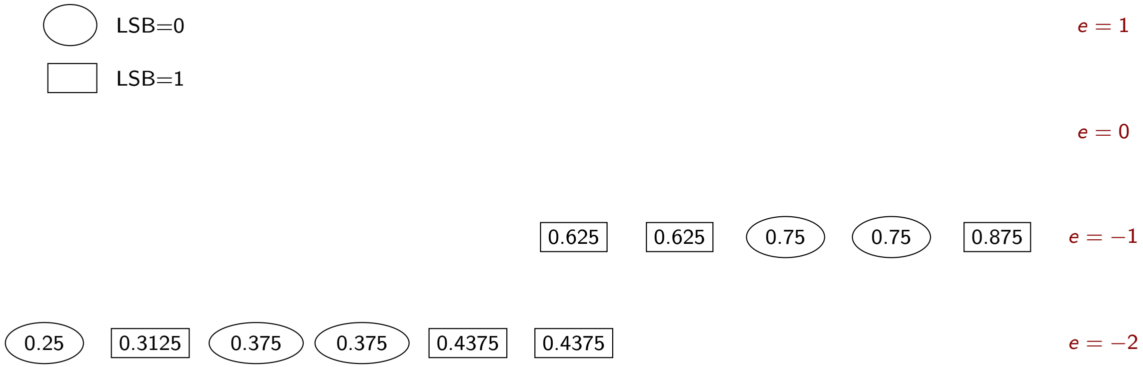
Graillat–Mary method

Consider the toy example

$$y = 0.25 + 0.3125 + 0.375 + 0.375 + 0.4375 + 0.4375 + 0.625 + 0.625 + 0.75 + 0.75 + 0.875$$

computed with 3-bit arithmetic:

$$\mathbb{F} = \{0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3\}$$



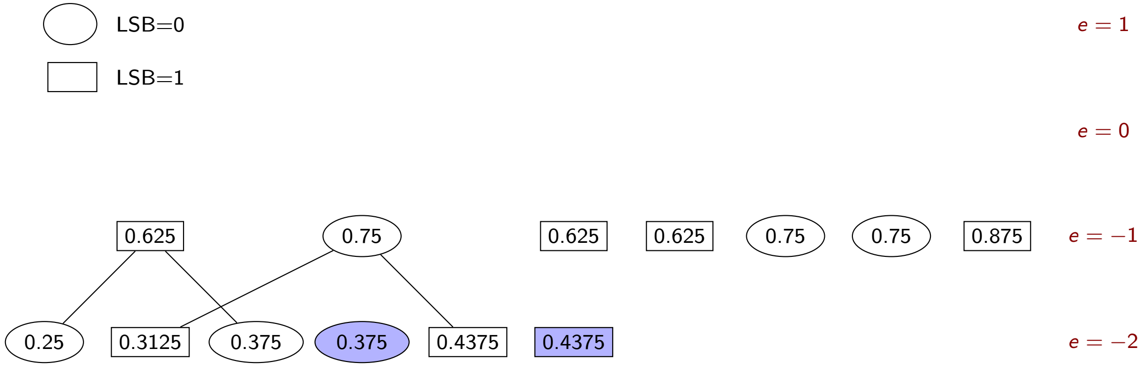
Graillat–Mary method

Consider the toy example

$$y = 0.25 + 0.3125 + 0.375 + 0.375 + 0.4375 + 0.4375 + 0.625 + 0.625 + 0.75 + 0.75 + 0.875$$

computed with 3-bit arithmetic:

$$\mathbb{F} = \{0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3\}$$



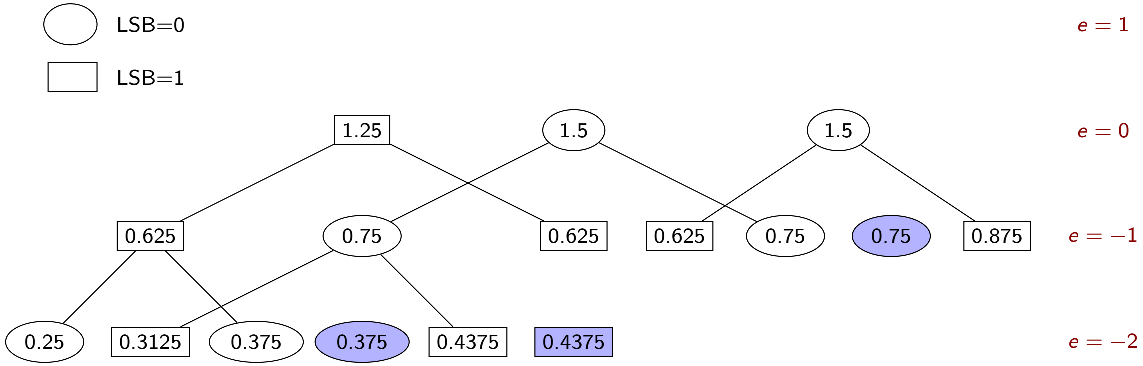
Graillat–Mary method

Consider the toy example

$$y = 0.25 + 0.3125 + 0.375 + 0.375 + 0.4375 + 0.4375 + 0.625 + 0.625 + 0.75 + 0.75 + 0.875$$

computed with 3-bit arithmetic:

$$\mathbb{F} = \{0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3\}$$



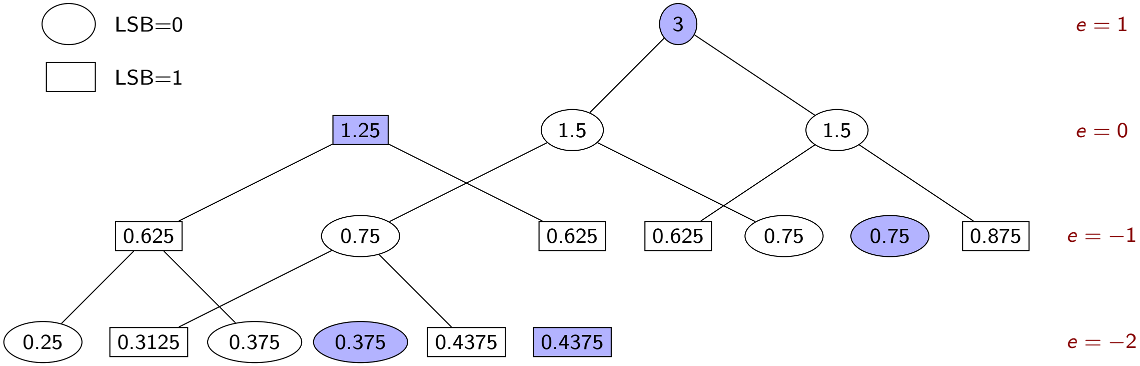
Graillat–Mary method

Consider the toy example

$$y = 0.25 + 0.3125 + 0.375 + 0.375 + 0.4375 + 0.4375 + 0.625 + 0.625 + 0.75 + 0.75 + 0.875$$

computed with 3-bit arithmetic:

$$\mathbb{F} = \{0.25, 0.3125, 0.375, 0.4375, 0.5, 0.625, 0.75, 0.875, 1, 1.25, 1.5, 1.75, 2, 2.5, 3\}$$



y = 0.375 + 0.4375 + 0.75 + 1.25 + 3

Graillat–Mary method

Input: n summands x_i and a distillation method `distill`
Output: $y = \sum_{i=1}^{n} x_i$

```
Initialize $\mathrm{Acc}(e, s, b)$ to 0 for $e = e_{\min} : e_{\max}$, $s \in \{-1, 1\}$, $b \in \{0, 1\}$.
for all $x_i$ in any order do
    $e = \mathtt{exponent}(x_i)$
    $s = \mathtt{sign}(x_i)$
    $b = \mathrm{LSB}(x_i)$
    insert$(\mathrm{Acc}, x_i, e, s, b)$
end for
$x_{\mathrm{condensed}} =$ gather$(\mathrm{Acc})$
$y = \mathtt{distill}(x_{\mathrm{condensed}})$
```

```
function insert$(\mathrm{Acc}, x, e, s, b)$
if $\mathrm{Acc}(e, s, b) = 0$ then
    $\mathrm{Acc}(e, s, b) = x$
else
    $x' = \mathrm{Acc}(e, s, b) + x$
    $\mathrm{Acc}(e, s, b) = 0$
    $b' = \mathrm{LSB}(x')$
    insert$(\mathrm{Acc}, x', e + 1, s, b')$
end if
end function
```

```
function $x_{\mathrm{condensed}} =$ gather$(\mathrm{Acc})$
$i = 0$
for all nonzero $\mathrm{Acc}(e, s, b)$ do
    $i = i + 1$
    $x_{\mathrm{condensed}}(i) = \mathrm{Acc}(e, s, b)$
end for
end function
```

Graillat–Mary method

Conceptual algorithm

```
$\mathbb{S} = \{x_1, \ldots, x_n\}$
Repeat for all pairs $(x_i, x_j) \in \mathbb{S}^2$ $(i \neq j)$ such that $x_i + x_j$ is exact
    $\mathbb{S} \leftarrow \mathbb{S} \setminus \{x_i, x_j\}$
    $\mathbb{S} \leftarrow \mathbb{S} \cup \{x_i + x_j\}$
until no such pair remains
Distill $\mathbb{S}$
```

- Can we easily determine when $x_i + x_j$ is exact? YES! It suffices to check the sign, exponent, and LSB of x_i and x_j
- Can we bound the maximum number of leftover summands? YES! At most $4L$ summands where L is the depth of the tree

$$L \leq \lceil \log_2 n \rceil + d$$

where d is independent of n and depends on the range of the values (at most 2047 in binary64)

Distillation vs condensation

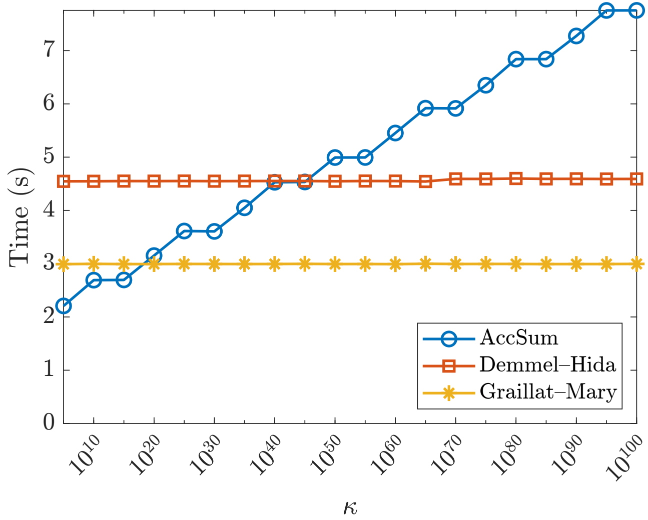
Distillation methods (AccSum, etc.)

- ☺ Entirely in the working precision
- ☺ Only uses standard arithmetic operations
- ☹ Strongly dependent on the conditioning
- ☹ Limited parallelism

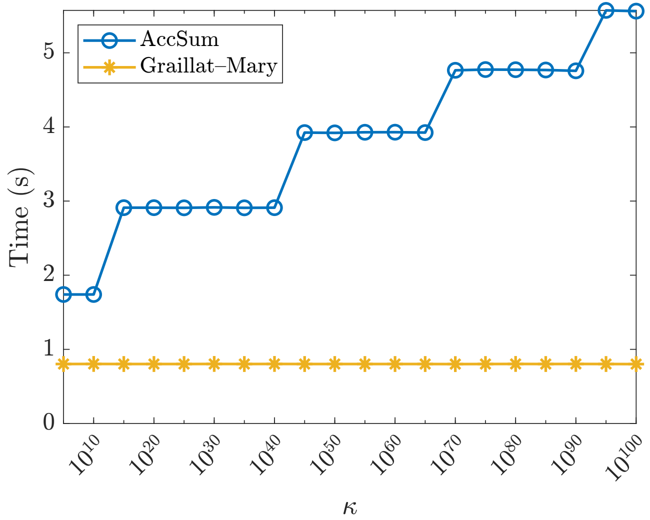
Condensation methods (Demmel–Hida, Graillat–Mary)

- ☺ Independent on the conditioning
- ☺ High level of parallelism
- ☹ Requires access to the exponent + LSB
- ~~☹ Requires extended precision arithmetic~~

Performance comparison



Quadruple working precision



Adaptive precision algorithms

- Given an algorithm and a prescribed accuracy ε, adaptively select the minimal precision for each instruction depending on the data

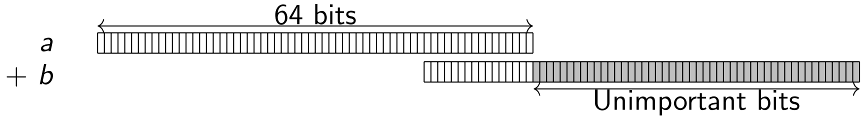
$\Rightarrow$ **First of all, why should the precisions vary?**

Adaptive precision algorithms

- Given an algorithm and a prescribed accuracy ε, adaptively select the minimal precision for each instruction depending on the data

$\Rightarrow$ **First of all, why should the precisions vary?**

- Because not all computations are equally "important"!
  Example:



$\Rightarrow$ **Opportunity for mixed precision:** adapt the precisions to the data at hand by storing and computing "less important" (which usually means smaller) data in lower precision

Sparse matrix–vector product (SpMV)

Goal: compute $y = Ax$, where A is a sparse matrix, with a prescribed accuracy ε

```
for $i = 1\colon m$ do
    $y_i = \sum_{j \in nnz_i(A)} a_{ij} x_j$
end for
```

If computed in precision ε, $\widehat{y}$ satisfies

$$|\widehat{y}_i - y_i| \le n_i \varepsilon \sum_{j \in nnz_i(A)} |a_{ij} x_j|$$

and thus

$$\|\widehat{y} - y\| \le c\varepsilon \|A\| \|x\| \qquad (c = \max_i n_i)$$

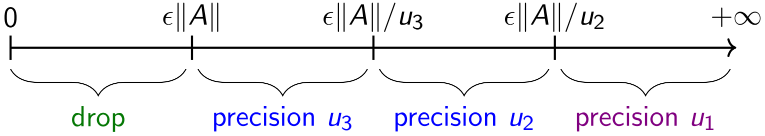
This is a normwise backward error bound: $\widehat{y} = (A + E)x$, $\|E\| \le c\varepsilon\|A\|$.

Adaptive precision SpMV

- Given p available precisions $u_1 < \varepsilon < u_2 < \ldots < u_p$, define partition $A = \sum_{k=1}^{p} A^{(k)}$ where

$$a_{ij}^{(k)} = \begin{cases} \mathrm{fl}_k(a_{ij}) & \text{if } |a_{ij}| \in (\varepsilon\|A\|/u_k, \varepsilon\|A\|/u_{k+1}] \\ 0 & \text{otherwise} \end{cases}$$

$\Rightarrow$ the precision of each element is chosen inversely proportional to its magnitude



$$\begin{pmatrix} \times & & \times \\ \times & \times & \\ & \times & \times \end{pmatrix} = \begin{pmatrix} d & & \\ & d & \\ & & d \end{pmatrix} + \begin{pmatrix} & & s \\ & & \\ & s & \end{pmatrix} + \begin{pmatrix} & & \\ h & & \\ & & \end{pmatrix}$$

Adaptive precision SpMV

```
for $i = 1: m$ do
    for $k = 1: p$ do
        $y_i^{(k)} = \sum_{j \in nnz_i(A^{(k)})} a_{ij}^{(k)} x_j$ in precision $u_k$
    end for
    $y_i = \sum_{k=1}^{p} y_i^{(k)}$ in precision $u_1$
end for
```

- Compute $y^{(k)} = A^{(k)} x$ in precision u_k. The computed $\widehat{y}^{(k)}$ satisfies

$$|\widehat{y}_i^{(k)} - y_i^{(k)}| \le (n_i^{(k)})^2 \varepsilon \|A\| \|x\|$$

- Compute $y = \sum_{k=1}^{p} y^{(k)}$ in precision u_1. The computed $\widehat{y}$ satisfies

$$\begin{aligned} \widehat{y}_i &= \sum_{k=1}^{p} \widehat{y}_i^{(k)} + e_i, \quad |e_i| \le p u_1 \|A\| \|x\| \\ &= y_i + f_i, \quad |f_i| \le c\varepsilon \|A\| \|x\| \end{aligned}$$

Graillat, Jézéquel, M., Molina (2024)

Custom precision formats

The more precisions we have, the more we can reduce storage $\Rightarrow$ can we exploit custom precision formats?

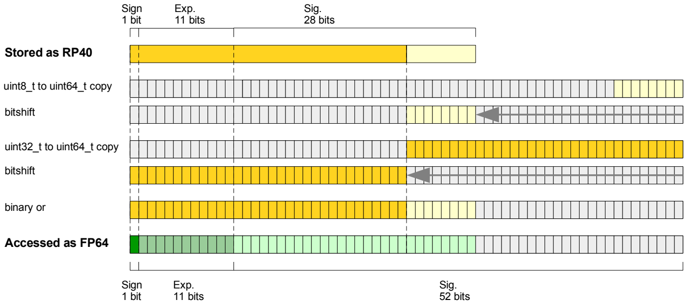
Emulated formats

	Bits			
Format	Signif. (t)	Exponent	Range	$u = 2^{-t}$
bf16	8	8	$10^{\pm 38}$	4×10^{-3}
fp24	16	8	$10^{\pm 38}$	2×10^{-5}
fp32	24	8	$10^{\pm 38}$	6×10^{-8}
fp40	29	11	$10^{\pm 308}$	2×10^{-9}
fp48	37	11	$10^{\pm 308}$	8×10^{-12}
fp56	45	11	$10^{\pm 308}$	3×10^{-14}
fp64	53	11	$10^{\pm 308}$	1×10^{-16}

How to efficiently implement custom precision storage?

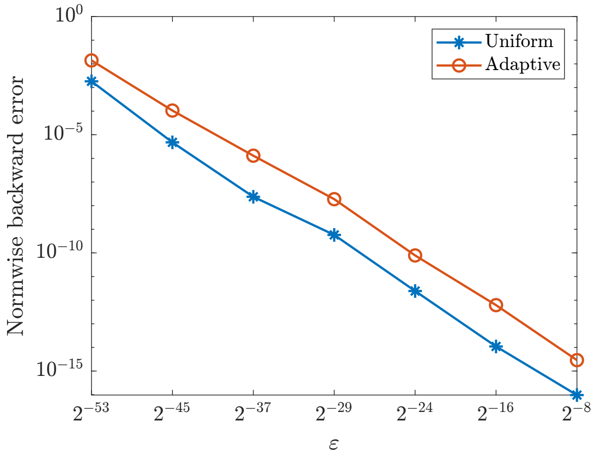
Custom precision accessor

```
union union64 {
uint64_t i;
double f;
};
double RpToFp (rp40 rp, size_t i){
union union64 u64;
uint64_t i64h, i64l;
i64h = (uint64_t)rp.i32[i];
i64h = i64h << 32;
i64l = (uint64_t)rp.i8[i];
i64l = i64l << 24;
u64.i = i64h | i64l;
return u64.f;
}
```



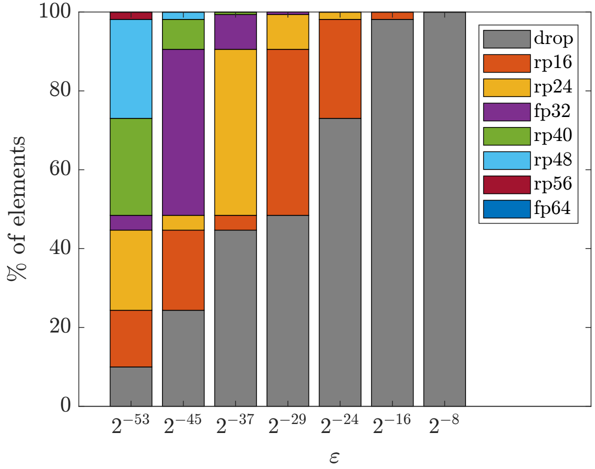
Graillat, Jézéquel, M., Molina, Mukunoki (2024)

Experimental results (Long_Coup_dt6 matrix, $n \approx 1.5\text{M}$)



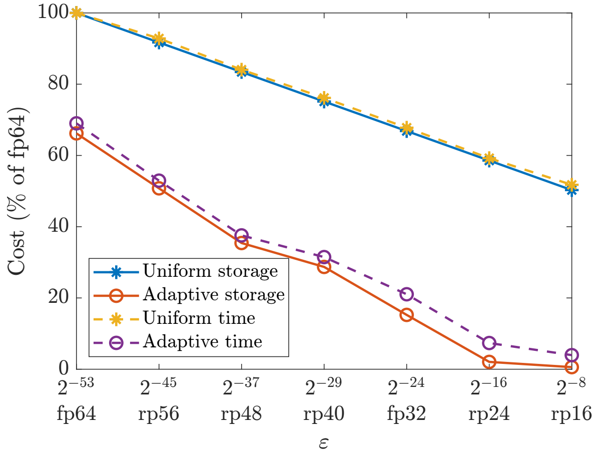
- Controlled accuracy

Experimental results (Long_Coup_dt6 matrix, $n \approx 1.5$M)



- Controlled accuracy

Experimental results (Long_Coup_dt6 matrix, $n \approx 1.5\text{M}$)



- Controlled accuracy
- Storage reduced by at least 30% and potentially much more for larger ε.
- Time cost matches storage.

Conclusion

$$\eta_{\mathrm{fwd}} \le \eta_{\mathrm{bwd}}\kappa, \qquad \eta_{\mathrm{bwd}} \le \gamma_{n-1} = (n-1)u + O(u^2), \qquad \kappa = \frac{\sum |x_i|}{|\sum x_i|}$$

- We have seen various summation methods with different properties/objectives: handling error accumulation, cancellation, using mixed precision...
- A common theme has been the reordering of the summands by grouping them into blocks/buckets,
  - either fixed-size groups of arbitrary summands
  - or groups of summands of similar magnitude.
- We have seen several possible uses of mixed precision arithmetic:
  - Mixed precision blocked summation (FABsum): reduce accumulation $\Rightarrow \eta_{\mathrm{bwd}}$ independent of n
  - Bucket summation with extended precision (Demmel-Hida): reduce cancellation $\Rightarrow \eta_{\mathrm{fwd}}$ independent of κ
  - Bucket summation with adaptive precision: exploit lower precisions while controlling η_{bwd}