浮動小数点演算のerror-free変換とその応用

荻田 武史
東京女子大学 文理学部 数理学科

joint work with
大石 進一, Siegfried M. Rump

目次

1. 概要

2. 浮動小数点演算のerror-free変換

3. 任意計算精度の総和・内積計算

4. 任意結果精度の総和・内積計算

5. 結論

概要

一般的に，浮動小数点演算によって得られた結果は，丸め誤差の影響で，しばしば信頼性がなくなることはよく知られている．

目的 ベクトルの総和 $\sum_{i=1}^{n} x_i$ や内積 $x^T y = \sum_{i=1}^{n} x_i\, y_i$ を浮動小数点演算のみで高精度に求める．

$\Longrightarrow$ 科学技術計算の基本のひとつ

応用 悪条件な連立一次方程式，固有値問題，特異値問題等

$$
\begin{aligned}
&\textbf{function} \quad s_n = \textbf{DSum}(p, n) \\
&s_0 = 0; \\
&for \quad i = 1 : n \\
&\qquad s_i = s_{i-1} \oplus p_i; \quad \% \oplus\text{: fl-pt add} \\
&end
\end{aligned}
$$

このとき，丸め誤差解析から，s_n の相対誤差は

$$
\left| \frac{\sum p_i - s_n}{\sum p_i} \right| \leq \mathcal{O}(\mathbf{u}) \mathrm{cond}(\sum p_i)
$$

ただし，$\mathbf{u}$ は計算精度，$\mathrm{cond}(\sum p_i)$ は問題の難しさを表す条件数.

背景

ベクトルの総和・内積の高精度なアルゴリズムはたくさんある(次頁)が，大別すると以下のようなタイプに分類できる．

a) 経験的に結果の精度が改善される方式
b) 内部演算が多倍長演算のように高精度になる方式 (precision)
c) 指定した精度の結果が得られる方式 (accuracy)

多くのアルゴリズムは b) に属する．

ベクトルの総和・内積計算についての歴史（抜粋）

1965 Møller (BIT)
1970 Nickel (ZAMM) (Kahan-Babuška's algorithm)
1971 Malcolm (Comm. ACM)
1972 Pichat (Numer. Math.)
1973 Kiełbasziński (Math. Stos.)
1974 Neumaier (ZAMM) (improved Kahan-Babuška's algorithm)
1977 Bohlender (IEEE Trans. Comput.)
1986 Kulisch, Miranker (SIAM Review, originally 1970's in Karlsruhe)
1991 Priest (Proc. IEEE Symposium)
1999 Anderson (SIAM J. Sci. Comput.)
2002 Li et al. (ACM Trans. Math. Softw., XBLAS)
2003 Demmel, Hida (SIAM J. Sci. Comput)
2005 Ogita, Rump, Oishi (SIAM J. Sci. Comput.)
2008 Rump, Ogita, Oishi (SIAM J. Sci. Comput.)

precision（演算精度）とaccuracy（結果精度）

b)に属するアルゴリズムで得られた結果を $\mathsf{res} \in \mathbb{F}$ とすると

$$\frac{|\mathsf{res} - \sum p_i|}{|\sum p_i|} \leq \mathbf{u}_{\mathrm{out}} + \mathbf{u}_{\mathrm{int}} \mathrm{cond}(\sum p_i)$$

但し，$\mathbf{u}_{\mathrm{int}}$ は内部計算精度，$\mathbf{u}_{\mathrm{out}}$ は出力フォーマットの精度，

$$\mathrm{cond}(\sum p_i) := \frac{\sum |p_i|}{|\sum p_i|} \quad \text{(ベクトルの総和の条件数)}$$

$\Longrightarrow$ $\mathbf{u}_{\mathrm{int}}$ precision (多倍長精度に対応．結果精度の保証はない)

precisionとaccuracy (2)

c) に属するアルゴリズムで得られた結果を $\mathbf{res} \in \mathbb{F}$ とすると

$$\frac{|\mathbf{res} - \sum p_i|}{|\sum p_i|} \leq \mathbf{u}_{\mathrm{tol}}$$

但し，$\mathbf{u}_{\mathrm{tol}}$ は指定した許容誤差

$\Longrightarrow$ $\mathbf{u}_{\mathrm{tol}}$ accuracy (条件数に依存しない．結果精度が保証されている)

計算速度の点で従来方式の不利な点

- 入力データのソートが必要

  i) 絶対値の大きい(小さい)順 ($\mathcal{O}(n \log n)$ operations)
  ii) 指数部の大きい（小さい）順 ($\mathcal{O}(n)$ operations)

- 内部ループが分岐を含む (less compiler optimization)

- 通常使用する精度(単精度・倍精度)の他に，より高精度なフォーマット(拡張倍精度など)が必要 (less portability)

- 仮数部や指数部へのアクセスが必要 (less portability)

本研究の方針

- 通常 (単精度・倍精度) の浮動小数点演算による和・差・積のみを用いる

- 内部ループに分岐や仮数部・指数部へのアクセス等がない

$\Longrightarrow$ 特別な高精度フォーマットや特別なハードウェアは必要ない
$\Longrightarrow$ これらの特徴から，提案手法は計算量の意味だけでなく実行時間の意味でも大変高速になる

Error-free変換

$\mathbb{F}$: 倍精度浮動小数点数の集合, $\mathrm{fl}(\cdot)$: 倍精度浮動小数点演算
$\mathbf{u} := 2^{-53}$: 浮動小数点演算の相対精度

たとえば，$a, b \in \mathbb{F}$に対して，次のような$x, y \in \mathbb{F}$を得られる.

Knuth (1969) $[x, y] = \mathbf{TwoSum}(a, b) \quad \Rightarrow \quad x + y = a + b$

Dekker (1971) $[x, y] = \mathbf{TwoProduct}(a, b) \quad \Rightarrow \quad x + y = a \times b$

Error-free! (情報の欠落がない)

TwoSum (Knuth, 1969)

$$[x, y] = \textbf{TwoSum}(a, b) \quad \Rightarrow \quad a + b = x + y \quad (|y| \leq \mathbf{u}\,|x|)$$

$\oplus, \ominus$: floating point addition and subtraction

function $[x, y] = \textbf{TwoSum}(a, b)$
$x = a \oplus b;$
$c = x \ominus a;$
$y = (a \ominus (x \ominus c)) \oplus (b \ominus c);$

Computational cost: 6 flops

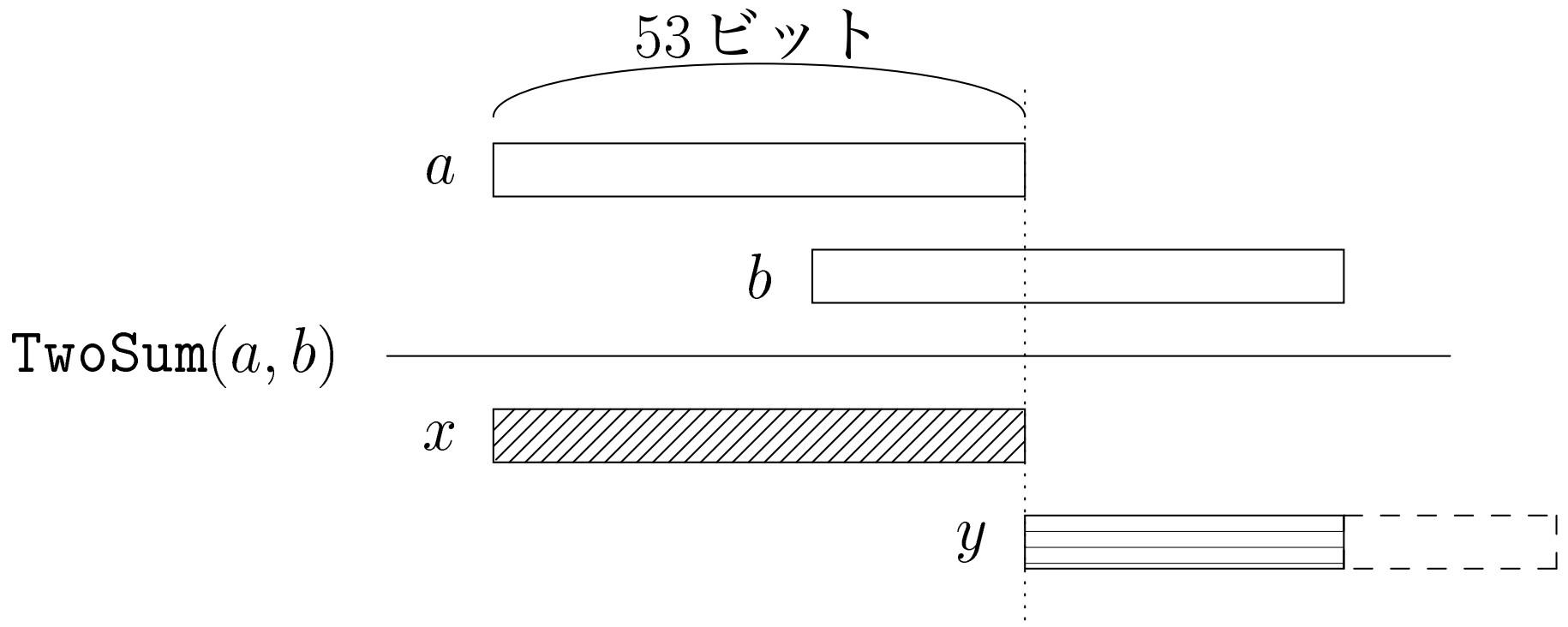


Figure 1: アルゴリズム **TwoSum** のイメージ．長方形が浮動小数点数を表し，左側にあるほど絶対値が大きいとする（この図では，$|a| > |b|$）．

TwoSum' (Dekker, 1971)

$$[x, y] = \textbf{TwoSum}'(a, b) \quad \Rightarrow \quad a + b = x + y \quad (|y| \leq \mathbf{u}\,|x|)$$

function $[x, y] = \textbf{TwoSum}'(a, b)$
$x = a \oplus b;$
if $|a| \geq |b|$
  $y = b \ominus (x \ominus a);$
else
  $y = a \ominus (x \ominus b);$

3 flops + taking 2 absolute values + 1 comparison

$\Longrightarrow$ Computational speed slows down

Split (Dekker, 1971)

$$[a_H, a_L] = \textbf{Split}(a) \quad \Rightarrow \quad a = a_H + a_L \quad (|a_H| \geq |a_L|)$$

$\otimes$: floating point multiplication

function $[a_H, a_L] = \textbf{Split}(a)$
$c = \texttt{factor} \otimes a;$ % $\texttt{factor} = 2^{\lceil \frac{t}{2} \rceil} + 1$
$a_H = c \ominus (c \ominus a);$
$a_L = a \ominus a_H;$

4 flops

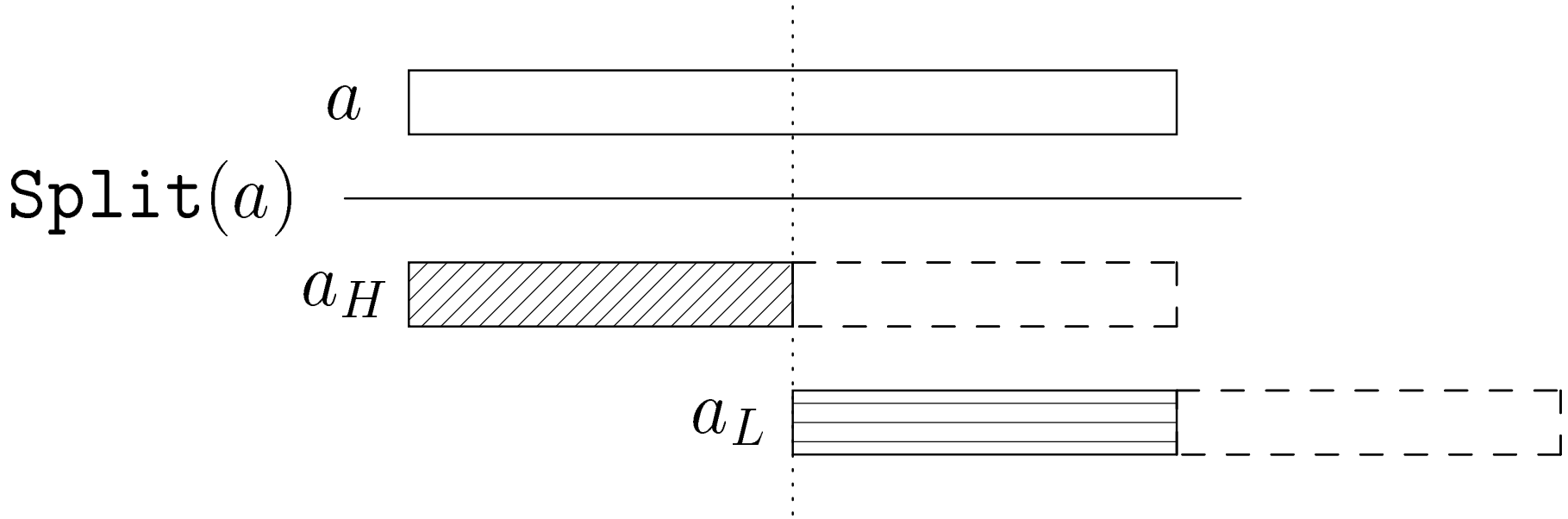


Figure 2: アルゴリズム**Split**のイメージ.

TwoProduct (Veltkamp [Dekker, 1971])

$$[x, y] = \textbf{TwoProduct}(a, b) \quad \Rightarrow \quad a \times b = x + y \quad (|y| \leq \mathbf{u}\,|x|)$$

function $[x, y] = \textbf{TwoProduct}(a, b)$
$x = a \otimes b;$
$[a_H, a_L] = \textbf{Split}(a);$ % $a_H + a_L \leftarrow a$
$[b_H, b_L] = \textbf{Split}(b);$ % $b_H + b_L \leftarrow b$
$y = a_L \otimes b_L \ominus (((x \ominus a_H \otimes b_H) \ominus a_L \otimes b_H) \ominus a_H \otimes b_L);$

17 flops

TwoProductFMA

FMA: Fused Multiply-Add

By **FMA**(a, b, c), $d := a * b + c$ can be calculated in 1 flop. Result d is rounded to the nearest floating point number.

function $[x, y] =$ **TwoProductFMA**(a, b)

$x = a \otimes b$;

$y =$ **FMA**$(a, b, -x)$;

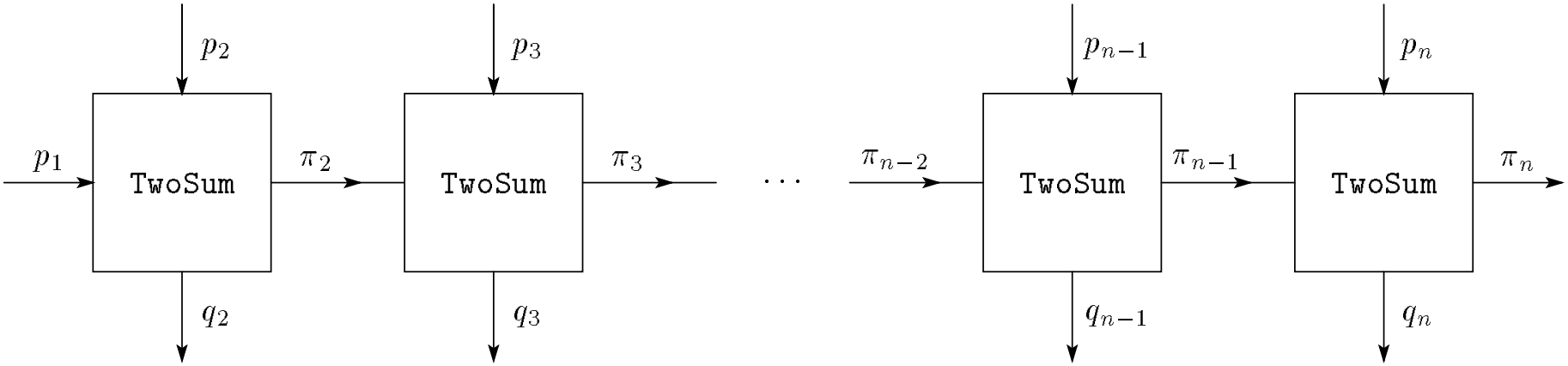
17 flops $\Longrightarrow$ 2 flops

計算精度（precision）を任意に上げるアルゴリズム

ベクトルの総和のerror-free変換

For $p \in \mathbb{F}^n$, we can obtain the following $p' \in \mathbb{F}^n$:

$$p' = \mathbf{VecSum}(p) \quad \Longrightarrow \quad \pi_n + \sum_{i=2}^{n} q_i = \sum_{i=1}^{n} p_i$$



VecSum

$$p' = \textbf{VecSum}(p) \quad \Rightarrow \quad \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i$$

(so-called distillation algorithm)

function $p = \textbf{VecSum}(p)$
for $i = 2 : n$
  $[p_i, p_{i-1}] = \textbf{TwoSum}(p_i, p_{i-1})$;

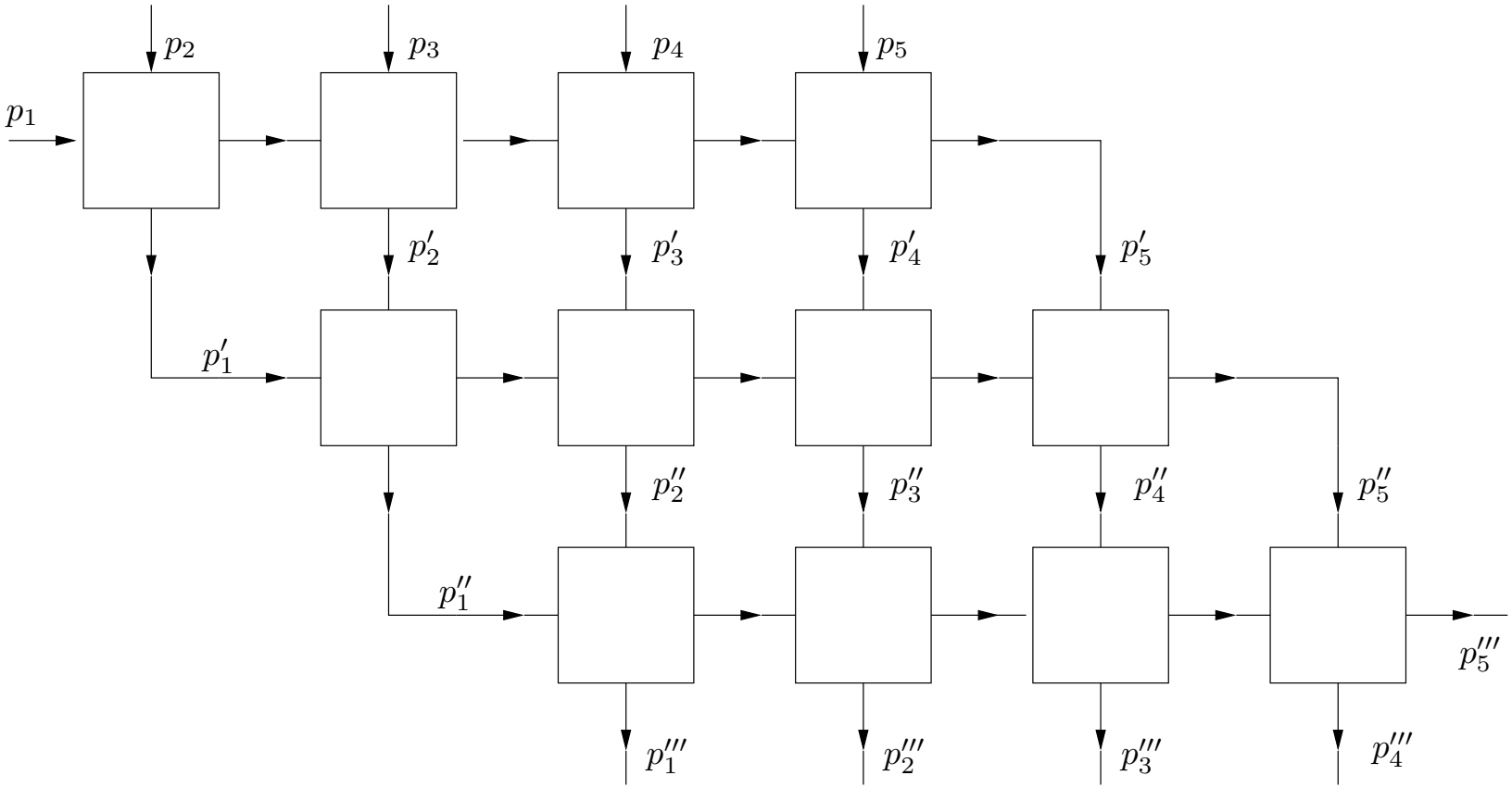
$6n$ flops

VecSum can be applied iteratively:

$$\sum_{i=1}^{n} p_i \xrightarrow{\textbf{VecSum}} \sum_{i=1}^{n} p'_i \xrightarrow{\textbf{VecSum}} \sum_{i=1}^{n} p''_i \xrightarrow{\textbf{VecSum}} \cdots$$

$$\Longrightarrow \quad \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} p'_i = \sum_{i=1}^{n} p''_i = \cdots$$

Error-free!



Example: $n = 5$, $K = 4$.

SumK

$$\textbf{res} = \textbf{SumK}(p, K) \quad \Rightarrow \quad \textbf{res} \approx \sum_{i=1}^{n} p_i$$

function $\textbf{res} = \textbf{SumK}(p, K)$
for $k = 1 : K - 1$
$p = \textbf{VecSum}(p);$
end
$\textbf{res} = p_n \oplus \mathrm{fl}\left(\sum_{i=1}^{n-1} p_i\right);$

$(6K - 5)n$ flops

SumKの誤差解析

$$\mathrm{cond}(\sum p_i) := \frac{\sum |p_i|}{|\sum p_i|} \quad \text{(総和の条件数)}$$

SumKの結果を**res** $\in \mathbb{F}$とすると

$$\frac{|\mathbf{res} - \sum p_i|}{|\sum p_i|} \leq \mathbf{u} + \mathcal{O}(\mathbf{u}^K)\mathrm{cond}(\sum p_i)$$

$\Longrightarrow$ 通常の計算精度のK倍

高精度内積

TwoProductを用いると内積計算に拡張可能

For $x, y \in \mathbb{F}^n$, using $[h_i, r_i] = \mathbf{TwoProduct}(x_i, y_i)$

$$\sum_{i=1}^{n} x_i\, y_i = \sum_{i=1}^{n} (h_i + r_i) \quad \left(= \sum_{i=1}^{2n} t_i \right)$$

Error-free! (provided no underflow occurs)

Dot2

function **res** = **Dot2**(x, y)
$p = 0;\ s = 0;$
for $i = 1 : n$
    $[h, r] =$ **TwoProduct**$(x_i, y_i);$
    $[p, q] =$ **TwoSum**$(p, h);$
    $s = s \oplus (q \oplus r);$
end
res $= p \oplus s;$

$25n$ flops

DotK

function **res** $=$ **DotK**(x, y, K)
$[p, r_1] =$ **TwoProduct**(x_1, y_1);
for $i = 2 : n$
    $[h, r_i] =$ **TwoProduct**(x_i, y_i);
    $[p, r_{n+i-1}] =$ **TwoSum**(p, h);
end
$r_{2n} = p$;
res $=$ **SumK**$(r, K-1)$;

$(12K + 1)n$ flops

DotKの誤差解析

$x, y \in \mathbb{R}^n$. 内積の条件数は

$$\mathrm{cond}(x^T y) = \frac{2\left|x^T\right| \ |y|}{|x^T y|}.$$

DotKによる結果を**res** $\in \mathbb{F}$とすると

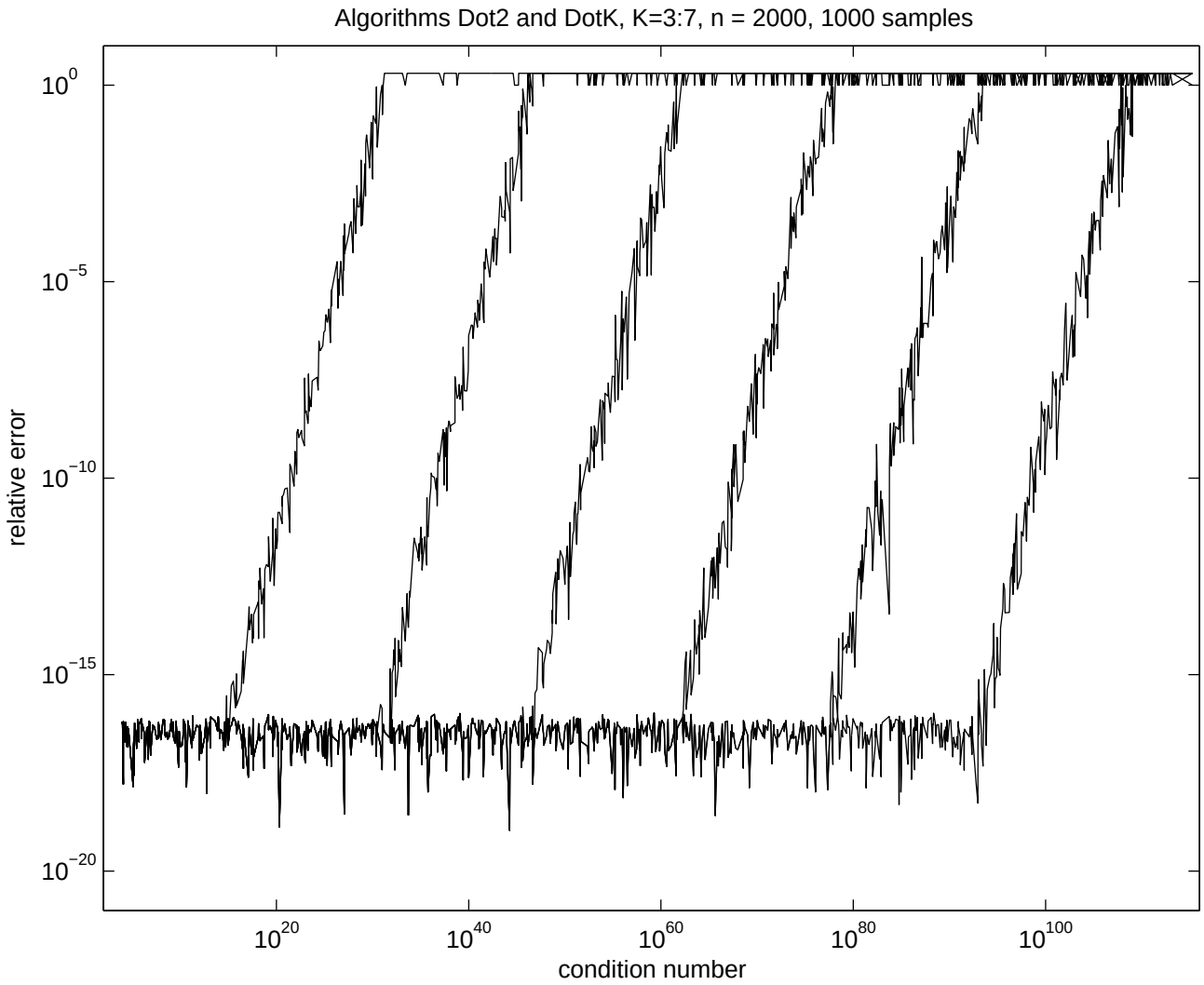
$$\frac{\left|\mathbf{res} - x^T y\right|}{|x^T y|} \leq \mathbf{u} + \mathcal{O}(\mathbf{u}^K)\mathrm{cond}(x^T y).$$

$\Longrightarrow$ 通常の計算精度のK倍

数値実験

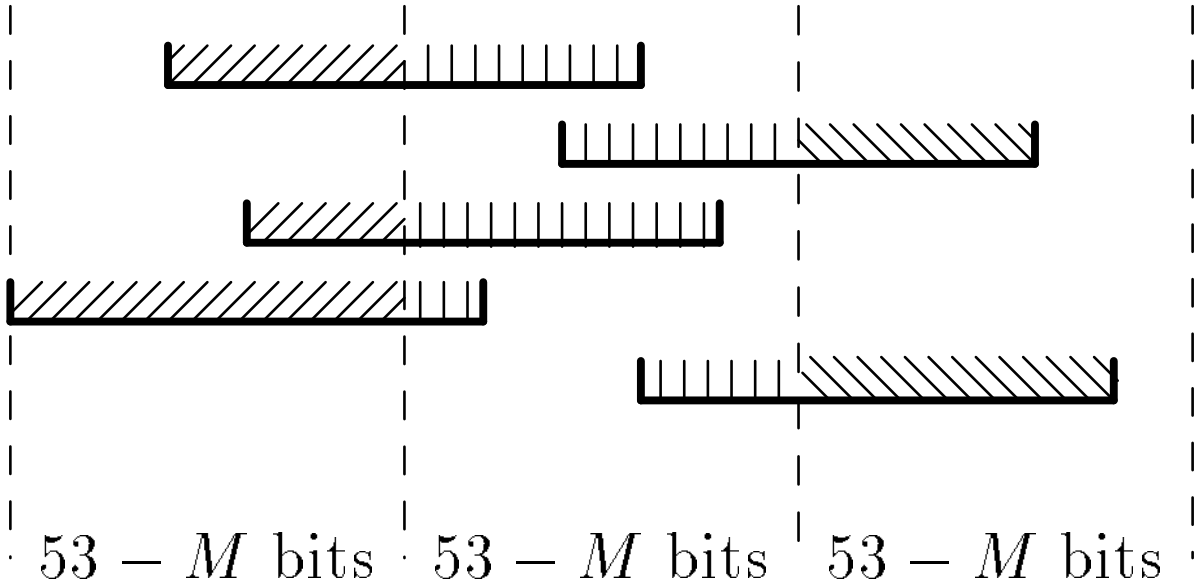
We evaluate accuracy of **Dot2** and **DotK**.

- $n = 2000$ (1000 samples)
- Elements of $x, y \in \mathbb{F}^n$: random numbers in $[-1, 1]$ s.t. $\mathrm{cond}(x^T y)$ is from 2 to 10^{120}
- Plot the relative error $\dfrac{|\mathbf{res} - x^T y|}{|x^T y|}$ where **res** is the result of **Dot2** or **DotK** $(K = 3 : 7)$

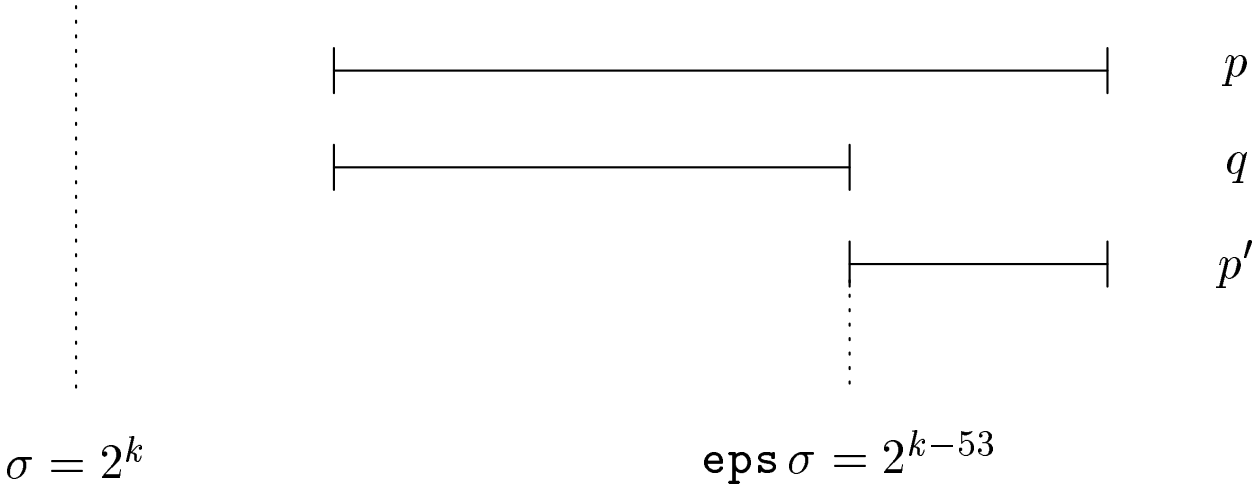

Algorithms Dot2 and DotK, K=3:7, n = 2000, 1000 samples
relative error
condition number
10^{0}
10^{-5}
10^{-10}
10^{-15}
10^{-20}
10^{20}
10^{40}
10^{60}
10^{80}
10^{100}

結果精度（accuracy）を保証するアルゴリズム

別のタイプのerror-free変換



ExtractScalar

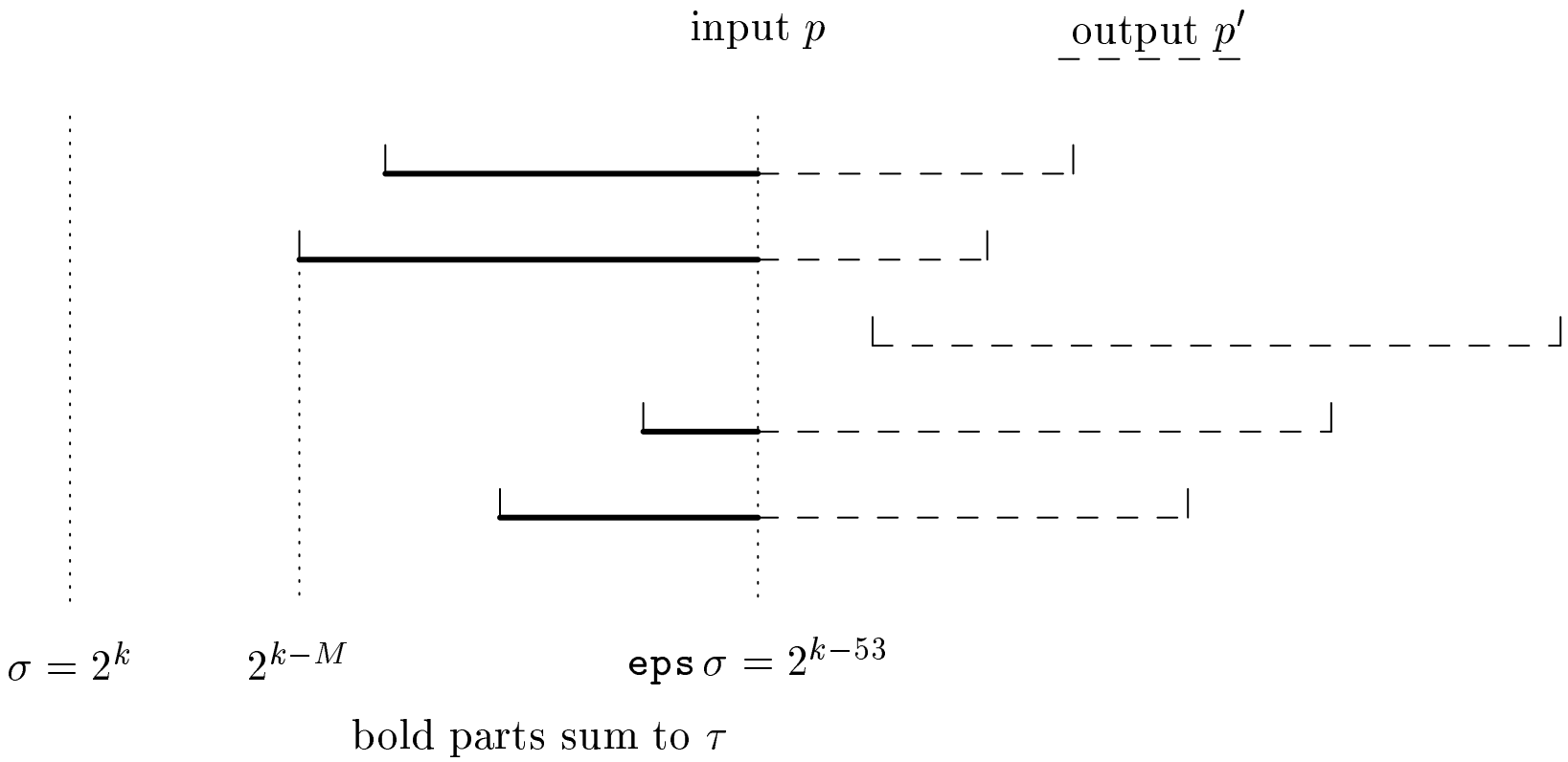


function $[q, p'] =$ **ExtractScalar**(σ, p)

$$q = \mathrm{fl}\left((\sigma + p) - \sigma\right)$$
$$p' = \mathrm{fl}\left(p - q\right)$$

$$\Longrightarrow \quad p = q + p' \quad \text{(Error-free!)}$$

ExtractVector



ExtractVector (提案方式の主計算)

function $[\tau, p'] = \textbf{ExtractVector}(\sigma, p)$

$\tau_0 = 0$

for $i = 1 : n$

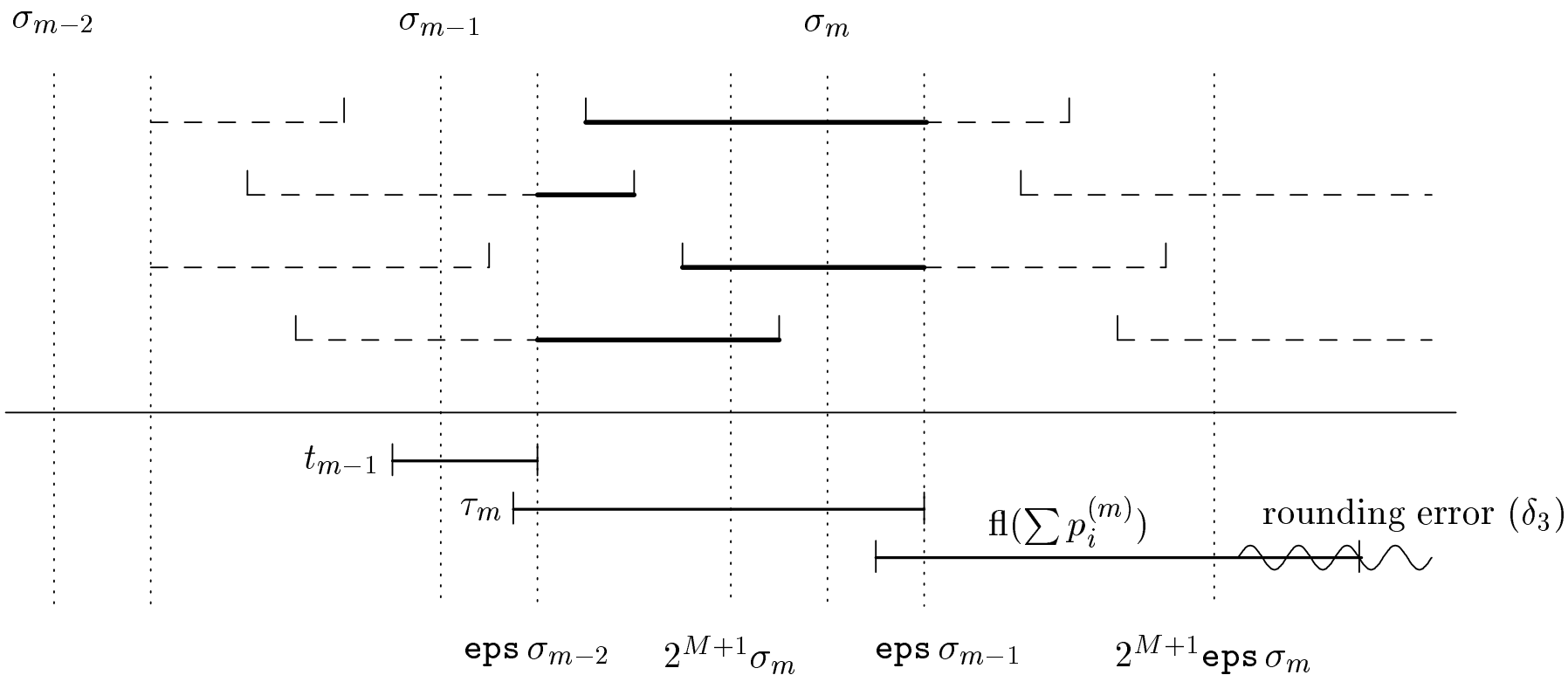
$[q_i, p'_i] = \textbf{ExtractScalar}(\sigma, p_i)$ % $p_i = q_i + p'_i$

$\tau_i = \mathrm{fl}\,(\tau_{i-1} + q_i)$ % $\tau_i = \tau_{i-1} + q_i$

end

$$\Longrightarrow \sum p_i = \tau_k + \sum p'_i, \quad 1 \le k \le n \quad \text{(Error-free!)}$$

提案方式（AccSum）の概要



提案方式の特性やlong accumulatorとの違い

- 欲しい精度の結果を得るために，適応的に必要な分だけの計算を実行
- コンパイラの最適化が掛かりやすい
- 非常に高速

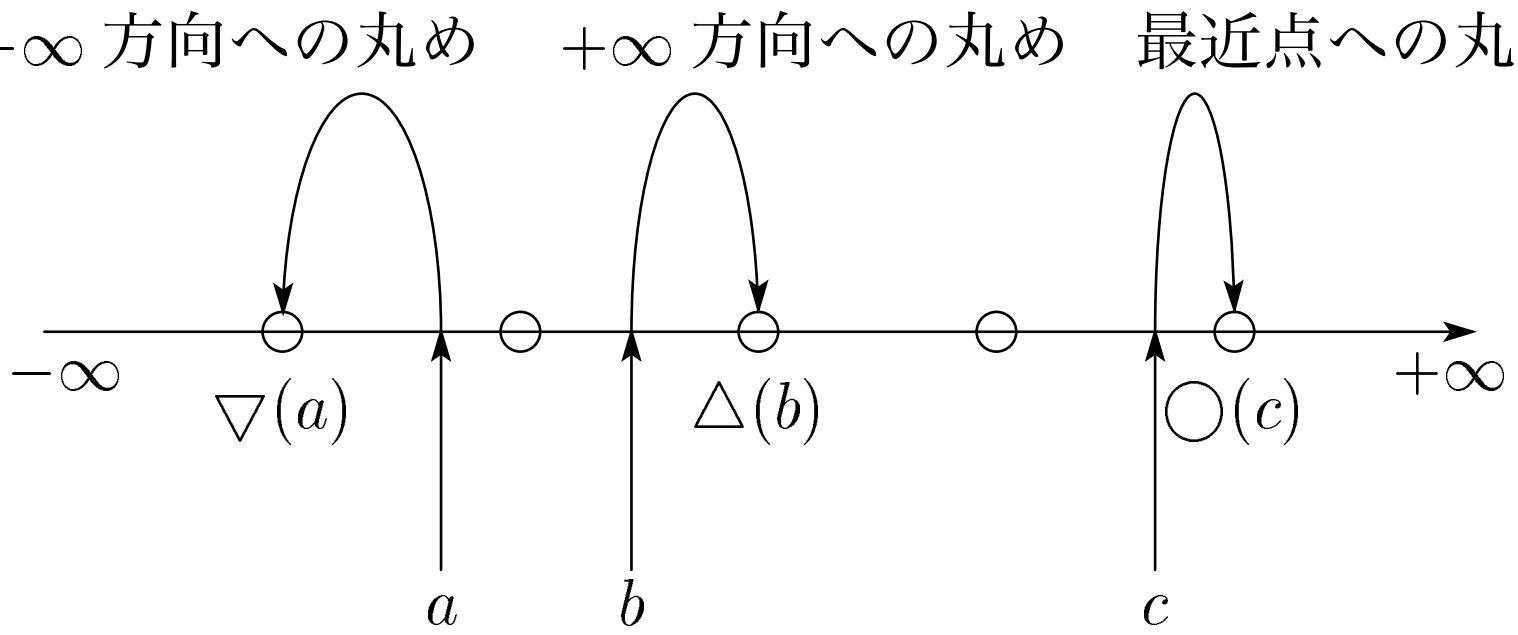
提案方式の誤差評価 (1)

提案方式で得られた結果を $\mathbf{res} \in \mathbb{F}$ とすると

$$\frac{|\mathbf{res} - \sum p_i|}{|\sum p_i|} < \mathbf{u} \quad \text{(faithful rounding of } \sum p_i\text{)}$$

$\Longrightarrow \sum p_i \in \mathbb{F}$ のとき，$\mathbf{res} = \sum p_i$.
$\Longrightarrow$ また，nearest roundingも計算可能:

$$\frac{|\mathbf{res} - \sum p_i|}{|\sum p_i|} \leq \frac{1}{2}\mathbf{u} \quad \text{(nearest rounding of } \sum p_i\text{)}$$


$-\infty$ 方向への丸め
$+\infty$ 方向への丸め
最近点への丸め
$-\infty$
$+\infty$
$\bigtriangledown(a)$
$\bigtriangleup(b)$
$\bigcirc(c)$
a
b
c

提案方式の誤差評価 (2)

さらに，$\mathbf{res}_{1:K} := \mathbf{res}_1 + \mathbf{res}_2 + \cdots + \mathbf{res}_K$，$(\mathbf{res}_k \in \mathbb{F})$ と出力を nonoverlapping な K 個の浮動小数点数の和として表現することによって

$$\frac{|\mathbf{res}_{1:K} - \sum p_i|}{|\sum p_i|} \leq \frac{2}{1-\mathbf{u}}\mathbf{u}^K \quad (K\text{-fold accuracy})$$

及び

$$\left|\mathbf{res}_{1:K} - \sum p_i\right| \leq 2\mathbf{u}^K \left|\mathbf{res}_1\right|$$

以上の議論はアンダーフローが起きても成立.

内積計算

内積計算$x^T y$, $(x, y \in \mathbb{F}^n)$については，ベクトルの総和$\sum_{i=1}^{2n} p_i$に帰着できるので，同様の議論で高精度な内積計算が可能.

参考: Dekker (1971)
$[x, y] = \mathbf{TwoProduct}(a, b) \quad \Rightarrow \quad x + y = a \times b$

数値実験(1)

提案方式（AccSum）の性能を評価する.

- $\sum_{i=1}^{n} p_i$ を計算
- $\mathrm{cond}(\sum p_i)$ は4倍精度演算に有利な 10^{16} に固定
- 通常の倍精度計算，以前開発した高精度計算Sum2(約4倍精度), XBLAS (約4倍精度) と比較

Table 1: Floating-point operations needed for different dimensions and condition numbers

n	cond	DSum	AccSum	Sum2	XBLAS
1000	10^6	n	$7n$	$7n$	$10n$
1000	10^{16}	n	$11n$	$7n$	$10n$
10^6	10^{16}	n	$15n$	$7n$	$10n$

DSum: 通常の倍精度演算
AccSum: 提案方式（結果精度が保証されている, faithful）
Sum2: 計算精度が2倍（約4倍精度）
XBLAS: 計算精度が2倍（BLASの高精度版）

Table 2: Measured computing times for $\mathrm{cond} = 10^{16}$, time of DSum normed to 1

CPU	Intel Pentium 4 (2.53GHz)		
Compiler	Intel Visual Fortran 9.1		
n	Sum2	XBLAS	AccSum
100	24.1	75.9	14.1
400	26.9	81.9	13.1
1,600	20.0	57.8	9.6
6,400	20.0	57.8	9.6
25,600	20.4	59.1	13.8

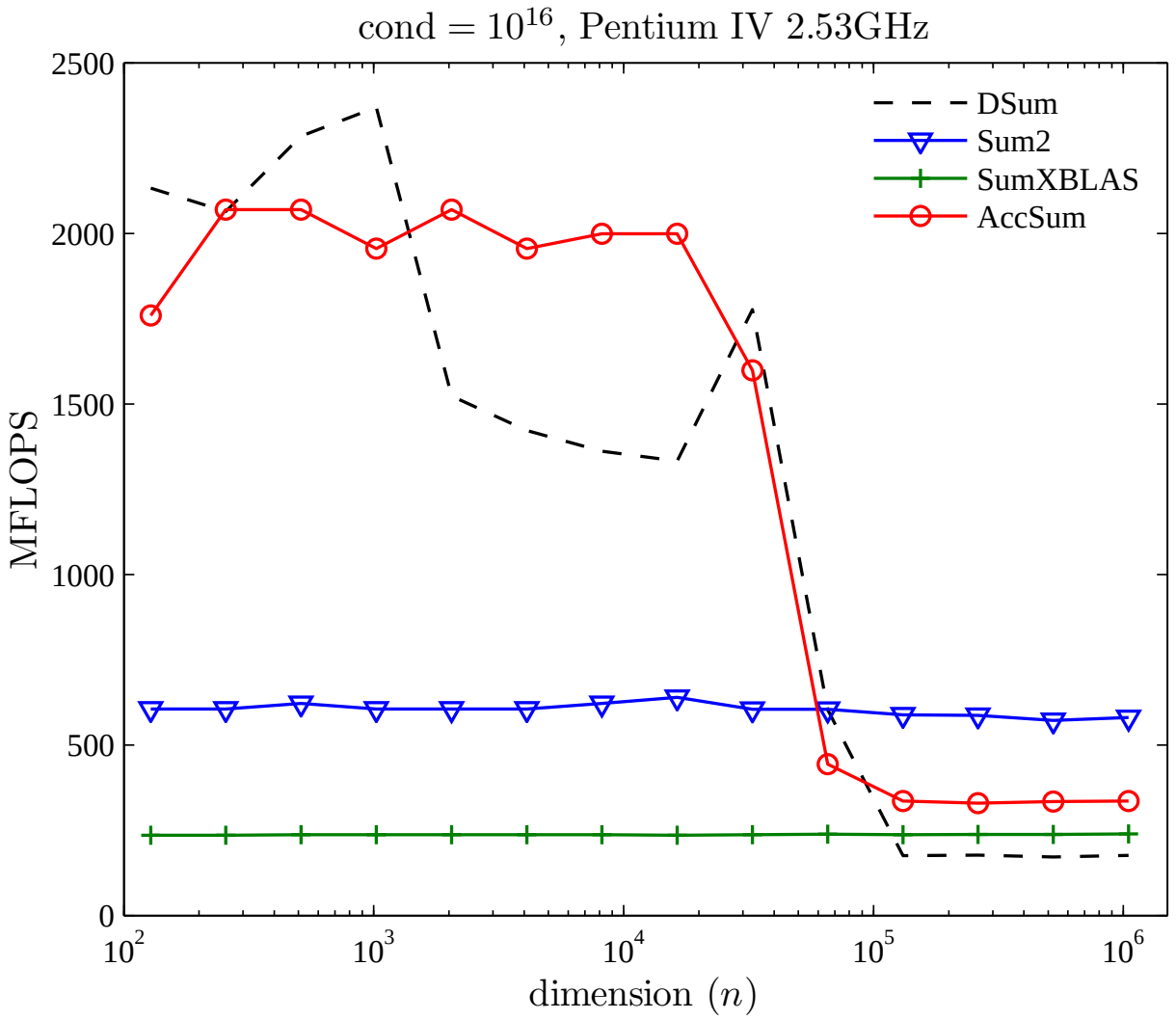

cond = 10^16, Pentium IV 2.53GHz
MFLOPS
dimension (n)
DSum
Sum2
SumXBLAS
AccSum
2500
2000
1500
1000
500
0
10^2
10^3
10^4
10^5
10^6

Table 3: Measured computing times for $\mathrm{cond} = 10^{16}$, time of DSum normed to 1

CPU	Intel Itanium 2 (1.4GHz)		
Compiler	Intel Fortran 9.0		
n	Sum2	XBLAS	AccSum
100	2.9	17.3	7.8
400	4.7	31.6	10.7
1,600	7.3	50.5	15.8
6,400	7.8	54.5	16.5
25,600	7.9	55.6	21.5
102,400	7.3	50.0	25.6

Table 4: Measured computing times for $\mathrm{cond} = 10^{16}$, time of DSum normed to 1

CPU	AMD Athlon 64 (2.2GHz)		
Compiler	GNU gfortran 4.1.1		
n	Sum2	XBLAS	AccSum
100	2.0	4.8	2.8
400	3.1	7.6	4.1
1,600	3.1	7.7	4.1
6,400	3.1	7.7	4.2
25,600	3.1	7.7	5.7
102,400	2.3	5.7	8.0

数値実験(2)

次に，提案方式（AccSum）と結果精度を保証する他のアルゴリズムの計算時間を比較する

DSum: 通常の倍精度演算
AccSum: 提案方式 (faithful)
Priest: Priest's doubly compensated summation (faithful)
Malcolm: Malcolm's efficient long accumulator (faithful)
LongAccu: traditional long accumulator (faithful)

Table 5: Measured computing times for $\mathrm{cond} = 10^{16}$, time of DSum normed to 1

CPU	Intel Pentium 4 (2.53GHz)			
Compiler	Intel Visual Fortran 9.1			
n	Priest	Malcolm	LongAccu	AccSum
100	187.1	175.9	711.2	14.1
400	311.9	148.1	761.9	13.1
1,600	305.7	97.8	540.9	9.6
6,400	345.7	96.5	540.9	9.6
25,600	407.1	98.7	554.2	13.8

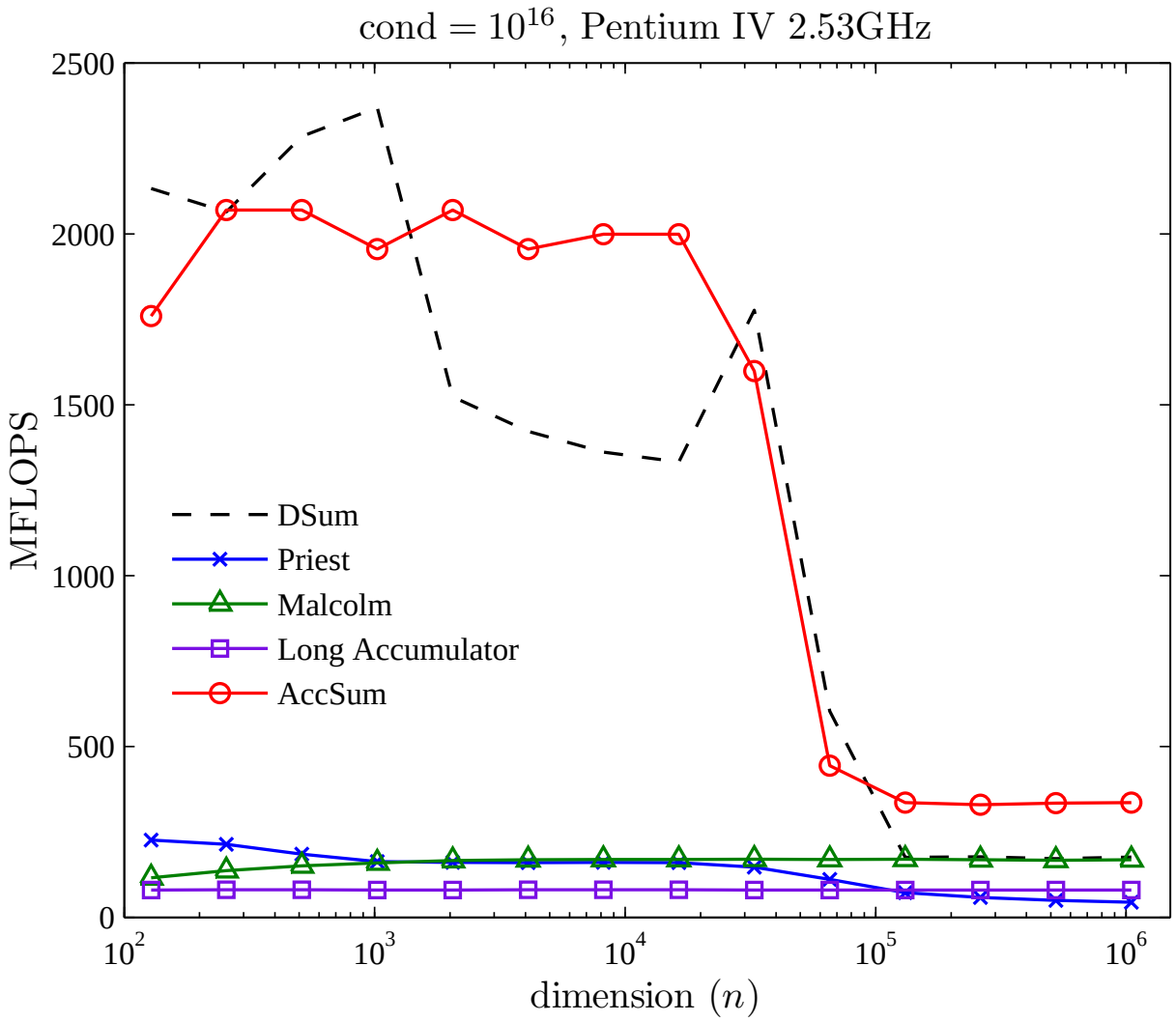

cond = 10^16, Pentium IV 2.53GHz
MFLOPS
dimension (n)
2500
2000
1500
1000
500
0
10^2
10^3
10^4
10^5
10^6
DSum
Priest
Malcolm
Long Accumulator
AccSum

Table 6: Measured computing times for $\mathrm{cond} = 10^{16}$, time of DSum normed to 1

CPU	Intel Itanium II (1.4GHz)			
Compiler	Intel Fortran Compiler 9.0			
n	Priest	Malcolm	LongAccu	AccSum
100	129.7	49.0	241.5	7.8
400	317.4	43.6	448.1	10.7
1,600	609.5	50.9	717.6	15.8
6,400	797.8	49.3	767.5	16.5
25,600	957.0	49.4	790.2	21.5
102,400	1056.2	43.8	711.3	25.6

結論

- 計算精度や結果精度を指定できる高速なベクトルの総和・内積アルゴリズムを提案した
- 計算量だけでなく実行時間の意味でも非常に高速
- 通常の浮動小数点操作しか用いていないのでポータブルかつスケーラブル