Robust Adaptive Floating-Point Geometric Predicates

Jonathan Richard Shewchuk
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213
jrs@cs.cmu.edu

+ additional notes by Petr Felkel, CTU Prague, 2020-2023

Version from 05.10.2023

Precise floats represented as expansions

Just the idea, not using IEEE float, but 4-digit decimal numbers …

$$\boxed{-\ 1.50000000000000000000000000000800000000000000000000000000000000077 \ast 10^{-1}}$$

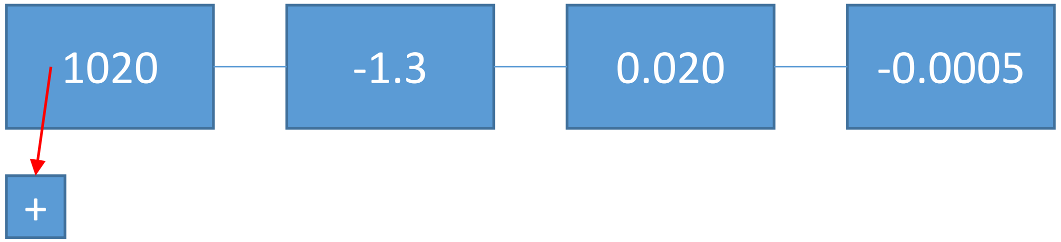
$$\boxed{-\ 1.500 \ast 10^{-1}} + \boxed{-\ 8.000 \ast 10^{-31}} + \boxed{-\ 7.700 \ast 10^{-64}}$$

Expansion

- **Sorted** sequence of **non-overlapping** machine native numbers (float, double) – each with its own exponent and significand (mantissa)
- Sorted by **absolute values**
- **Signum** of the highest FP number is the signum of the expansion
- Zero members of the expansion will be not added.

```
1020
-   1.3
-------
1018.7
+   0.02
---------
1018.7200
-   0.0005
---------
1018.7195
```

$|x_4| \quad > \quad |x_3| \quad > \quad |x_2| \quad > \quad |x_1|$



represents $x = +1018.7195$

approximated $x \sim +1020 = x_4$

Expansions are not unique

binary	decimal (overlap)
1001.1	$\dots 9.5$

Possibly stored as

$1100 \ + \ (-10.1)$	$\dots 12 + (-2.5)$
$= \ 1100.0 - \ 10.1$	$\dots 12 \ - 2.5$
$= \ 1001 \ + \ 0.1$	$\dots \ \ 9 + 0.5$
$= \ 1000 \ + \ 1 \ + \ 0.1$	$\dots \ \ 8 + 1 + 0.5$

Meaning of symbols

p-bit floating point operations with exact rounding (float, double):

$\oplus$ addition

$\ominus$ subtraction

$\otimes$ multiplication

Perform the operation with higher precision

Round the result to the representable number

Exact rounding

Operations with exact rounding to p-bits (32 / 64) store result:

exact results store exact, and
non-precise results store rounded

More than 4-bits arithmetic (precise)	With exact rounding to 4-bits
$010 \times 011 \;=\; 100$ $2 \times 3 = 6$	$010 \otimes 011 \;=\; 100$ $2 \otimes 3 = 6$
$111 \times 101 \;=\; 100011$ $7 \times 5 = 35$	$111 \otimes 101 \;=\; 1.001 \times 2^5$ $7 \otimes 5 \;=\; 36$

if (possible)
    store exact
else
    store rounded

Operations on expansions

IEEE 754 standard on floating point format and computing rules.

Operations on expansions require *exact rounding* of each op. to 32 / 64bit.

```
Fast-Two-Sum(a, b) :  (a>=b) -> (x, y),          a+b=x+y
Two-Sum(a, b) -> (x, y)
Linear-Expansion-Sum(a interleaved with b) -> correct expansion
                                               (non-overlapping)

Split (a) -> (a_hi, a_lo),           a = a_hi + a_lo
Two-Product (a,b) -> (x, y)
```

Theorem 1 (Dekker [4]) *Let a and b be p-bit floating-point numbers such that $\boxed{|a| \geq |b|}$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$, where $\boxed{x}$ is an approximation to $a + b$ and $\boxed{y}$ represents the roundoff error in the calculation of x.* $\blacksquare$

2-bits mantissa

$8 \Leftarrow 6 \oplus 1$

$2 \Leftarrow 8 \ominus 6$

$-1 \Leftarrow 1 \ominus 2$

return $(8, -1)$

FAST-TWO-SUM(a, b)

```
x ⇐ a ⊕ b                     // Rounded sum = approximation
b_virtual ⇐ x ⊖ a             // What was truly added - Rounded
y ⇐ b ⊖ b_virtual             // round-off error
return (x, y)
```

$\textsc{Fast-Two-Sum}(a, b)$

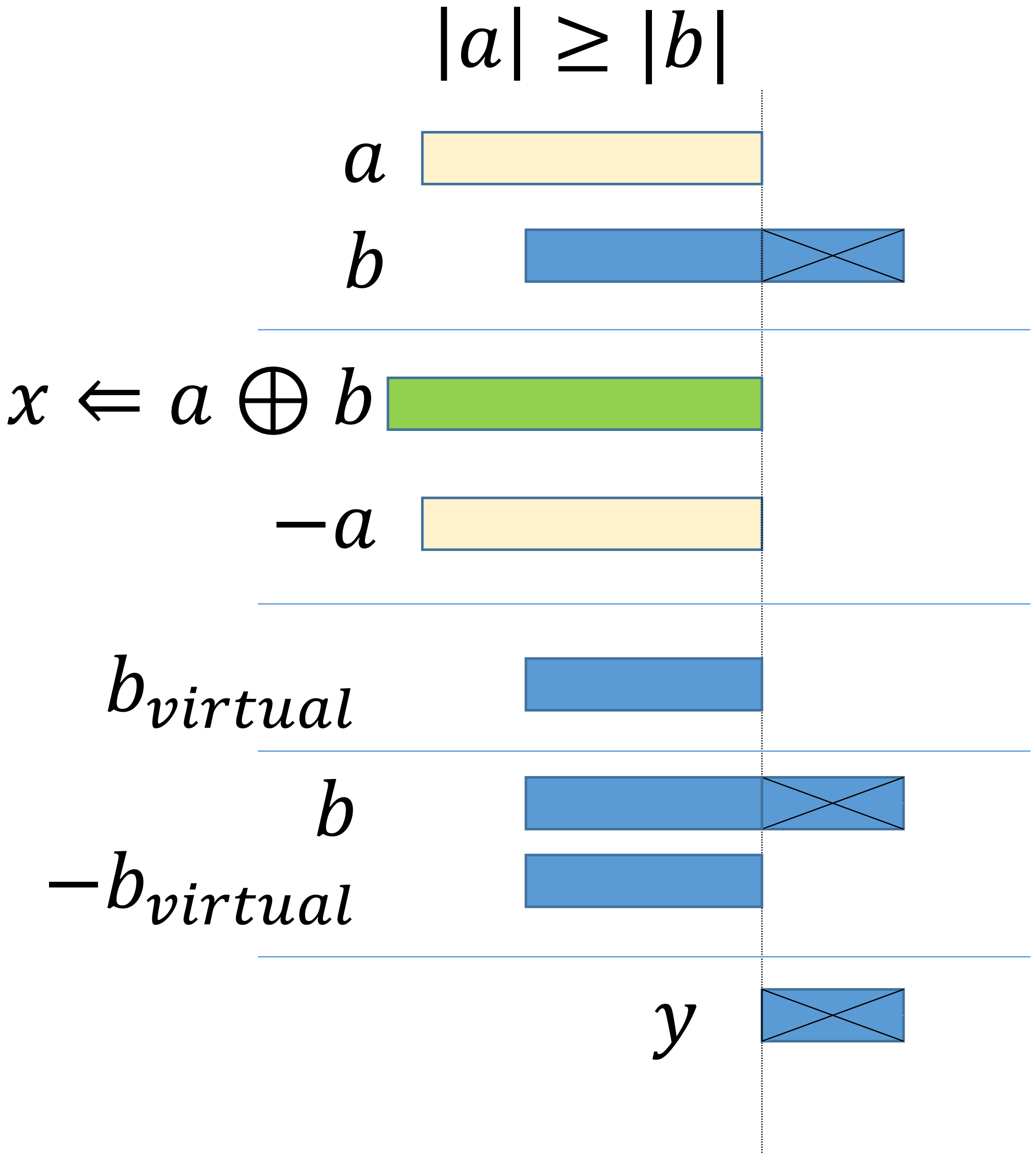
$x \Leftarrow a \oplus b$

$b_{\text{virtual}} \Leftarrow x \ominus a$

$y \Leftarrow b \ominus b_{\text{virtual}}$

return (x, y)

$$a + b = x + y$$
$$= a \oplus b + \; b \ominus b_{virtual}$$



Fast TwoSum with result rounded up (on 4-digits decimal numbers)

Correct		Rounded up	Really added	Correction
$a = 5081$		$a = 5081$	$x = 5175$	$b = 93.5$
$b = 93.5$		$b = 93.5$	$-a = -5081$	$-b_{virtual} = -94$
----------		----------	----------	----------
$5174.\underline{5}$	$\xrightarrow{\text{up}}$	$x = 5175$	$b_{virtual} = 94$	$y = -0.5$

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

$$5081 + 93.5 = (5175 - 0.5)$$

Fast TwoSum with result rounded down (on 4-digits decimal numbers)

Correct	Rounded down	Really added	Correction
$a = 5081$	$a = 5081$	$x = 5174$	$b = 93.4$
$b = 93.4$	$b = 93.4$	$-a = -5081$	$-b_{virtual} = -94$
----------	----------	----------	----------
$5174.\underline{4}$ $\xrightarrow{\text{down}}$	$x = 5174$	$b_{virtual} = 93$	$y = 0.4$

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

$$5081 + 93.4 = (5174 + 0.4)$$

Theorem 2 (Knuth [10]) *Let a and b be p-bit floating-point numbers, where $p \geq 3$. Then the following algorithm will produce a nonoverlapping expansion $x + y$ such that $a + b = x + y$.* ■

TWO-SUM(a, b)

$x \Leftarrow a \oplus b$ // Rounded sum = approximation

$b_{\text{virtual}} \Leftarrow x \ominus a$ // What b was truly added – Rounded for $a > b$

$a_{\text{virtual}} \Leftarrow x \ominus b_{\text{virtual}}$ // What a was truly added – Rounded for $b > a$

$b_{\text{roundoff}} \Leftarrow b \ominus b_{\text{virtual}}$ // round-off error of b

$a_{\text{roundoff}} \Leftarrow a \ominus a_{\text{virtual}}$ // round-off error of a

$y \Leftarrow a_{\text{roundoff}} \oplus b_{\text{roundoff}}$

return (x, y)

Sum of two expansions (4-bit arithmetic)

Input: 1111+0.1001 and 1100 + 0.1

Output: 11100 + 0 + 0.0001

Zeroes slow down the computation – removed afterwards

1. Merge both input expansions into a single sequence g respecting the order of magnitudes

1111+ 1100 + 0.1001 + 0.1 numbers in the sequence overlap

2. Use LINEAR-EXPANSION-SUM (g) to create a correct expansion

$$g5 + g4 + g3 + g2 + g1 \rightarrow \mathrm{h}5 + h4 + h3 + h2 + h1$$

overlapping input → non-overlapping output

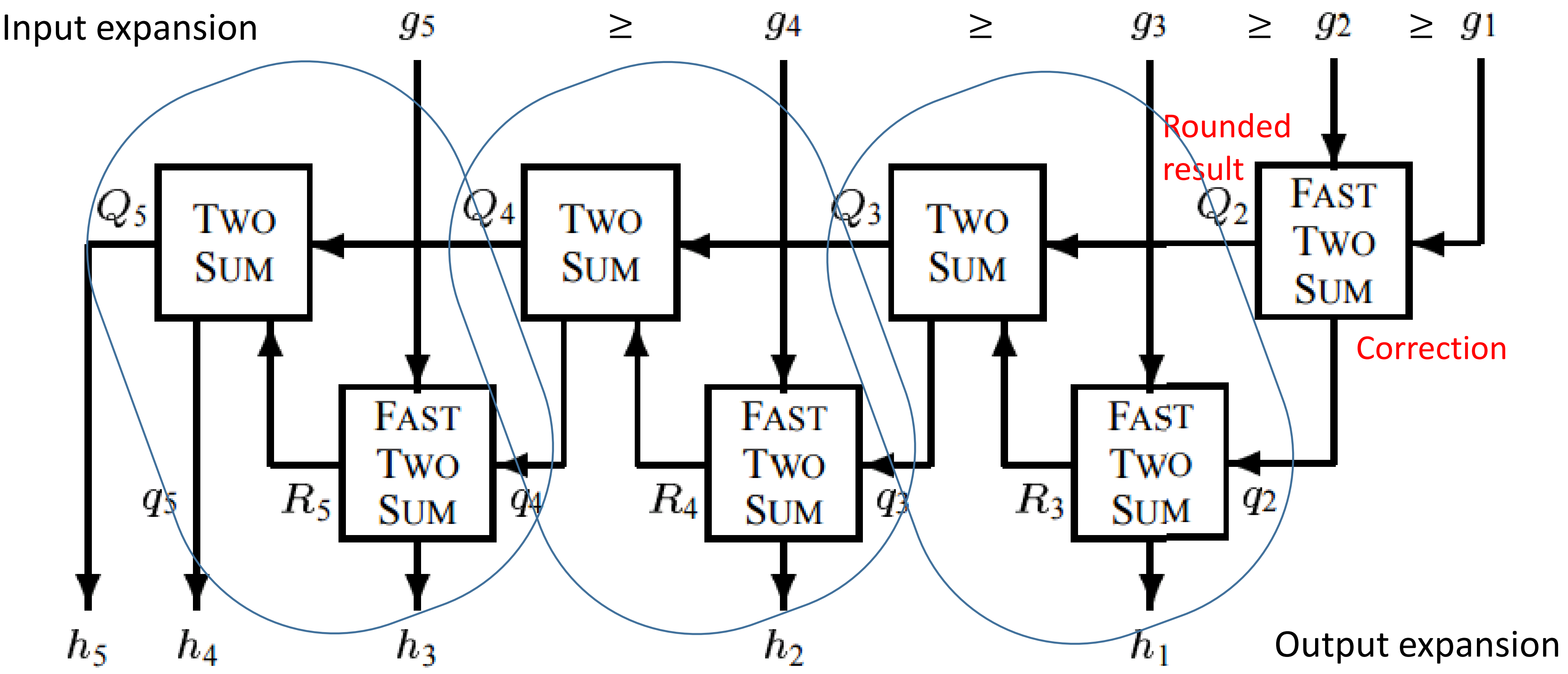


Figure 1: Operation of LINEAR-EXPANSION-SUM. The expansions g and h are illustrated with their most significant components on the left. $Q_i + q_i$ maintains an approximate running total. The FAST-TWO-SUM operations in the bottom row exist to clip a high-order bit off each q_i term, if necessary, before outputting it.

LINEAR-EXPANSION-SUM

1111+ 1100 + 0.1001 + 0.1

```
1111
1100
   0.1001
   0.1
----------------
11100.0001
```

```
 1111           1100          0.1001
 1101              1          0.1
------------    --------      ----------
11100           1101+0        1 + 0.0001
```

11100 + 0 + 0.0001

11100 + 0.0001

Multiplication

Multiplies two p-bit values a and b

1. Split both p-bit values into two halves (with ~p/2 bits)
2. perform four exact multiplications on these fragments.

$$a_{hi} \times a_{hi}, a_{hi} \times a_{lo}, a_{lo} \times a_{hi}, a_{lo} \times a_{lo},$$

The trick is to find a way to split a floating-point value into two.

SPLIT(a) operation

- Splits p bits into two non-overlapping halves ($\left\lfloor\frac{p}{2}\right\rfloor \text{bits } \mathrm{a}_{\mathrm{hi}}$ and $\left\lceil\frac{p}{2}\right\rceil - 1 \ \text{bits } \mathrm{a}_{lo}$)
- Missing bit is hidden in the signum of a_{lo}
- Example

7bit number splits to two 3 bit significands

1001001 splits to 1010000 (101×2^4) and -111

73 = 80 - 7

Theorem 4 (Dekker [4]) *Let a be a p-bit floating-point number, where $p \geq 3$. The following algorithm will produce a $\lfloor \frac{p}{2} \rfloor$-bit value a_{hi} and a nonoverlapping $(\lceil \frac{p}{2} \rceil - 1)$-bit value a_{lo} such that $|a_{\text{hi}}| \geq |a_{\text{lo}}|$ and $a = a_{\text{hi}} + a_{\text{lo}}$.* ■

$\boxed{\textsc{Split}(a)}$

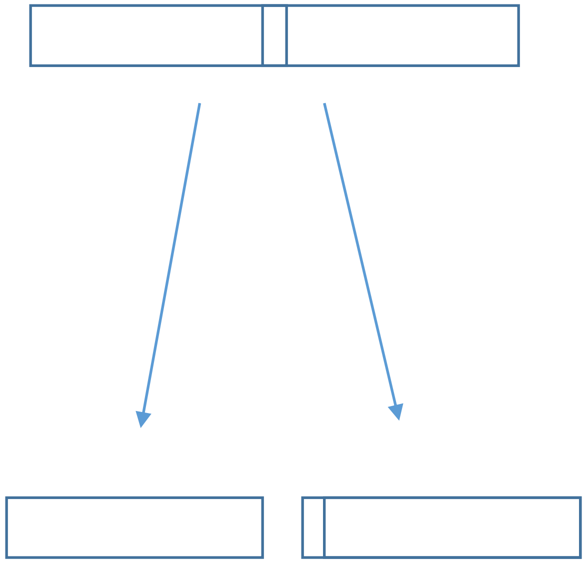
$c \Leftarrow (2^{\lceil p/2 \rceil} + 1) \otimes a$

$a_{\text{big}} \Leftarrow c \ominus a$

$a_{\text{hi}} \Leftarrow c \ominus a_{\text{big}}$

$a_{\text{lo}} \Leftarrow a \ominus a_{\text{hi}}$

return $(a_{\text{hi}}, a_{\text{lo}})$



Theorem 5 (Veltkamp) *Let a and b be p-bit floating-point numbers, where $p \geq 4$. The following algorithm will produce a nonoverlapping expansion $x + y$ such that $ab = x + y$.* $\blacksquare$

$\textsc{Two-Product}(a, b)$

$\quad x \Leftarrow a \otimes b$

$\quad (a_{\text{hi}}, a_{\text{lo}}) = \textsc{Split}(a)$

$\quad (b_{\text{hi}}, b_{\text{lo}}) = \textsc{Split}(b)$

$\quad err_1 \Leftarrow x \ominus (a_{\text{hi}} \otimes b_{\text{hi}})$

$\quad err_2 \Leftarrow err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}})$

$\quad err_3 \Leftarrow err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}})$

$\quad y \Leftarrow (a_{\text{lo}} \otimes b_{\text{lo}}) \ominus err_3$

$\quad$ **return** (x, y)

Demonstration of SPLIT splitting a five-bit number into two two-bit numbers

a	$=$							1	1	1	0	1		29
$2^3 a$	$=$				1	1	1	0	1				$\times 2^3$	232
c	$=$	$(2^3+1)\otimes a$	$=$	1	0	0	0	0	0	1	0	1	$\times 2^4$	$261 \rightarrow 256$
														-29
a	$=$							1	1	1	0	1		
a_{big}	$=$	$c \ominus a$	$=$		1	1	1	0	0	0	1	1	$\times 2^3$	$224 \leftarrow 227$
														$256 - 224$
a_{hi}	$=$	$c \ominus a_{\text{big}}$	$=$				1	0	0	0	0		$\times 2^1$	32
														$29 - 32$
a_{lo}	$=$	$a \ominus a_{\text{hi}}$	$=$							$-$	1	1		-3

$\boxed{29 \rightarrow (32,\ -3)}$

Demonstration of TWO-PRODUCT in six-bit arithmetic

$$
\begin{array}{rcccccccccccccccl}
a & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
b & = & & & & & & & & & 1 & 1 & 1 & 0 & 1 & 1 & \\
\cline{5-16}
x & = & a \otimes b & = & 1 & 1 & 0 & 1 & 1 & 0 & & & & & & & \times 2^6 \\
 & & a_{\text{hi}} \otimes b_{\text{hi}} & = & 1 & 1 & 0 & 0 & 0 & 1 & & & & & & & \times 2^6 \\
\cline{5-10}
err_1 & = & x \ominus (a_{\text{hi}} \otimes b_{\text{hi}}) & = & & & & 1 & 0 & 1 & 0 & 0 & 0 & & & & \times 2^3 \\
 & & a_{\text{lo}} \otimes b_{\text{hi}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
\cline{8-14}
err_2 & = & err_1 \ominus (a_{\text{lo}} \otimes b_{\text{hi}}) & = & & & & & 1 & 0 & 0 & 1 & 1 & 0 & & & \times 2^2 \\
 & & a_{\text{hi}} \otimes b_{\text{lo}} & = & & & & & 1 & 0 & 1 & 0 & 1 & 0 & & & \times 2^2 \\
\cline{9-14}
err_3 & = & err_2 \ominus (a_{\text{hi}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 0 & 0 & 0 & 0 & \\
 & & a_{\text{lo}} \otimes b_{\text{lo}} & = & & & & & & & & & 1 & 0 & 0 & 1 & \\
\cline{11-16}
-y & = & err_3 \ominus (a_{\text{lo}} \otimes b_{\text{lo}}) & = & & & & & & & - & 1 & 1 & 0 & 0 & 1 &
\end{array}
$$

The resulting expansion is $110110 \times 2^6 + 11001$

$54 * 2^6 + 25$

$\boxed{56^2 = 3481 \rightarrow (3456 + 25)}$

Adaptive arithmetic

- Expensive – avoid when possible
- Some applications need results with absolute error below a threshold
- Set of procedures with different precision (& speed) + error bounds
- For each input – compute the error bounds and choose the procedure

But

- Sometimes hard to determine error before computation
- Especially when relative error needed – like sign of expression compar.
  - Result can be much larger than error bound – rounded arithmetic will suffice
  - Result can be near zero – must be evaluated exactly

Shewchuk predicates

- Compute a sequence of increasingly accurate results
- Testing each for accuracy
- Not using separate procedures BUT
- Using intermediate results as steps to more accurate results (work already done is not discarded, but refined)
- Idea: presented routines can be split to two parts
  - Line 1 gives an approximate result - run each time
  - Remaining lines compute the roundoff error – delayed until needed, if ever …

Principle of adaptive computation

Distance of two points $$(b_x - a_x)^2 + \left(b_y - a_y\right)^2$$

Store $b_x - a_x$ as $x_1 + y_1$

and $b_y - a_y$ as $x_2 + y_2$

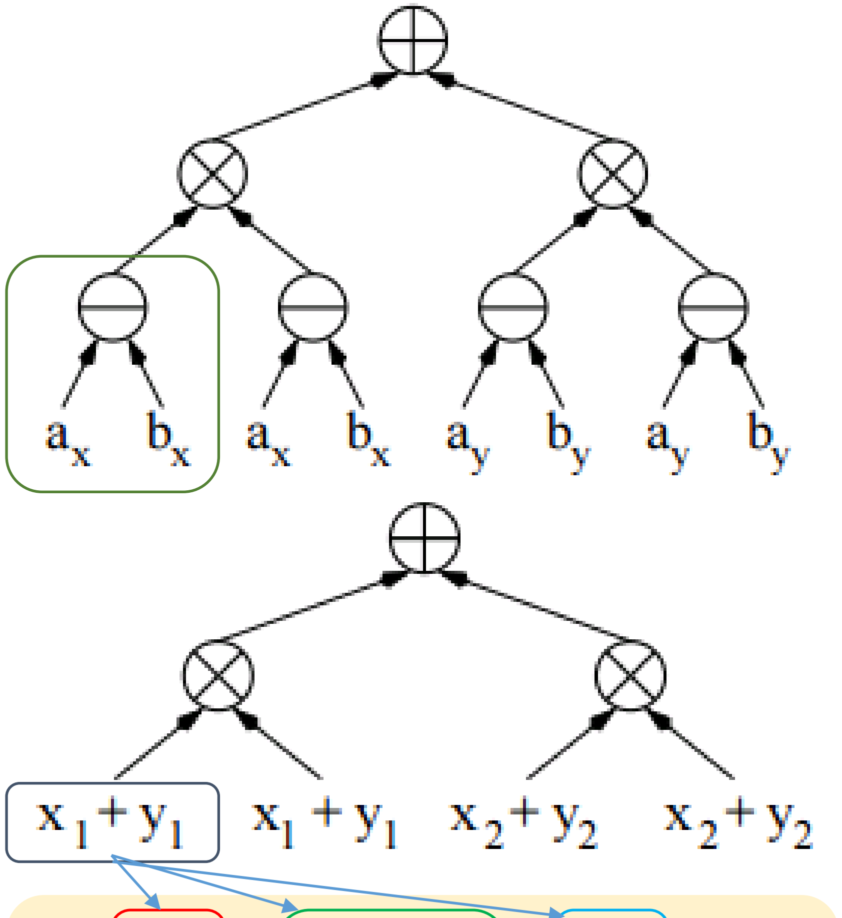
$$(x_1^2 + 2x_1y_1 + y_1^2) + (x_2^2 + 2x_2y_2 + y_2^2)$$

Reorder terms according to their size

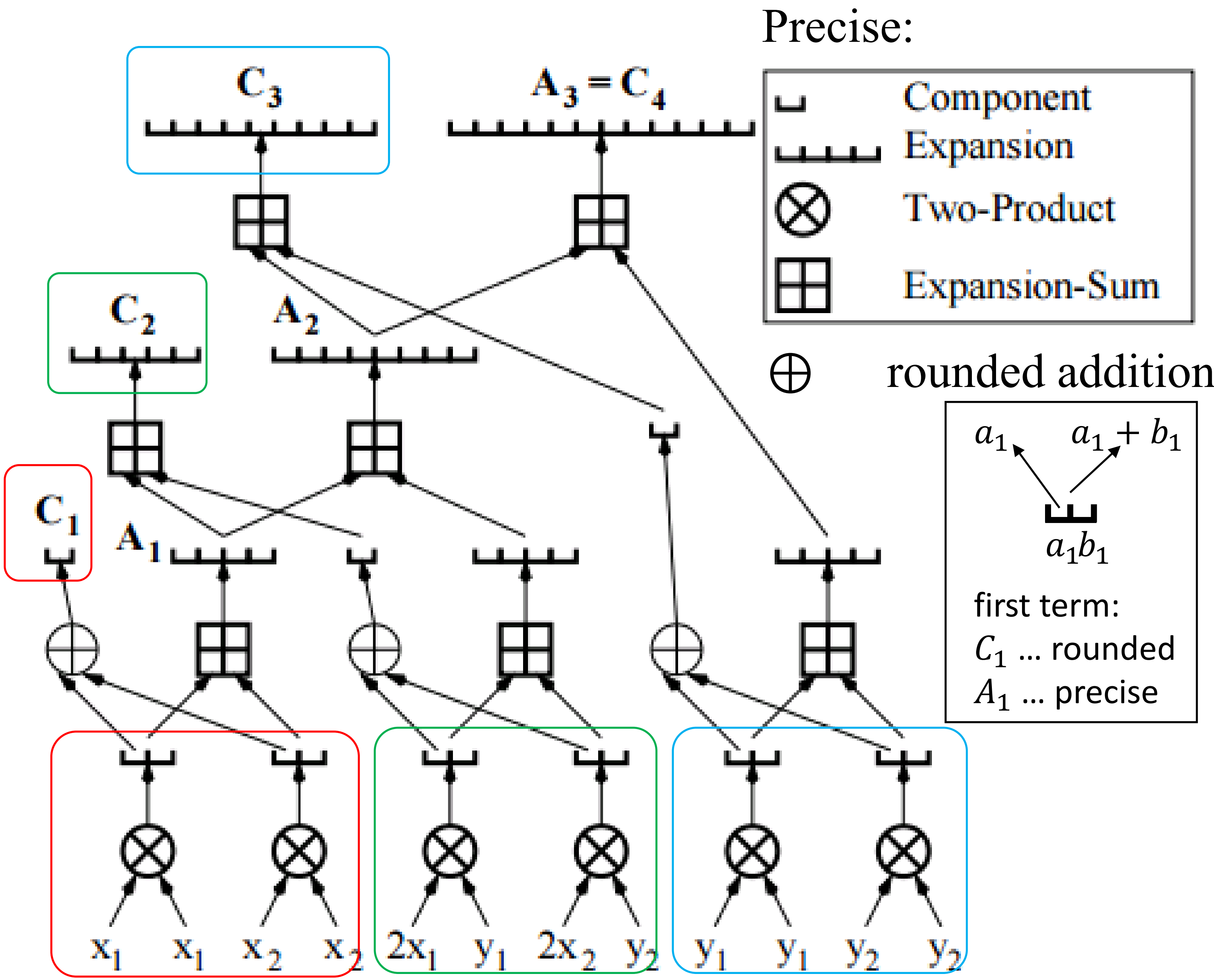
$$(x_1^2 + x_2^2) + (2x_1y_1 + 2x_2y_2) + (y_1^2 + y_2^2)$$

Compute them only if needed

$$(b_x - a_x)^2 + \left(b_y - a_y\right)^2$$



$$(x_1^2 + 2x_1y_1 + y_1^2) +$$
$$(x_2^2 + 2x_2y_2 + y_2^2)$$



pivot p

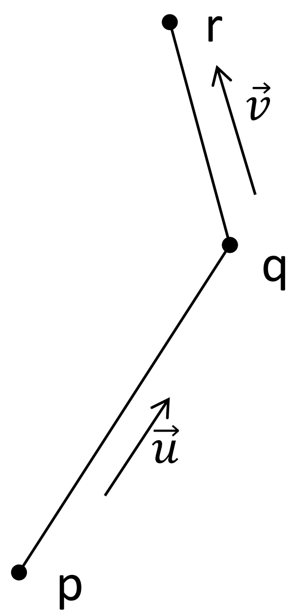
Orientation predicate - definition

$$\text{orientation}(p, q, r) = \text{sign}\left(\det\begin{bmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{bmatrix}\right) =$$

$$= \text{sign}\left((p_x - r_x)(q_y - r_y) - (p_y - r_y)(q_x - r_x)\right),$$

where point $p = (p_x, p_y), \ldots$

$$= \text{third coordinate of} = (\vec{u} \times \vec{v}),$$



Three points

- lie on common line
- form a left turn
- form a right turn

$\text{orientation}(p, q, r) =$

= 0
= +1 (positive)
= –1 (negative)

pivot r

Experiment with orientation predicate

- orientation(p,q,r) = sign($(p_x-r_x)(q_y-r_y)-(p_y-r_y)(q_x-r_x)$)

Ideal return values