Floating-Point Training
Module 3
Core Algorithms

Paul Zimmermann, Inria Nancy

Plan of the training

- Module 1: The IEEE 754 Standard
- Module 2: Math Fundamentals
- **Module 3: Core Algorithms**
- Module 4: Elementary Function Approximation
- Module 5: Software Tools

Module 3: Core Algorithms

Double-double arithmetic

TwoSum and FastTwoSum (also with directed rounding)

Veltkamp's splitting

Dekker's product with and without FMA

Horner's rule

Estrin's scheme

Computer games have floating-point bugs too!

Half-Life 2 computer game. Bug discovered in 2013, dated back in 2004.

Image credit: Valve.

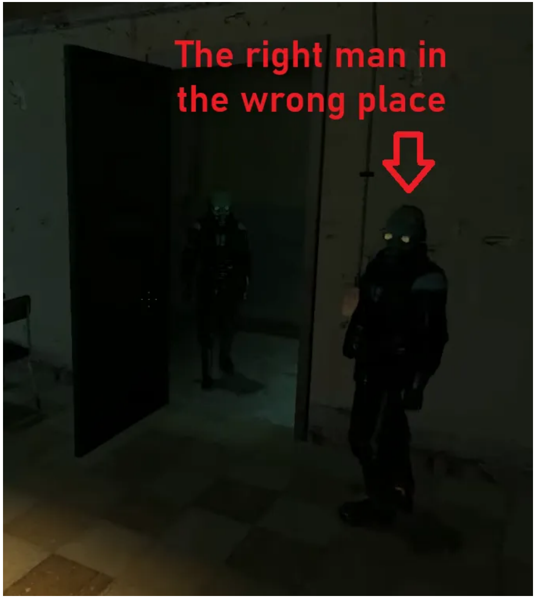

Image credit: Valve/Tom Forsyth

The guard is too close to the door

His toe collides with the door

The door cannot open

Players cannot access the room

Why did the bug not happen
back in 2004?

In 2004, floating-point was using x87

Now it uses SSE...

Which arithmetic for internal computations?

Internal computations need more bits than the target precision p, usually at least $p + 10$ or $p + 15$ bits.

- for target binary32, use binary64
- for target binary64, use `long double` or binary128 (slow)
- for target binary128, use integer-based arithmetic

Other solution: only use the target format, with double-double arithmetic.

Double-double arithmetic

Idea: represent a number x by a sum of two values $h + \ell$.

If both h and ℓ have p bits of precision, x may have up to $2p$ or $2p + 1$ bits of precision.

Example: π might be represented by:

```
double pi_hi = 0x1.921fb54442d18p+1;
double pi_lo = 0x1.1a62633145c07p-53;
```

with relative error about $2^{-109.7}$.

If h, ℓ are two binary32 numbers: double-float.

Pros:

- no need to implement arbitrary precision arithmetic
- can rely on fast operations on native floating-point types
- on GPU, using binary64 for target binary32 might be slow
- generic algorithms independent of the format of h, ℓ

Cons:

- need to normalize h, ℓ during the computations to keep full precision $2p$
- if no normalization, error analysis is tricky

We will now see how to perform arithmetic on double-doubles.

Double-double algorithms

TwoSum, FastTwoSum: $a + b \Longrightarrow h + \ell$

TwoMul: $a \cdot b \Longrightarrow h + \ell$

Double-double addition: $(a_h + a_\ell) + (b_h + b_\ell) \Longrightarrow h + \ell$

Multiplication by a double: $(a_h + a_\ell) \cdot b \Longrightarrow h + \ell$

Double-double multiplication: $(a_h + a_\ell) \cdot (b_h + b_\ell) \Longrightarrow h + \ell$

Normalizing a double-double: TwoSum

Algorithm TwoSum(a, b) (Møller 1965):

```
1. $s \leftarrow a \oplus b$
2. $a' \leftarrow s \ominus b$
3. $b' \leftarrow s \ominus a$
4. $\delta_a \leftarrow a \ominus a'$
5. $\delta_b \leftarrow b \ominus b'$
6. $t \leftarrow \delta_a \oplus \delta_b$
```

Lemma

If correct rounding to nearest (with ties resolved any way), no overflow nor underflow (or gradual underflow), then Algorithm TwoSum satisfies:

$$a + b = s + t.$$

TwoSum: Example

```
sage: a,b = RR(log(2)),RR(pi)
sage: s = a+b
sage: a_prime = s-b
sage: b_prime = s-a
sage: delta_a = a - a_prime
sage: delta_b = b - b_prime
sage: t = delta_a+delta_b
sage: s.hex(), t.hex()
('0x3.d5b1828057728p+0', '-0x8p-56')
sage: a.exact_rational() + b.exact_rational()
34540265776279119/9007199254740992
sage: s.exact_rational() + t.exact_rational()
34540265776279119/9007199254740992
```

Computing $x_1 + x_2 + \cdots + x_n$

Naive algorithm:

```
$s \leftarrow x_1$
for $i$ from 2 to $n$
    $s \leftarrow \circ(s + x_i)$
return $s$
```

Assuming all x_i are positive, the rounding error is bounded by $n \cdot \mathrm{ulp}(s)$.

Computing $x_1 + x_2 + \cdots + x_n$

Double-double algorithm:

```
$s \leftarrow x_1$
$e \leftarrow 0$
for $i$ from 2 to $n$
    $(s, e_i) \leftarrow \mathrm{TwoSum}(s, x_i)$
    $e \leftarrow \circ(e + e_i)$
return $\circ(s + e)$
```

Theorem (Ogita, Rump, Oishi)

Assume precision p with $u = 2^{-p}$. If no overflow, and $nu < 1$, then with $\gamma_n = nu/(1 - nu)$:

$$|s - \sum x_i| \leq u|\sum x_i| + \gamma_{n-1}^2 \sum |x_i|$$

Normalizing a double-double: FastTwoSum

Algorithm FastTwoSum(a, b) (Møller 1965):

```
1. $s \leftarrow a \oplus b$
2. $z \leftarrow s \ominus a$
3. $t \leftarrow b \ominus z$
```

Lemma

If $|a| \geq |b|$, correct rounding to nearest (with ties resolved any way), no overflow nor underflow (or gradual underflow), then Algorithm FastTwoSum satisfies:

$$a + b = s + t.$$

FastTwoSum: Example

```
sage: a,b = RR(pi),RR(log(2))
sage: a.hex(), b.hex()
('0x3.243f6a8885a3p+0', '0xb.17217f7d1cf78p-4')
sage: s = a+b
sage: z = s-a
sage: t = b-z
sage: s.hex(), t.hex()
('0x3.d5b1828057728p+0', '-0x8p-56')
sage: a.exact_rational() + b.exact_rational()
34540265776279119/9007199254740992
sage: s.exact_rational() + t.exact_rational()
34540265776279119/9007199254740992
```

FastTwoSum: example that fails

```
sage: a,b = RR(log(2)),RR(pi)
sage: a.hex(), b.hex()
('0xb.17217f7d1cf78p-4', '0x3.243f6a8885a3p+0')
sage: s = a+b
sage: z = s-a
sage: t = b-z
sage: s.hex(), t.hex()
('0x3.d5b1828057728p+0', '0x0p+0')
sage: a.exact_rational() + b.exact_rational()
34540265776279119/9007199254740992
sage: s.exact_rational() + t.exact_rational()
2158766611017445/562949953421312
```

The assumption $|a| \geq |b|$ is not satisfied.

FastTwoSum: another example that fails

```
sage: R = RealField(53,rnd='RNDU')
sage: a,b = R(pi*2^56),R(log(2))
sage: s = a+b; z = s-a; t = b-z
sage: s.hex(), t.hex()
('0x3.243f6a8885a34p+56', '-0x1.f4e8de8082e3p+4')
sage: a.exact_rational() + b.exact_rational()
127438138015862329074875318117279/562949953421312
sage: s.exact_rational() + t.exact_rational()
3982441812995697783589853691165/17592186044416
```

The rounding mode is not to nearest.
In this case $(a+b)-s$ is not representable: it has 54 bits.

```
sage: (a.exact_rational()+b.exact_rational())-s.exact_rational()
-17624191336471649/562949953421312
sage: 17624191336471649.nbits()
54
```

Variants of FastTwoSum

We consider in this training the following variant:

$$s = a \oplus b, \quad z = s \ominus a, \quad t = b \ominus z$$

A variant:

$$s = a \oplus b, \quad z = a \ominus s, \quad t = b \oplus z$$

Both variants return the same values for RN/RZ, but not for RU/RD!

Be careful which variant you use.

Sufficient exactness condition for RN

For rounding to nearest, the classical condition for exactness of FastTwoSum is $|a| \geq |b|$. Then we have $s + t = a + b$.

But it suffices to have $e_a \geq e_b$, where e_a denotes the exponent of a.

Or even that a is an integer multiple of $\mathrm{ulp}(b)$.

In all these cases, the 2nd and 3rd roundings can be any IEEE rounding.

Cf Theorem 1 of FastTwoSum revisited.

The second operation is exact

Algorithm FastTwoSum(a, b):

```
1. $s \leftarrow a \oplus b$
2. $z \leftarrow s \ominus a$
3. $t \leftarrow b \ominus z$
```

Lemma

When $a \in \mathrm{ulp}(b)\mathbb{Z}$ and the first rounding is any IEEE rounding, the second operation is exact: $z = s - a$.

As a consequence, we have:

$$t = \circ(a + b - s),$$

thus t is the rounding of the error in $s = a \oplus b$.

Cf Lemma 2 of FastTwoSum revisited.

FastTwoSum with directed rounding

Lemma

In precision p, if $a \in \mathrm{ulp}(b)\mathbb{Z}$ and $(s,t) = \mathrm{FastTwoSum}(a,b,\mathrm{RZ})$, then:

$$s + t = \begin{cases} a + b & \text{if } ab \geq 0 \text{ or } e_a - e_b \leq p \\ \mathrm{RZ}_{2p}(a+b) & \text{otherwise} \end{cases}$$

Cf Theorem 9 of FastTwoSum revisited.

Similar theorems for RU/RD.

Double $\times$ double $\to$ double-double

- without FMA: the hard way

First split x into $x_h + x_\ell$, same for y (Veltkamp's splitting)

Then multiply $x_h y_h$, $x_h y_\ell$, $x_\ell y_h$, $x_\ell y_\ell$ exactly, and accumulate the products (Dekker's product)

- with FMA: the easy way

Veltkamp's splitting

Algorithm VeltkampSplit(x, s):

```
1. $C \leftarrow 2^s + 1$
2. $\gamma \leftarrow RN(Cx)$
3. $\delta \leftarrow RN(x - \gamma)$
4. $x_h \leftarrow RN(\gamma + \delta)$
5. $x_\ell \leftarrow RN(x - x_h)$
```

Lemma

If Cx does not overflow, Algorithm VeltkampSplit in precision p satisfies:

$$x = x_h + x_\ell$$

where x_h fits on $p - s$ bits, and x_ℓ fits on s bits ($s - 1$ if $s \geq 2$).

Boldo extended the lemma to any radix β.

Veltkamp’s splitting: example

```
sage: x = RR(pi)
sage: C = RR(2^27+1)
sage: gamma = C * x
sage: delta = x - gamma
sage: xh = gamma + delta
sage: xl = x - xh
sage: xh.hex(), xl.hex()
('0x3.243f6bp+0', '-0x7.77a5dp-28')
sage: x.exact_rational()
884279719003555/281474976710656
sage: xh.exact_rational() + xl.exact_rational()
884279719003555/281474976710656
```

This is a relative splitting. With $C = 2^{27} + 1$, both x_h and x_ℓ fit on 26 bits.

Dekker's product (1971)

Algorithm DekkerProduct(x, y):

```
1. $x_h, x_\ell \leftarrow \mathrm{VeltkampSplit}(x, \lceil p/2 \rceil)$
2. $y_h, y_\ell \leftarrow \mathrm{VeltkampSplit}(y, \lceil p/2 \rceil)$
3. $h \leftarrow RN(xy)$
4. $u \leftarrow RN(x_h y_h - h)$        [$x_h y_h$ is exact]
5. $v \leftarrow RN(x_h y_\ell + u)$     [$x_h y_\ell$ is exact]
6. $w \leftarrow RN(x_\ell y_h + v)$     [$x_\ell y_h$ is exact]
7. $\ell \leftarrow RN(x_\ell y_\ell + w)$  [$x_\ell y_\ell$ is exact]
```

Lemma

Assuming no overflow nor underflow, the values h, ℓ returned by Algorithm DekkerProduct satisfy:

$$xy = h + \ell.$$

Dekker's product

Algorithm DekkerProduct(x, y):

```
1. $x_h, x_\ell \leftarrow \mathrm{VeltkampSplit}(x, \lceil p/2 \rceil)$
2. $y_h, y_\ell \leftarrow \mathrm{VeltkampSplit}(y, \lceil p/2 \rceil)$
3. $h \leftarrow RN(xy)$
4. $u \leftarrow RN(x_h y_h - h)$
5. $v \leftarrow RN(x_h y_\ell + u)$
6. $w \leftarrow RN(x_\ell y_h + v)$
7. $\ell \leftarrow RN(x_\ell y_\ell + w)$
```

Each VeltkampSplit requires 4 operations, thus DekkerProduct requires 17 operations (some might be performed in parallel).

$xy \to h + \ell$ with FMA

Algorithm TwoMul(x, y):

```
1. $h \leftarrow \circ(xy)$
2. $\ell \leftarrow \mathrm{fma}(x, y, -h) = \circ(xy - h)$
```

Lemma

Assuming no overflow nor underflow, the values h, ℓ returned by Algorithm TwoMul satisfy:

$$xy = h + \ell,$$

whatever the rounding mode.

Uses 2 floating-point operations instead of 17!

$xy \to h + \ell$ with FMA (proof 1/2)

Algorithm TwoMul(x, y):

```
1. $h \leftarrow \circ(xy)$
2. $\ell \leftarrow FMA(x, y, -h) = \circ(xy - h)$
```

Proof: Assume without loss of generality that $x, y > 0$.

xy is exactly representable on $2p$ bits, it can be written $u + v$ where u corresponds to the p most significant bits, and v to the lower bits, with $\mathrm{ulp}(v) = 2^{-p}\mathrm{ulp}(u)$ (v is not necessarily in normal form).

Case (1): if $h = u$, then since $xy - h = xy - u = v$ is exactly representable, we necessarily have $\ell = v$.

$xy \to h + \ell$ with FMA (proof 2/2)

Algorithm TwoMul(x, y):

1. $h \leftarrow \circ(xy)$
2. $\ell \leftarrow FMA(x, y, -h) = \circ(xy - h)$

We have $xy = u + v$, where u corresponds to the p most significant bits, and v to the lower bits, with $\mathrm{ulp}(v) = 2^{-p}\mathrm{ulp}(u)$.

Case (2): if $h > u$, then $h = u + \mathrm{ulp}(u)$, and

$$xy - h = xy - (u + \mathrm{ulp}(u)) = v - \mathrm{ulp}(u).$$

Since $0 \leq v < 2^p\mathrm{ulp}(v) = \mathrm{ulp}(u)$, we have:

$$-2^p\mathrm{ulp}(v) \leq v - \mathrm{ulp}(u) < 2^p\mathrm{ulp}(v)$$

and $v - \mathrm{ulp}(u)$ is an integer multiple of $\mathrm{ulp}(v)$, thus $v - \mathrm{ulp}(u)$ is exactly representable.

TwoSum and TwoMul in IEEE 754

IEEE 754-2019 recommends augmentedAddition(x, y), which returns h, ℓ where $h = \mathrm{RoundTiesTowardZero}(x + y)$, and $\ell = x + y - h$.

IEEE 754-2019 recommends augmentedMultiplication(x, y), which returns h, ℓ where $h = \mathrm{RoundTiesTowardZero}(xy)$, and $\ell = xy - h$.

The rounding mode of both operations is always RoundTiesTowardZero, whatever the current rounding mode.

If/when both operations are available in hardware, there will be a major speed up for double-double arithmetic.

TwoSum and TwoMul in IEEE 754

Why RoundTiesTowardZero?

Consider $x = 786429 \cdot 2^{485}$ and $y = 22906579627 \cdot 2^{485}$, then $xy = 2^{1024} \cdot (1 - 2^{-54})$ is exactly in the middle of `DBL_MAX` and 2^{1024}.

With round to nearest even, we would get overflow for h.

With RoundTiesTowardZero we get $h = 2^{1024} \cdot (1 - 2^{-53})$ and $\ell = 2^{970}$.

Double-double addition

Algorithm DDadd:

```
1. Input: $a_h + a_\ell$ and $b_h + b_\ell$
2. $h_1, \ell_1 \leftarrow \mathrm{TwoSum}(a_h, b_h)$
3. $\ell_2, \nu_2 \leftarrow \mathrm{TwoSum}(a_\ell, b_\ell)$
4. $h_3, \ell_3 \leftarrow \mathrm{FastTwoSum}(h_1, \ell_2)$
5. $\ell_4 \leftarrow \ell_1 \oplus \nu_2$
6. $\ell_5 \leftarrow \ell_4 \oplus \ell_3$
7. return $h, \ell = \mathrm{FastTwoSum}(h_3, \ell_5)$
```

Lemma

Assuming rounding to nearest, $|a_\ell| \le \frac{1}{2}\mathrm{ulp}(a_h)$ *and* $|b_\ell| \le \frac{1}{2}\mathrm{ulp}(b_h)$*, we have* $|\ell| \le \frac{1}{2}\mathrm{ulp}(h)$*, and* $|h + \ell - (a + b)| \le 2^{-(2p-1)}|a + b|$*.*

Reference: Zhang and Aiken, 2025 (Figure 2).

Double-double addition

If you know $|a_h| \geq |b_h|$ (for example in Horner's evaluation) and normalization of the result is not needed, a faster variant:

Algorithm DDaddFast:

```
1. Input: $a_h + a_\ell$ and $b_h + b_\ell$
2. $h, \ell_1 \leftarrow \mathrm{FastTwoSum}(a_h, b_h)$
3. $\ell \leftarrow \ell_1 \oplus (a_\ell \oplus b_\ell)$
```

Double-double multiplication

Algorithm DDmul:

```
1. Input: $a_h + a_\ell$ and $b_h + b_\ell$
2. $h_1, \ell_1 \leftarrow \mathrm{TwoMul}(a_h, b_h)$
3. $\ell_2 \leftarrow a_h \otimes b_\ell$
4. $\ell_3 \leftarrow a_\ell \otimes b_h$
5. $\ell_4 \leftarrow \ell_2 \oplus \ell_3$
6. $\ell_5 \leftarrow \ell_1 \oplus \ell_4$
7. return $h, \ell = \mathrm{FastTwoSum}(h_1, \ell_5)$
```

Lemma

Assuming rounding to nearest, $|a_\ell| \leq \frac{1}{2}\mathrm{ulp}(a_h)$ *and* $|b_\ell| \leq \frac{1}{2}\mathrm{ulp}(b_h)$*, we have* $|\ell| \leq \frac{1}{2}\mathrm{ulp}(h)$*, and* $|h + \ell - ab| \leq 2^{-(2p-3)}|ab|$.

Reference: Zhang and Aiken, 2025 (Figure 5).

Double-double multiplication

Algorithm DDmulFast:

```
1. Input: $a_h + a_\ell$ and $b_h + b_\ell$
2. $h, \ell_1 \leftarrow \mathrm{TwoMul}(a_h, b_h)$
3. $\ell_2 \leftarrow \circ(\mathrm{fma}(a_h, b_\ell, \ell_1))$
4. $\ell \leftarrow \circ(\mathrm{fma}(a_\ell, b_h, \ell_2))$
```

Avoids the final FastTwoSum call, replaces additions and multiplications by FMAs.

Multiplication of a double-double by a double

Algorithm DDmulD:

```
1. Input: $a_h + a_\ell$ and $b$
2. $h, \ell_1 \leftarrow \mathrm{TwoMul}(a_h, b)$
3. $\ell \leftarrow \circ(\mathrm{fma}(a_\ell, b, \ell_1))$
```

Application: multiplication by a constant

Assume we want to multiply a double x by a constant C, say $C = 1/\log 2$ (useful for range reduction, see Module 4).

Precompute $C_h = \mathrm{RN}(C)$:

$y_h \leftarrow \mathrm{RN}(C_h x)$

We get only 53 bits of accuracy in y_h.

Application: multiplication by a constant

Precompute $C_h = \mathrm{RN}(C)$ and $C_\ell = \mathrm{RN}(C - C_h)$:

$y_h, y_\ell \leftarrow \mathrm{DDmulD}(C_h, C_\ell, x)$

We get an accuracy of about 106 bits in $y_h + y_\ell$.

Example: for $C = 1/\log 2$, we have $C_h =$ `0x1.71547652b82fep+0` and $C_\ell =$ `0x1.777d0ffda0d24p-56`.

Floating-point expansions

A floating-point expansion is an unevaluated sum:

$$x_1 + x_2 + \ldots + x_n$$

where x_{i+1} is smaller than x_i in absolute value.

Enables to emulate a precision of np bits.

Full normalization (to nearest): $|x_{i+1}| \leq \frac{1}{2}\mathrm{ulp}(x_i)$.

Lazy normalization: $|x_{i+1}| \leq \mathrm{ulp}(x_i)$.

Other more lazy normalizations are possible (normalizing at each operation might be expensive).

Triple-double and beyond

If double-double is not enough, we can use "triple-double" (about 160 bits for binary64):

• Algorithms for triple-word arithmetic, Nicolas Fabiano, Jean-Michel Muller and Joris Picot, IEEE Transactions on Computers, 2019.

And beyond:

• High-Performance Branch-Free Algorithms for Extended-Precision Floating-Point Arithmetic, David K. Zhang and Alex Aiken, Supercomputing 2025.

They present conjecturally optimal floating-point accumulation networks (FPANs) for addition/multiplication of 2-term, 3-term and 4-term expansions.

Horner's rule

To evaluate a polynomial $a_0 + a_1 x + a_2 x^2 + a_3 x^3$:

Naive way:

```
double m1 = a1 * x;
double x2 = x * x, m2 = a2 * x2;
double x3 = x2 * x, m3 = a3 * x3;
return (a0 + m1) + (m2 + m3);
```

Horner's rule

To evaluate a polynomial $a_0 + a_1 x + a_2 x^2 + a_3 x^3$:

Horner's rule:

```
double t2 = a2 + x * a3;
double t1 = a1 + x * t2;
return a0 + x * t1;
```

Simpler and uses less operations.

In addition, usually x is small: better numerical behaviour.

Note: attributed to Lagrange by Horner, and possibly older.

Horner's rule with FMA

Horner's rule without FMA for $a_0 + a_1 x + a_2 x^2 + a_3 x^3$:

```
double t2 = a2 + x * a3;
double t1 = a1 + x * t2;
return a0 + x * t1;
```

Horner's rule with FMA:

```
double t2 = fma (x, a3, a2);
double t1 = fma (x, t2, a1);
return fma (x, t1, a0);
```

Only 3 floating-point operations!

Estrin's scheme (1960)

Useful to improve the parallelism (reduce the depth of the evaluation tree).

Example: evaluate the polynomial $a_0 + a_1 x + a_2 x^2 + \cdots + a_7 x^7$

Horner's rule:

```
double t6 = a6 + x * a7;
double t5 = a5 + x * t6;
double t4 = a4 + x * t5;
double t3 = a3 + x * t4;
double t2 = a2 + x * t3;
double t1 = a1 + x * t2;
double y  = a0 + x * t1;
```

The evaluation depth is 7 since each operation depends on the previous one: t_5 cannot be computed before t_6 is obtained, t_4 cannot be computed before t_5 is obtained, ...

Estrin's scheme

Estrin's scheme:

```
double t01 = a0 + x * a1, t23 = a2 + x * a3,
       t45 = a4 + x * x5, t67 = a6 + x * a7, x2 = x * x;
double t03 = t01 + x2 * t23, t47 = t45 + x2 * t67, x4 = x2 * x2;
double y = t03 + x4 * t47;
```

The evaluation depth is reduced to 3: `t01, t23, t45, t67, x2` can be computed in parallel at the first cycle, then `t03, t47, x4` can be computed in parallel at the 2nd cycle, then `y` is obtained at the 3rd cycle.

Two more operations (`x2` and `x4`), but gain of 4 in depth.

Exercises

Exercise 1: Implement algorithm TwoSum, run it on random binary64 inputs, and check you always get $a + b = s + t$. Do this still hold for directed roundings?

Exercise 2: Implement FastTwoSum and the variant that computes $z = a \ominus s$ then $t = b \oplus z$, and compare them with directed roundings, when the exponent of b varies from e_a to $e_a - 2p$. Which one is better?

Exercise 3: Implement algorithm DekkerProduct, run it on random binary64 inputs, and check you always get $x \cdot y = h + \ell$. Do this still hold for directed roundings?

Exercise 4: Construct an example of augmentedAddition that would overflow with round to nearest even, and that does not overflow with RoundTiesTowardZero.

Exercise 5: Implement algorithm DDaddFast, and run it on random inputs satisfying condition $|a_\ell| \leq \frac{1}{2}\mathrm{ulp}(a_h)$ and $|b_\ell| \leq \frac{1}{2}\mathrm{ulp}(b_h)$. What is the largest error you get? Compare it to the bound $2^{-(2p-1)}|a+b|$ of algorithm DDadd.

Exercise 6: Same question as Exercise 5, but comparing algorithms DDmul and DDmulFast.

Takeover message

• double-double arithmetic provides about 106 bits of accuracy for binary64 (48 bits for binary32)

• this will be plenty enough for the fast path of mathematical functions, and almost enough for the accurate path (see Module 4)

• basic routines (Fast)TwoSum and TwoMul are very efficient on modern processors

• they could even be more efficient if TwoSum and TwoMul were implemented in hardware (augmentedAddition and augmentedMultiplication from IEEE 754-2019)

References

Handbook of Floating-point Arithmetic, Jean-Michel et al., 2nd edition, Birkhäuser, 2018.

FastTwoSum revisited, Claude-Pierre Jeannerod and Paul Zimmermann, Proceedings of the 32nd IEEE International Symposium on Computer Arithmetic (ARITH), 2025.

High-Performance Branch-Free Algorithms for Extended-Precision Floating-Point Arithmetic, David K. Zhang and Alex Aiken, SuperComputing 2025, `https://theory.stanford.edu/~aiken/publications/papers/sc25.pdf`.

Questions, comments ?

`Paul.Zimmermann@inria.fr`