Accurate calculation of Euclidean Norms using Double-word arithmetic

Vincent Lefèvre[b] Nicolas Louvet[d] Jean-Michel Muller[a]
Joris Picot[c] Laurence Rideau[b]

[a]CNRS, [b]INRIA, [c]ENS de Lyon, [d]Univ. Lyon 1

Computation of Euclidean norms

- Euclidean (a.k.a. L2) norm of the vector $(a_0, a_1, a_2, \ldots, a_{n-1})$

$$N = \sqrt{\sum_{i=0}^{n-1} a_i^2}$$

  in radix-2, precision-p Floating-Point arithmetic;

- we assume $n \geq 3$ (and rather large);
- Goals:
  - avoid spurious overflows and underflows;
  - very accurate results (error bound very slightly above 0.5ulp);
  - formally proven behavior (work in progress);
- we start from an algorithm due to Graillat, Lauter, Tang, Yamanaka and Oishi (ACM TOMS – 2015).

Spurious under/overflows can jeopardize the computation

- IEEE 754 binary64 arithmetic with $n = 3$, and the round-to-nearest, ties-to-even, rounding function;
- "straightforward" solution (first sum the squares serially, then take the square root):
  - with $a_0 = 1.5 \times 2^{511}$, $a_1 = 0$, and $a_2 = 2^{512}$, we obtain $+\infty$, whereas the exact result is 5×2^{510};
  - with $a_0 = a_1 = a_2 = (45/64) \times 2^{-537}$, the computed result is 0, whereas the exact result is around 1.2178×2^{-537}.
- From an accuracy point-of-view, spurious underflow is a problem only if all terms a_i are tiny.
- However, itcan be very harmful from a performance point-of-view when subnormal numbers are handled in software.

Avoiding spurious overflow and harmful underflow

$$C \cdot \sum_{i=0}^{n-1} a_i^2$$

- possible scaling factor C

naive method

- find $m = \max |a_i|$
- compute $\sum_{i=0}^{n-1} \left(\frac{a_i}{m}\right)^2$

variant: scale only if m larger than m_{big}
or less than m_{small}

Hammarling

- before step j, $m = \max_{i<j} |a_i|$ and $s = \sum_{i<j} \left(\frac{a_i}{m}\right)^2$
- if $|a_j| \leq m$ then $s \leftarrow s + \left(\frac{a_j}{m}\right)^2$
- else $s \leftarrow s \cdot \left(\frac{m}{a_j}\right)^2 + 1$ and $m \leftarrow |a_j|$

variant: scale only if $m \geq m_{big}$ or $m \leq m_{small}$

Blue

- 3 classes of FPNs: TINY, MED, and BIG
- 2 constants: t_{tiny} and t_{big}, powers of 2
- Separately compute:

$$S_{\text{tiny}} = \sum_{a_i \in \text{TINY}} (t_{\text{tiny}} \cdot a_i)^2$$
$$S_{\text{med}} = \sum_{a_i \in \text{MED}} a_i^2$$
$$S_{\text{big}} = \sum_{a_i \in \text{BIG}} (t_{\text{big}} \cdot a_i)^2$$

variant: BIG nonempty $\rightarrow$ no need to consider TINY

A few landmarks

- Blue (1978): robust yet complex algorithm (with final operations that are no longer necessary on IEEE-754-compliant systems);
- Lawson et al. (1979): variant of Hammarling's algorithm where scaling only if $m \geq m_{big}$ or $m \leq m_{small}$;
- Hammarling's algorithm included in Lapack (1987);
- Kahan (1997 ?): compensated summation of the sums of squares (but each square represented by 1 FPN only), scaling by constant factors when needed, correct handling of IEEE-754 exceptions;
- Graillat et al. (2015): variant of Blue's algorithm with double-word arithmetic, that targets faithful rounding;
- Hanson & Hopkins (2017): hybrid method (compensated summation—suffices for most common cases, then Kahan's algorithm if an exception occurred);

Underlying FP arithmetic

- Radix-2, precision-p (with $p \geq 5$), FP arithmetic,
- extremal exponents $e_{\min}$ and $e_{\max} = 1 - e_{\min}$
- $\mathrm{RN}(t)$ stands for t rounded to nearest FP number.

Table 1: Notation for the important FP parameters

Notation	numerical value	explanation
Ω	$2^{e_{\max}} \cdot (2 - 2^{-p+1})$	largest finite FPN
α	$2^{e_{\min}-p+1}$	smallest positive FPN
η	$2^{(e_{\min}+p)/2}$	the square of a FPN $\geq \eta$ is the sum of two FPNs
u	2^{-p}	roundoff error unit
$\mathrm{ulp}(x)$	$2^{\max\{\lfloor \log_2 \lvert x \rvert \rfloor, e_{\min}\}-p+1}$	unit in the last place

Blue's algorithm: classes MED, BIG and TINY

- MED Class: FPN that can be squared, and whose squares can be accumulated (up to $n_{\max}$ terms), without under/overflows.
  $a_i \in$ MED if minmed $\leq |a_i| \leq$ maxmed. We compute

$$S_{\mathrm{med}} = \sum_{a_i \in \mathrm{MED}} a_i^2;$$

- $a_i \in$ BIG if maxmed $< |a_i|$. All FPN $\in$ BIG are "scaled down", i.e., multiplied by *the same* constant t_{big}. We compute

$$S_{\mathrm{big}} = \sum_{a_i \in \mathrm{BIG}} \left(t_{\mathrm{big}} \cdot a_i\right)^2.$$

- $a_i \in$ TINY if $|a_i| <$ minmed. All FPN $\in$ TINY are "scaled up", i.e., multiplied by *the same* constant t_{tiny}. We compute

$$S_{\mathrm{tiny}} = \sum_{a_i \in \mathrm{TINY}} \left(t_{\mathrm{tiny}} \cdot a_i\right)^2.$$

- we need $t_{\mathrm{big}} \times \mathrm{BIG} \subseteq \mathrm{MED}$ and $t_{\mathrm{tiny}} \times \mathrm{TINY} \subseteq \mathrm{MED}$.

One needs to cleverly combine S_{big}, S_{med}, and S_{tiny}.

Parameters

$$\mathsf{minmed}^2 \geq 2^{e_{\min}}, \tag{1a}$$

$$n_{\max} \cdot \mathsf{maxmed}^2 \cdot (1+\rho) < \Omega + \frac{1}{2}\mathrm{ulp}(\Omega) = 2^{e_{\max}+1} - 2^{e_{\max}-p}, \tag{1b}$$

$$\mathsf{maxmed} \cdot t_{\mathrm{big}} \geq \mathsf{minmed}, \tag{1c}$$

$$\Omega \cdot t_{\mathrm{big}} \leq \mathsf{maxmed}, \tag{1d}$$

$$\mathsf{minmed} \cdot t_{\mathrm{tiny}} \leq \mathsf{maxmed}, \tag{1e}$$

$$\alpha \cdot t_{\mathrm{tiny}} \geq \mathsf{minmed}, \tag{1f}$$

where

- Ω and α are the largest and smallest positive FPNs;
- ρ is a bound on the relative error of the algorithm used for computing the sum of squares in MED;
- If maxmed and $n_{\max}$ are powers of 2, (1b) can be replaced by

$$n_{\max} \cdot \mathsf{maxmed}^2 < 2^{e_{\max}+1} - 2^{e_{\max}-p}.$$

Basic building blocks

Algorithm 1 – Fast2Sum(a, b) (Dekker, 1971)

$s \leftarrow \mathrm{RN}(a+b)$
$z \leftarrow \mathrm{RN}(s-a)$
$t \leftarrow \mathrm{RN}(b-z)$

If $|a| \geq |b|$ and no overflow occurs, $s + t = a + b$, i.e., t is the error of the FP addition of a and b.

Algorithm 2 – 2Sum(a, b) (Knuth). It takes 6 FP operations.

$s \leftarrow \mathrm{RN}(a+b)$
$a' \leftarrow \mathrm{RN}(s-b)$
$b' \leftarrow \mathrm{RN}(s-a')$
$\delta_a \leftarrow \mathrm{RN}(a-a')$
$\delta_b \leftarrow \mathrm{RN}(b-b')$
$t \leftarrow \mathrm{RN}(\delta_a+\delta_b)$

Same as Algorithm 1 without any condition on a and b.

Basic building blocks

Algorithm 3 – Fast2Mult(a, b)**.** Requires the availability of an FMA instruction.

$\pi_h \leftarrow \mathrm{RN}(a \cdot b)$
$\pi_\ell \leftarrow \mathrm{RN}(a \cdot b - \pi_h)$

- used for expressing the square of a FP number a as a double-word number;
- barring overflow, the condition for that algorithm to guarantee that $\pi_h + \pi_\ell = a^2$ is

$$|a| \geq \eta = 2^{(e_{\min}+p)/2}. \tag{2}$$

When the augmented operations specified by IEEE 754-2019 become efficiently implemented, 2Sum, Fast2Sum and Fast2Mult may be replaced by them.

Double Word Arithmetic

Also called "double-double" in the literature. Goes back to Dekker (1971).
Double-word (DW) number x: unevaluated sum $x_h + x_\ell$ of two FPN x_h and x_ℓ such that

$$x_h = \mathrm{RN}(x).$$

Algorithm 4 – DWPlusFP(x_h, x_ℓ, y).Computes $(x_h, x_\ell) + y$ (implemented in the QD library), $x = (x_h, x_\ell)$ is a DWN and y is a FPN.

1: $(s_h, s_\ell) \leftarrow \mathrm{2Sum}(x_h, y)$
2: $v \leftarrow \mathrm{RN}(x_\ell + s_\ell)$
3: $(z_h, z_\ell) \leftarrow \mathrm{Fast2Sum}(s_h, v)$
4: **return** (z_h, z_ℓ)

- In general, asymptotically optimal relative error bound

$$\frac{2 \cdot u^2}{1 - 2u} = 2u^2 + 4u^3 + 8u^4 + \cdots$$

- if x and y are nonnegative, the bound becomes u^2.

Double Word Arithmetic

Algorithm 5 – SloppyDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$. Computes $(x_h, x_\ell) + (y_h, y_\ell)$.

1: $(s_h, s_\ell) \leftarrow \text{2Sum}(x_h, y_h)$
2: $v \leftarrow \text{RN}(x_\ell + y_\ell)$
3: $w \leftarrow \text{RN}(s_\ell + v)$
4: $(z_h, z_\ell) \leftarrow \text{Fast2Sum}(s_h, w)$
5: **return** (z_h, z_ℓ)

- can be very inaccurate in the general case;
- when the inputs operands x_h and y_h have the same sign, asymptotically optimal relative error bound $3u^2$.

Graillat et al. (2015)

- Goal: faithful rounding, and no spurious under/overflow;
- handling of scalings: essentially Blue's method, adapted;
- sum-of-squares in Double-Word arithmetic, using Algorithm SloppyDWPlusDW, final result FP;
- then, FP square root.

Note: Kahan (1997) uses compensated summation of the sum of squares to obtain accuracy of the same order of magnitude (but not enough to guarantee faithful rounding).

Suggested modifications

- more accurate DW sum-of-squares using Algorithm DWPlusFP (and our new bound u^2 for positive operands), final result DW;
- ad-hoc algorithm SQRTDWtoFP: avoids the loss of information due to converting to FP before square root;
- slightly different choices of parameters (minmed, maxmed...)
- Goal: error bound very slightly above 0.5ulp.

Double-Word to FP square root

Algorithm 6 – SQRTDWtoFP(x_h, x_ℓ). Computes the square-root of the DW number (x_h, x_ℓ) and returns a FPN z.

1: $s_h \leftarrow \mathrm{RN}(\sqrt{x_h})$
2: $\rho_1 \leftarrow \mathrm{RN}(x_h - s_h^2)$ (with an FMA instruction)
3: $\rho_2 \leftarrow \mathrm{RN}(x_\ell + \rho_1)$
4: $s_\ell \leftarrow \mathrm{RN}(\rho_2/(2 \cdot s_h))$
5: $z \leftarrow \mathrm{RN}(s_h + s_\ell)$
6: **return** z

Theorem 1

If $x = (x_h, x_\ell)$ is a DW number, $p \geq 5$, $x \geq 2^{2\lceil (e_{\min}+p)/2 \rceil}$, then z is within

$$\left(\frac{1}{2} + \frac{7}{4} \cdot 2^{-p}\right) \cdot \mathrm{ulp}(\sqrt{x_h + x_\ell})$$

from $\sqrt{x_h + x_\ell}$, and the relative error of that algorithm is bounded by $u + \frac{17}{8}u^2 + \frac{33}{8}u^3$.

Theorem 1 has been formally proven using the Coq proof assistant.

Sequential DW computation of the sum of squares

Algorithm 7 Sequential computation of $\sum_{i=0}^{n-1} a_i^2$ assuming no under/overflow.

1. For $i = 0 \dots n-1$, compute $(y_i^h, y_i^\ell) = \mathrm{Fast2Mult}(a_i, a_i)$.
   (gives $a_i^2 = y_i^h + y_i^\ell$).
2. Accumulate the terms y_i^h in DW arithmetic: starting from $(x_1^h, x_1^\ell) = \mathrm{2Sum}(y_0^h, y_1^h)$, for $i = 2 \dots n-1$, compute $(x_i^h, x_i^\ell) = \mathrm{DWPlusFP}(x_{i-1}^h, x_{i-1}^\ell, y_i^h)$.
3. Accumulate the terms y_i^ℓ in FP arithmetic: for $i = 0 \dots n-2$, compute $\sigma_{i+1} = \mathrm{RN}(\sigma_i + y_{i+1}^\ell)$, with $\sigma_0 = y_0^\ell$.
4. Obtain the approximation to (S_h, S_ℓ) to $\sum_{i=0}^{n-1} a_i^2$ as

$$(S_h, S_\ell) = \mathrm{DWPlusFP}(x_{n-1}^h, x_{n-1}^\ell, \sigma_{n-1}).$$

Blockwise DW computation of the sum of squares

- the a_i are separated into k blocks of m numbers, with $n = k \times m$;
- parallelizing the calculation & obtaining a more accurate result;
- block j ($j = 0, \ldots, k-1$) contains $a_{mj}, a_{mj+1}, \ldots, a_{m(j+1)-1}$.

Algorithm 8 Blockwise computation of $\sum_{i=0}^{n-1} a_i^2$ assuming no under/overflow.

1. for $j = 0, 1, \ldots, k-1$, compute an approximation (Z_j^h, Z_j^ℓ) to $\sum_{i=mj}^{m(j+1)-1} a_i^2$ using the sequential summation algorithm applied to $a_{mj}, a_{mj+1}, a_{mj+2}, \ldots, a_{m(j+1)-1}$;
2. accumulate the terms Z_j^h in DW arithmetic, i.e., starting from $(\Sigma_1^h, \Sigma_1^\ell) = \mathrm{2Sum}(Z_0^h, Z_1^h)$, iteratively compute, for $j = 2 \ldots k-1$ the terms $(\Sigma_j^h, \Sigma_j^\ell) = \mathrm{DWPlusFP}\left(\Sigma_{j-1}^h, \Sigma_{j-1}^\ell, Z_j^h\right)$;
3. accumulate the terms Z_j^ℓ using the conventional "recursive" summation, i.e., for $j = 0 \ldots k-2$, compute $\tau_{j+1} = \mathrm{RN}(\tau_j + Z_{j+1}^\ell)$, with $\tau_0 = Z_0^\ell$;
4. obtain the approximation (S_h, S_ℓ) to $\sum_{i=0}^{n-1} a_i^2$ as

$$(S_h, S_\ell) = \mathrm{DWPlusFP}\left(\Sigma_{k-1}^h, \Sigma_{k-1}^\ell, \tau_{k-1}\right).$$

Computing Euclidean norms barring underflow/overflow

Theorem 2

Assume all $a_i \in \mathrm{MED}$, the sequential or blockwise summation (with k blocks of m elements) is used to compute (S_h, S_ℓ) and Algorithm SQRTDWtoFP is used to approximate $\sqrt{S_h + S_\ell}$ by a FP number R. Let

$$\lambda(t) = (2t-1) + (t-1)u + (2t-2)u^2 + (t-1)u^3,$$

and define

$$\nu = \begin{cases} \lambda(n) & \textit{with the sequential summation} \\ \lambda(k) + \lambda(m) + \lambda(k)\lambda(m) & \textit{with the blockwise summation} \end{cases}$$

If $\nu < \frac{1}{2u}$, then:

$$\left| R - \sqrt{\sum_{i=0}^{n-1} a_i^2} \right| \leq \left(\frac{1}{2} + u \cdot \left(\frac{7}{4} + \frac{\nu}{1 - \nu \cdot u^2} \right) \right) \mathrm{ulp}\left(\sqrt{\sum_{i=0}^{n-1} a_i^2} \right). \qquad (3)$$

Computing Euclidean norms in the general case

Parameters:

- $n_{\mathsf{max}} = 1/u = 2^p$, i.e., we wish to guarantee a correct behavior of the algorithms for vectors of dimension up to 2^p;
- MED as large as possible:
  - → minmed is the power of 2 just above η (so that the squares of the elements of MED are computed without error), i.e.,

$$\mathsf{minmed} = 2^{\lceil (e_{\mathsf{min}} + p)/2 \rceil};$$

  - → maxmed is the power of 2 just below $\sqrt{\Omega/2^p}$, i.e.,

$$\mathsf{maxmed} = 2^{\lfloor (e_{\mathsf{max}} - p)/2 \rfloor};$$

- $t_{\mathsf{tiny}} = 1/t_{\mathsf{big}}$ is an even power of 2 (so that multiplying/dividing by it and its square root is errorless);
- we need $t_{\mathsf{big}} \times \mathsf{BIG} \subseteq \mathsf{MED}$ and $t_{\mathsf{tiny}} \times \mathsf{TINY} \subseteq \mathsf{MED}$.

Computing Euclidean norms in the general case

Table 2: The various parameters of our algorithm for the binary16 format, the bfloat16 format, and the binary32, binary64, and binary128 formats.

parameters	binary16	bfloat16	binary32	binary64	binary128
p	11	8	24	53	113
$e_{\mathbf{max}}$	15	127	127	1023	16383
minmed	1/2	2^{-59}	2^{-51}	2^{-484}	2^{-8134}
maxmed	4	2^{59}	2^{51}	2^{485}	2^{8135}
Constraints on $t_{\mathbf{big}}$	$\frac{1}{4} \leq t_{\mathbf{big}} \leq 2^{-14}$ (IMPOSSIBLE)	$2^{-118} \leq t_{\mathbf{big}} \leq 2^{-70}$	$2^{-102} \leq t_{\mathbf{big}} \leq 2^{-78}$	$2^{-968} \leq t_{\mathbf{big}} \leq 2^{-540}$	$2^{-16268} \leq t_{\mathbf{big}} \leq 2^{-8250}$
Constraints on $t_{\mathbf{tiny}}$	$2^{24} \leq t_{\mathbf{tiny}} \leq 4$ (IMPOSSIBLE)	$2^{74} \leq t_{\mathbf{tiny}} \leq 2^{118}$	$2^{98} \leq t_{\mathbf{tiny}} \leq 2^{102}$	$2^{590} \leq t_{\mathbf{tiny}} \leq 2^{968}$	$2^{8360} \leq t_{\mathbf{tiny}} \leq 2^{16268}$

Computing Euclidean norms in the general case

Algorithm 9 Obtaining the norm from S_{big}, S_{med}, S_{tiny}

if BIG is nonempty **then**
  if $S^h_{\text{med}} < u^2 \text{minmed}^2 / t^2_{\text{big}}$ **or** $S^h_{\text{big}} > \text{maxmed}^2 \cdot t^2_{\text{big}} / u^3$ **then**
    return $\frac{1}{t_{\text{big}}} \cdot \text{SQRTDWtoFP}(S^h_{\text{big}}, S^\ell_{\text{big}}) = t_{\text{tiny}} \cdot \text{SQRTDWtoFP}(S^h_{\text{big}}, S^\ell_{\text{big}})$
  else
    compute $\hat{\chi} = \text{SloppyDWPlusDW}(t_{\text{tiny}} S^h_{\text{big}}, t_{\text{tiny}} S^\ell_{\text{big}}, t_{\text{big}} S^h_{\text{med}}, t_{\text{big}} S^\ell_{\text{med}})$
    return $\text{SQRTDWtoFP}(\hat{\chi})$
  end if
else
  if MED is nonempty **then**
    if TINY is empty **or** $S^h_{\text{tiny}} < \text{minmed}^2 u^2 / t^2_{\text{big}}$ **or** $S^h_{\text{med}} > \text{maxmed}^2 \cdot t^2_{\text{big}} / u^3$
    then
      return $\text{SQRTDWtoFP}(S^h_{\text{med}}, S^\ell_{\text{med}})$
    else
      compute $\hat{\chi} = \text{SloppyDWPlusDW}(t_{\text{tiny}} S^h_{\text{med}}, t_{\text{tiny}} S^\ell_{\text{med}}, t_{\text{big}} S^h_{\text{tiny}}, t_{\text{big}} S^\ell_{\text{tiny}})$
      return $\text{SQRTDWtoFP}(\hat{\chi})$
    end if
  else
    return $t_{\text{big}} \times \text{SQRTDWtoFP}(S^k_{\text{tiny}}, S^\ell_{\text{tiny}})$
  end if

Computing Euclidean norms in the general case

Theorem 3

If $n \leq \frac{1}{u}$, $k + m \leq \frac{1}{4u} - 2$ and $u \leq \frac{1}{32}$ and if the blockwise algorithm (Algorithm 8) is used for the summation of squares, with k blocks of m elements, then Algorithm 9 computes

$$\sqrt{\sum_{i=0}^{n-1} a_i^2}$$

with an error bounded by

$$\left(\frac{1}{2} + \frac{\big(3.12 + 2(k+m)\big)u + 1.8u^2}{1 - \frac{u}{2}}\right) \mathrm{ulp}\left(\sqrt{\sum_{i=0}^{n-1} a_i^2}\right),$$

without any risk of spurious underflow or overflow.

Accuracy comparisons

Table 3: For $S = 7, 8, \ldots, 14$, we generate $4096 \cdot 2^{14-S}$ arrays of uniform random lengths between 2^{S-1} and 2^S and for each term, we generate a uniform random exponent between $e_{\min} + p$ and $e_{\max} - p$ (to avoid non-spurious underflow/overflow) and uniform random significand between 1 and $2 - 2u$.

Algorithm	p	Maximum relative error/u	Rounding Faithful	Rounding Correct
Hammarling	24	15.3468	9 %	4 %
	53	6.4166	59 %	32 %
Graillat et al.	24	1.4916	100 %	87 %
	53	1.4608	100 %	89 %
Ours	24	0.9990	100 %	100 %
	53	0.9989	100 %	100 %

Important: we do not guarantee correct rounding (one can build ad-hoc cases for which Algorithm 9 does not return a correctly rounded result). The error bound being very near $\frac{1}{2}$ulp, incorrect rounding is just very unlikely (so that we do not observe it in experiments).

Performance comparisons on AMD Zen 2

Table 4: Comparisons of four algorithms on AMD Zen2, for three different array sizes, and three different profiles of input. For each entry, the mean value and standard deviation of a population of 100 000 runs is given.

AMD Zen2 (AVX2)				
Algorithm	*n*	Timing averages in microseconds AROUND_ONE	FULL_RANGE	REALLY_SMALL
straightforward (no scaling)	256	0.2(0)	0.4(0)	0.4(0)
	1024	1.0(0)	1.4(1)	1.4(1)
	4096	3.9(1)	5.7(2)	5.5(2)
Hammarling	256	0.4(0)	1.1(1)	0.8(1)
	1024	1.6(1)	4.4(1)	3.0(1)
	4096	6.5(2)	17.7(5)	12.1(4)
Graillat et al.	256	0.6(0)	0.6(0)	0.8(1)
	1024	2.1(0)	2.1(1)	3.2(1)
	4096	8.3(3)	8.3(3)	12.9(4)
Ours	256	0.5(0)	0.5(0)	0.7(1)
	1024	1.7(0)	1.7(0)	2.7(1)
	4096	6.7(2)	6.7(2)	10.6(3)

Performance comparisons on Intel Skylake

Table 5: Comparisons of four algorithms on Intel Skylake (AVX512), for three different array sizes, and three different profiles of input. For each entry, the mean value and standard deviation of a population of 100 000 runs is given.

Intel Skylake (AVX512) @3.0 GHz				
Algorithm	*n*	Timing averages in microseconds		
		AROUND_ONE	FULL_RANGE	REALLY_SMALL
Straightforward (no scaling)	256	0.4(0)	0.6(1)	1.1(1)
	1024	1.5(0)	2.3(2)	4.0(3)
	4096	6.1(1)	9.1(4)	15.8(6)
Hammarling	256	0.5(0)	1.3(3)	1.9(3)
	1024	1.6(1)	5.0(6)	7.3(6)
	4096	6.2(1)	19.6(12)	28.8(11)
Graillat et al.	256	0.5(1)	0.6(1)	1.5(2)
	1024	1.8(1)	1.8(1)	5.8(3)
	4096	6.8(1)	6.8(2)	22.8(7)
Ours	256	0.5(0)	0.6(1)	1.5(2)
	1024	1.6(0)	1.6(1)	5.4(3)
	4096	6.1(1)	6.1(2)	21.1(6)

Conclusion

- algorithm that computes euclidean norms of large vectors very accurately, and without spurious underflows or overflows;
- performance similar to that of the less accurate algorithm of Graillat et al.;
- not more than twice slower than the straightforward, under/overflow-prone, method;
- besides the euclidean norm:
  - when the operands are positive, the DWPlusFP algorithm has relative error bound u^2, and that bound is asymptotically optimal;
  - DW SQRT algorithm, with asymptotically optimal relative error bound.
- preprint available at `https://hal.archives-ouvertes.fr/hal-03482567`;
- code and formal proofs available on demand