Check for updates

A Rapid Euclidean Norm Calculation Algorithm that Reduces Overflow and Underflow

Takeyuki Harayama[1], Shuhei Kudo[2], Daichi Mukunoki[2], Toshiyuki Imamura[2], and Daisuke Takahashi[3(✉)]

[1] Graduate School of System and Information Engineering, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan
[2] Center for Computational Science, RIKEN, Kobe, Hyogo 650-0047, Japan
[3] Center for Computational Sciences, University of Tsukuba, Tsukuba, Ibaraki 305-8577, Japan
daisuke@cs.tsukuba.ac.jp

Abstract. The Euclidean norm is widely used in scientific and engineering calculations. However, a straightforward implementation may cause overflow/underflow and loss of accuracy. The Blue algorithm selects a scaling value from three conditional branches according to the absolute value to reduce overflow and underflow. We improve this algorithm and propose the Two-Accumulator method, which is expected to be faster by selecting it from two conditional branches. The input range of our method is slightly smaller than the Blue algorithm. However, we mitigate this problem by dynamically setting the scaling value depending on the vector size. Moreover, we combine it with double-double arithmetic to prevent rounding errors. An evaluation shows that our method combined with double-double arithmetic can be approximately 15% faster than the Blue algorithm while maintaining the same error level, but it can be approximately 46% slower, depending on the input range.

Keywords: Euclidean norm · Accurate computation · Overflow · Underflow

1 Introduction

The Euclidean norm is widely used in scientific and engineering calculations. For example, the Euclidean norm is used in the field of numerical analysis, as well as in linear calculations such as the Conjugate Gradient (CG) method and the Gram-Schmidt orthogonalization method. The Euclidean norm for the vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ shown in Eq. (1) is a simple calculation, where n is the vector size of the input vector.

$$\|\boldsymbol{x}\| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \tag{1}$$

However, the straightforward norm calculation algorithm shown as Algorithm 1 can only handle a limited range of inputs due to the effects of overflow and underflow. The Blue algorithm [4,7] and the Kahan algorithm [7] have

© Springer Nature Switzerland AG 2021
O. Gervasi et al. (Eds.): ICCSA 2021, LNCS 12949, pp. 95–110, 2021.
https://doi.org/10.1007/978-3-030-86653-2_7

been proposed to suppress overflow and underflow. However, the Blue algorithm requires at least three conditional branches, and the Kahan algorithm requires conditional branching and divisions, which are slow operations. On the other hand, there is a demand for accurate computations in scientific and engineering calculations, and MPLAPACK [13] provides high-precision BLAS and LAPACK routines. Euclidean norm calculations are prone to large errors due to the accumulation of rounding errors.

In this paper, we propose a method called the Two-Accumulator method to prevent overflow and underflow. We show that the implementation of double-word arithmetic (so-called double-double (DD) arithmetic) in the proposed method is as simple as the implementation in the Blue algorithm and can easily achieve higher accuracy. Dynamic scaling is also proposed to further reduce the overflow and underflow in the Two-Accumulator method.

Our contributions in this paper are as follows:

- The proposed method is able to suppress overflow and underflow with little change in the input range of the Blue algorithm,
- It combines well with DD arithmetic, making it easy to achieve high precision,
- It has the potential to be faster than the Blue algorithm.

In what follows, only double-precision arithmetic according to IEEE 754 is used, but application to other types of precision is possible as well.

This paper consists of the following sections. Section 2 explores related research. In Sect. 3, the Blue algorithm and the proposed method are described. Section 4 provides an overview of DD arithmetic, which is used as a precision expansion method to reduce rounding errors. The accuracy and performance are evaluated in Sect. 5. Finally, Sect. 6 summarizes the paper including suggestions for future work in this domain.

Algorithm 1. Straightforward norm calculation algorithm

Require: Vector $\boldsymbol{x}$, size n
Ensure: The l_2 norm of $\boldsymbol{x}$
1: sum $= x_0^2$;
2: **for** $i \leftarrow 1, n-1$ **do**
3: sum+= x_i^2;
4: **end for**
5: **return** $(\sqrt{\text{sum}})$;

2 Related Work

To suppress overflow and underflow in Euclidean norm calculations, the Blue and Kahan algorithms bring the value of the exponential part close to zero in advance by scaling. The Kahan algorithm continuously changes the scaling value of a vector $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ according to the maximum absolute value

of the input values up to x_i, which not only involves branching but also causes performance problems due to the frequent execution of slow division. The Blue algorithm is explained in the next section.

OpenBLAS [15] implements the Euclidean norm using the x87 instruction set, which uses 80-bit floating-point format. Even squares of values that overflow or underflow in double precision can be expressed entirely without overflow or underflow in 80-bit floating-point format. This makes it possible to calculate the Euclidean norm without using the Blue algorithm or the Kahan algorithm, which suppress overflow and underflow. Since the norm calculation in double-precision arithmetic has a large bytes/flop ratio and is memory-intensive, the performance may not change significantly whether the x87 instruction set or Single Instruction, Multiple Data (SIMD) instruction set is used. However, this approach cannot be used in environments where the x87 instruction set is not supported.

With the objective of improving accuracy, XBLAS [12] was proposed, which uses double-word arithmetic, like DD arithmetic, for internal operations. Further, MPLAPACK can arbitrarily select the precision from the GNU Multiple Precision Arithmetic Library (GMP) [1], MPFR [6], and the QD library [2,8].

3 Blue Algorithm and Two-Accumulator Method

3.1 Blue Algorithm

The maximum value of double-precision numbers, DBL_MAX, according to IEEE 754 is approximately 2^{1024}, and the minimum value, DBL_MIN, is approximately 2^{-1022}. Algorithm 2 shows the pseudo-code for the Blue algorithm. The Blue algorithm selects a scaling value from three conditional branches depending on the absolute value of the input. Inputs above 2^{300} are scaled by 2^{-600}. Inputs less than 2^{-300} are scaled by 2^{600}. Therefore, the value after scaling does not overflow or underflow when squared. The number of operations in the Blue algorithm is only one greater (a multiplication) than the straightforward norm calculation algorithm. However, the disadvantage of the Blue algorithm is that it requires branching for the scaling selection, making it inefficient to apply performance-enhancing methods such as vectorization.

3.2 Two-Accumulator Method with Fixed Scaling Value

The Two-Accumulator method is a simplified version of the Blue algorithm. This algorithm selects a scaling value from two conditional branches. The advantage of the Two-Accumulator method is the expected increase in speed because the number of conditional branches is smaller than in the Blue algorithm.

However, because the Two-Accumulator method only considers two conditional branches, some ranges that can be handled by the Blue algorithm cannot be handled by this simplified variant. In this section, we discuss the Two-Accumulator method with fixed scaling value 2^{498} for inputs less than 1.0 (hereinafter called *the Two-Accumulator method with fixed scaling value*). For example, the scaling value for inputs less than 1.0 in the Two-Accumulator method

Algorithm 2. Blue algorithm

Require: Vector $\boldsymbol{x}$, size n
Ensure: The l_2 norm of $\boldsymbol{x}$

$a_{\mathrm{big}} = a_{\mathrm{sml}} = a_{\mathrm{med}} = 0$;
overFlowLimit = DBL_MAX $* s_{\mathrm{h}}$;
for $i \leftarrow 0, n-1$ **do**
  absx $= |x_i|$
  if absx $\geq \gamma$ **then**
    a_{big}+= $(\mathrm{absx} * s_{\mathrm{h}})^2$;
  else if absx $\geq$ low **then**
    a_{med}+= absx^2;
  else
    a_{sml}+= $(\mathrm{absx} * s_{\mathrm{l}})^2$;
  end if
end for
if $a_{\mathrm{sml}} == 0$ **then**
  if $\sqrt{a_{\mathrm{big}}} \geq$ overFlowLimit **then**
    return (∞);
  else if $a_{\mathrm{med}} == 0$ **then**
    return $(\sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
  else
    $y_{\min} = \min(\sqrt{a_{\mathrm{med}}}, \sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
    $\mathrm{res} = \max(\sqrt{a_{\mathrm{med}}}, \sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
  end if
else if $a_{\mathrm{big}} == 0$ **then**
  if $a_{\mathrm{med}} == 0$ **then**
    return $(\sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}})$;
  else
    $y_{\min} = \min(\sqrt{a_{\mathrm{med}}}, \sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}})$;
    $\mathrm{res} = \max(\sqrt{a_{\mathrm{med}}}, \sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}})$;
  end if
else
  return $(\sqrt{a_{\mathrm{med}}})$;
end if
if $y_{\min} < \sqrt{\epsilon} * \mathrm{res}$ **then**
  return (res);
else
  return $(\mathrm{res} * \sqrt{1 + (y_{\min}/\mathrm{res})^2})$;
end if

with fixed scaling value is 2^{498}, so inputting values less than 2^{-1009} will cause underflow in the calculation of a square. Also, scaling for the input of $2^0 - 2^0 \times \epsilon$ (ϵ is the machine epsilon in double precision) results in $2^{996} \times (2^0 - \epsilon)$. Therefore, if the same input is given 2^{28} times, it will overflow. However, as will be shown in the next subsection, this can be mitigated by dynamic scaling.

3.3 Two-Accumulator Method with Dynamic Scaling

As already noted, the Two-Accumulator method with fixed scaling value cannot accommodate input values that are too large or too small. This is partly because calculating the square of a floating-point number will more than double the value of the exponent part. We should also take into account that the calculation of the Euclidean norm requires a sum of squares, which has an increasing partial sum. In other words, there is a relationship between n and the range of computable input values and the scaling values. Therefore, we propose a dynamic scaling method that sets the scaling value so that the range of computable input values is handled according to the length n of the input values. The scaling value (s_1) for inputs less than 1.0 is $s_1 = 2^{512-r}$ using r in Eq. (2).

$$r \geq \lceil (\log_2 n)/2 \rceil \tag{2}$$

If n is less than or equal to 4^r, which is the vector size that can be input, it will not overflow or underflow. However, the scaling value should be as large as possible because the input range becomes smaller when the scaling value is reduced. Therefore, r is the smallest value among the possible values satisfying $(\log_2 n)/2$. The reason for $r \geq \lceil (\log_2 n)/2 \rceil$ is that n must be less than or equal to the size of the input vector. Further, the maximum value of the input range below 1.0 is $1.0 - 1.0 \times \epsilon$, which is extremely close to 2^0. If 2^0 is multiplied by a power of two and squared, the result is always an even power of two. Therefore, it is possible to add an even power of two to the maximum value $2^{1024} - 2^{1024} \times \epsilon$ that can be expressed in double precision. Algorithm 3 is the pseudo-code for the Two-Accumulator method. The constants used in Algorithms 2 and 3 are listed in Table 3.

For inputs above 1.0, the vector size that can be input changes depending on the range of input values. Therefore, the scaling value for inputs above 1.0 is fixed at 2^{-511}, and by restricting the input range, overflow and underflow can be prevented. Hereafter, when we refer simply to the Two-Accumulator method, we mean the Two-Accumulator method with dynamic scaling.

3.4 Input Range and Vector Size

In the Euclidean norm calculation, overflow or underflow is most likely to occur when calculating a square. The range and vector size that can be input for each method are different because the number of partitions for each method is different. In this subsection, we discuss the vector sizes and ranges that can be computed for the inputs that are most likely to have overflow and underflow for each method. The worst-case input value is the one that minimizes the computable vector size when all the values of the input vector consist of this value.

The worst-case input values for each method and vector sizes that can be input are shown in Table 1. Table 2 shows the maximum range of input values for which a vector of at least two elements can be computed. Since the Euclidean norm for a vector $\boldsymbol{x}$ of size 1 is $\|\boldsymbol{x}\| = \sqrt{x_1^2} = |x_1|$, it can be calculated in all ranges except NaN (Not a Number). The maximum range shown in Table 2 is valid only when implemented with double-precision arithmetic according to IEEE 754, and the lower limit changes when DD arithmetic is used. The straightforward norm calculation algorithm can only calculate up to 2^0 in vector size when $2^{512} \times (2^0 - \epsilon)$ is input consecutively.

The range of possible inputs for the Two-Accumulator method and the Two-Accumulator method with fixed scaling value depends on the vector size of the input vectors. In the discussion of the worst-case input value, input values greater than 1.0 of the Two-Accumulator method are excluded. The reason for this is that, as explained in the last paragraph of the previous subsection, the worst-case input value for inputs greater than 1.0 of the Two-Accumulator method depends on the vector size of the input vector. The scaling value in the Two-Accumulator method with fixed scaling value is 2^{498}. The worst-case input value, in this case, is $2^0 - \epsilon$. Therefore, we can add 2^{28} times without going over DBL_MAX (approximately 2^{1024}). The worst-case input value in the Two-Accumulator method is also $2^0 - \epsilon$, and the minimum scaling value is 2^{496}, so if $2^0 - \epsilon$ is input consecutively, it can be input 2^{32} times. This means that inputs with a larger vector size and wider range can be used with the Two-Accumulator method compared to what is possible with the Two-Accumulator method with fixed scaling value.

In contrast, the worst-case input value for the Blue algorithm is DBL_MAX, which can be input 2^{176} times if this value is input consecutively. This is because the scaling value of the Blue algorithm for values less than 2^{300} is 2^{-600}. Therefore, the vector size that can be input is inferior to that of the Blue algorithm even for the Two-Accumulator method. However, in practical contexts where the Euclidean norm is used in scientific and engineering calculations, inputs that cannot be handled by the Two-Accumulator method would rarely occur.

Table 1. Worst-case input values for each method and vector sizes that can be input

	Worst value	Vector size
Straightforward norm calculation algorithm	$2^{512} \times (2^0 - \epsilon)$	2^0
Two-Accumulator method with fixed scaling value	$2^0 - \epsilon$	2^{28}
Two-Accumulator method	$2^0 - \epsilon$	2^{32}
Blue Algorithm	DBL_MAX	2^{176}

Algorithm 3. Two-Accumulator method

Require: Vector $\boldsymbol{x}$, size n
Ensure: The l_2 norm of $\boldsymbol{x}$
$a_{\mathrm{big}} = a_{\mathrm{sml}} = 0$;
overFlowLimit = DBL_MAX $* s_{\mathrm{h}}$;
$r = \lceil (\log_2 n)/2 \rceil$;
$s_{\mathrm{l}} = 2^{512-r}$;
for $i \leftarrow 0, n-1$ **do**
  if $|x_i| \geq \gamma$ **then**
    $a_{\mathrm{big}} \mathrel{+}= (x_i * s_{\mathrm{h}})^2$;
  else
    $a_{\mathrm{sml}} \mathrel{+}= (x_i * s_{\mathrm{l}})^2$;
  end if
end for
if $a_{\mathrm{sml}} == 0$ **then**
  return $(\sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
else if $a_{\mathrm{big}} == 0$ **then**
  return $(\sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}})$;
else
  if $\sqrt{a_{\mathrm{big}}} \geq$ overFlowLimit **then**
    return (∞);
  end if
  $y_{\min} = \min(\sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}}, \sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
  $\mathrm{res} = \max(\sqrt{a_{\mathrm{sml}}}/s_{\mathrm{l}}, \sqrt{a_{\mathrm{big}}}/s_{\mathrm{h}})$;
  if $y_{\min} < \sqrt{\epsilon} * \mathrm{res}$ **then**
    return (res);
  else
    return $(\mathrm{res} * \sqrt{1 + (y_{\min}/\mathrm{res})^2})$;
  end if
end if

Table 2. Maximum range that can be input for each method

	Range
Straightforward norm calculation algorithm	$2^{-511} \leq x_i \leq 2^{511}$
Two-Accumulator method with fixed scaling value	$2^{-1009} \leq x_i \leq 2^{1009}$
Two-Accumulator method	DBL_MIN $\leq x_i \leq 2^{1022}$
Blue Algorithm	DBL_MIN $\leq x_i \leq$ DBL_MAX

Table 3. Constants in the Blue algorithm, the Two-Accumulator method with fixed scaling value, and the Two-Accumulator method

	γ	low	s_{h}	s_{l}
Blue Algorithm	2^{300}	2^{-300}	2^{-600}	2^{600}
Two-Accumulator method with fixed scaling value	1.0	—	2^{-498}	2^{498}
Two-Accumulator method	1.0	—	2^{-511}	$2^{496} \sim 2^{512}$

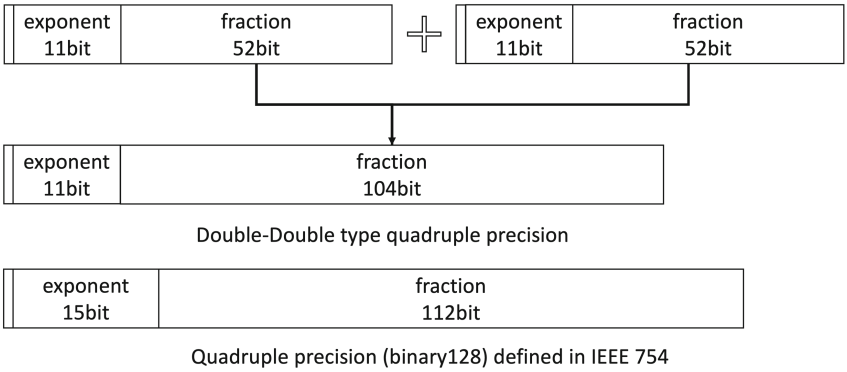


Fig. 1. Format difference between DD type and binary128

4 DD Arithmetic

4.1 Overview of DD Arithmetic

Precision expansion of floating-point arithmetic is a simple approach for reducing the accumulation of rounding errors and achieving more accurate calculations. In this paper, we leverage DD arithmetic, which is a low-cost method for achieving precision expansion. DD-type quadruple precision has 4 fewer bits in its exponent part and 8 fewer bits in its mantissa part than the quadruple precision (binary128) defined in IEEE 754 and shown in Fig. 1. DD arithmetic is faster than binary128 using software emulation. Also, the range of the exponential part that can be represented by DD-type quadruple precision is the same as that of double precision.

DD arithmetic is based on the error-free transformations by Knuth [10] and Dekker [5]. First, the addition by Knuth is shown in Algorithm 4. The result of the addition of the double-precision floating-point numbers a and b is denoted by s, and the error generated during the addition is denoted by e. The double-precision addition without error is expressed as $a + b = s + e$. Next, Dekker's addition is shown in Algorithm 5. Whilst this addition requires fewer operations compared to that of Knuth, it can only be used when $|a| \geq |b|$ is true. Next, the multiplication with error-free transformation using a Fused Multiply-Add (FMA) instruction is shown in Algorithm 6. Let p be the result of the calculation without considering the rounding error of double-precision multiplication $a \times b$ and let e be the error generated. The multiplication with error-free transformation can be expressed as $a \times b = p + e$. The multiplication with error-free transformation without using FMA instructions has also been proposed by Dekker [5].

We can calculate a DD addition using Algorithms 4 and 5, shown in Algorithm 7. The DD addition QuadAdd$(a_H, a_L, b_H, b_L, c_H, c_L)$ calculates $c = a + b$ for $a = a_H + a_L$, $b = b_H + b_L$, and $c = c_H + c_L$. QuadAdd needs to be normalized so that the calculation result $c = c_H + c_L$ satisfies $c_L \leq 0.5\text{ulp}(c_H)$, and Quick-TwoSum of Algorithm 5 is used for normalization. DD addition can be used so that calculations incur less rounding error compared to double-precision addition.

Algorithm 4. Addition with error-free transformation [10]

```
function [s, e] = TwoSum(a, b)
    s = a + b;
    v = s − a;
    e = (a − (s − v)) + (b − v);
end function
```

Algorithm 5. Dekker's addition [5]

```
function [s, e] = Quick-TwoSum(a, b)
    s = a + b;
    e = b − (s − a);
end function
```

DD addition can be accelerated using CPairSum by Lange and Rump [11,14]. The pseudo-code for CPairSum is shown in Algorithm 8. CPairSum omits the assumption that the calculation result satisfies $c_L \leq 0.5\mathrm{ulp}(c_H)$. This means that Quick-TwoSum, which is executed twice in QuadAdd, will not be executed in CPairSum. Therefore, CPairSum is slightly less accurate than QuadAdd, but can be computed with fewer operations. Specifically, CPairSum performs DD addition using the difference between the results of double-precision addition and TwoSum calculation. Also, QuadAdd uses TwoSum to calculate the lower bits, while CPairSum uses double-precision operations. QuadAdd requires 20 double-precision operations, while CPairSum can perform a DD addition in 11 operations.

4.2 Euclidean Norm Calculation Using DD Arithmetic

In this paper, we use DD addition to improve the accuracy of the Euclidean norm calculation where the input and output are double precision. In the Blue algorithm and the Two-Accumulator method, the scaling and squaring of the input values do not need to be computed using DD arithmetic. Both methods multiply powers of two as scaling values, and powers of two do not cause errors even in double-precision arithmetic. The square can be calculated with TwoProd-FMA shown in Algorithm 6.

When comparing the operations for the combination with DD arithmetic, the Blue method and the Two-Accumulator method only require one more double-precision multiplication for scaling compared to the straightforward norm calculation algorithm. If DD arithmetic is used in the Kahan algorithm, DD division

Algorithm 6. Multiplication using the FMA instruction [9]

```
function [p, e] = TwoProd-FMA(a, b)
    p = a ∗ b;
    e = a ∗ b − p;                    ▷ FMA
end function
```

Table 4. Measurement environment

CPU	Intel Xeon Gold 6126 (2.6 GHz)× 2 sockets
Memory	192GiB (DDR4)
OS	CentOS 7.7 (x86-64), kernel 3.10.0
Compiler	Intel Compiler 19.0.5.281

is required inside the loop, which complicates the implementation and makes it difficult to achieve high speed. Therefore, the Blue algorithm and the Two-Accumulator method are more effective when combined with DD arithmetic.

Algorithm 7. DD addition [2]

function $[c_H, c_L] = \mathrm{QuadAdd}(a_H, a_L, b_H, b_L)$
  $[sh, eh] = \mathrm{TwoSum}(a_H, b_H)$;
  $[sl, el] = \mathrm{TwoSum}(a_L, b_L)$;
  $eh = eh + sl$;
  $[sh, eh] = \text{Quick-TwoSum}(sh, eh)$;
  $eh = eh + el$;
  $[c_H, c_L] = \text{Quick-TwoSum}(sh, eh)$;
end function

Algorithm 8. Fast DD addition [11]

function $[c_H, c_L] = \mathrm{CPairSum}(a_H, a_L, b_H, b_L)$
  $c_H = a_H + b_H$;
  $[t, s] = \mathrm{TwoSum}(\mathrm{a_H}, \mathrm{b_H})$;
  $t = (t - c_H) + s$;
  $c_L = t + (a_L + b_L)$;
end function

5 Evaluation of Performance and Relative Error

5.1 Evaluation Method

We implemented algorithms for calculating the Euclidean norm (the straightforward norm calculation algorithm, the Blue algorithm, and the Two-Accumulator method) with double precision and combined them with precision expansion methods (QuadAdd and CPairSum). We evaluated the relative error and performance of these implementations as follows. The measurement environment is shown in Table 4. Evaluation was performed using a single core and a single thread. We specified -O3 -ipo -xSKYLAKE-AVX512 -fprotect-parens -no-vec -fma -unroll0 as the compilation options in the Intel C Compiler to evaluate

the relative error, and -O3 -ipo -xSKYLAKE-AVX512 -fprotect-parens -finline -restrict to evaluate the performance.

- Straightforward norm calculation algorithm + double-precision
- Straightforward norm calculation algorithm + QuadAdd
- Straightforward norm calculation algorithm + CPairSum
- Blue algorithm + double-precision
- Blue algorithm + QuadAdd
- Blue algorithm + CPairSum
- Two-Accumulator method + double-precision
- Two-Accumulator method + QuadAdd
- Two-Accumulator method + CPairSum

First, we input uniform random numbers of $[0, 1]$ generated with double precision for the input vector size $n = 10, 100, 1000, 10000, 100000$ and compare the relative error between the value computed by each method and the value computed using MPFR with a 256-bit mantissa part. Further, we compare the relative error when random numbers of $[2^{-1014}, 2^{1014}]$ are input to each method, and evaluate whether the Two-Accumulator method can prevent overflow and underflow. We normalize the difference between the values computed using MPFR and the values computed by each method by half of the computer epsilon to highlight how much error is introduced.

In the performance evaluation, we compare the extent to which the process of preventing overflow and underflow affects the performance. We evaluate the performance difference between combinations with CPairSum or QuadAdd. The vector size at the time of measurement is 100000.

5.2 Evaluation Results

Relative Error. Table 5 shows the relative error in $[2^{-1014}, 2^{1014}]$ when each method is implemented with double-precision arithmetic. The relative errors of each Euclidean norm calculation algorithm combined with each accuracy expansion method for uniform random numbers in $[0, 1]$ are shown in Tables 6, 7 and 8. In this evaluation, auto-vectorization by the compiler is turned off to evaluate the inherent relative error of the algorithm.

First, from the relative error of each Euclidean norm calculation algorithm for the input of $[2^{-1014}, 2^{1014}]$ combined with double-precision arithmetic, the relative error is Inf because of the overflow and underflow in the straightforward norm calculation algorithm. The Blue algorithm and the Two-Accumulator method do not overflow or underflow in the calculation of the sum of squares because of the scaling when the vector size is less than $n = 100000$ in the input range $[2^{-1014}, 2^{1014}]$.

Next, we describe the relative error of each Euclidean norm calculation algorithm for the input range $[0, 1]$ combined with double-precision arithmetic. Even the straightforward norm calculation algorithm does not overflow or underflow in the input range $[0, 1]$. The Blue algorithm and the Two-Accumulator method

Table 5. Relative error of each method combined with double-precision arithmetic for the input range of $[2^{-1014}, 2^{1014}]$

n	Straightforward norm calculation method		Blue algorithm		Two-Accumulator method	
	Average	Variance	Average	Variance	Average	Variance
10	N/A	N/A	0.05	0.03	0.05	0.03
100	N/A	N/A	0.30	0.09	0.30	0.09
1000	N/A	N/A	0.55	0.19	0.55	0.19
10000	N/A	N/A	1.25	0.98	1.25	0.98
100000	N/A	N/A	5.26	13.79	5.26	13.79

Table 6. Relative error of each method combined with double-precision arithmetic for the input range of $[0, 1]$

n	Straightforward norm calculation method		Blue algorithm		Two-Accumulator method	
	Average	Variance	Average	Variance	Average	Variance
10	0.47	0.12	0.47	0.12	0.47	0.12
100	1.01	0.58	1.01	0.58	1.01	0.58
1000	3.30	6.18	3.30	6.18	3.30	6.18
10000	9.96	56.34	9.96	56.34	9.96	56.34
100000	26.51	439.52	26.51	439.52	26.51	439.52

have comparable relative errors and variances. This is because the input value is from a single region for both methods, and the computation procedure for both is the same except for the scaling by powers of two. Further, the accuracy can be improved by enabling auto-vectorization by the compiler. This is because the number of accumulators in each region changes in the Blue algorithm and the Two-Accumulator method. The accumulation of rounding errors can be mitigated by varying the number of accumulators when calculating the sum [3].

For DD-type quadruple precision, the maximum and minimum absolute values that can be represented do not change, because the exponent values are the same as in double precision. Therefore, even if the straightforward norm calculation algorithm is implemented using DD arithmetic, the conditions for the occurrence of overflow and underflow are unchanged. The use of DD arithmetic extends the number of bits in the mantissa part and prevents the accumulation of rounding errors from propagating to the double-precision output. The relative error between the Blue algorithm and the Two-Accumulator method is smaller than that combined with double-precision arithmetic.

CPairSum is a fast algorithm for DD addition (QuadAdd), so the accuracy is approximately the same as combined with QuadAdd.

Table 7. Relative error of each method combined with QuadAdd for the input range of $[0, 1]$

n	Straightforward norm calculation method		Blue algorithm		Two-Accumulator method	
	Average	Variance	Average	Variance	Average	Variance
10	0.38	0.07	0.38	0.07	0.38	0.07
100	0.39	0.07	0.39	0.07	0.39	0.07
1000	0.47	0.09	0.47	0.09	0.47	0.09
10000	0.30	0.04	0.30	0.04	0.30	0.04
100000	0.42	0.07	0.42	0.07	0.42	0.07

Table 8. Relative error of each method combined with CPairSum for the input range of $[0, 1]$

n	Straightforward norm calculation method		Blue algorithm		Two-Accumulator method	
	Average	Variance	Average	Variance	Average	Variance
10	0.38	0.07	0.38	0.07	0.38	0.07
100	0.39	0.07	0.39	0.07	0.39	0.07
1000	0.47	0.09	0.47	0.09	0.47	0.09
10000	0.30	0.04	0.30	0.04	0.30	0.04
100000	0.42	0.07	0.42	0.07	0.42	0.07

Performance. The performance of each method combined with double-precision arithmetic is shown in Fig. 2, and the performance combined with QuadAdd and CPairSum is shown in Fig. 3. GFLOPS is usually used as a measure of the performance of the Euclidean norm, but GB/s is used for the vertical axes in Figs. 2 and 3. The reason for this is that the computational amount varies slightly depending on the method and input values, and since DD arithmetic is used, GB/s, which is the unit of input vector size divided by the execution time, is more suitable for performance comparison. The straightforward norm calculation algorithm has overflow and underflow for input ranges other than $[0, 1]$ and $[2^{-300}, 2^{300}]$. Therefore, this excludes a performance comparison. First, we focus on the extent to which the process of suppressing overflow and underflow affects the performance. In the case of the implementation combined with double precision in the input range of $[0, 1]$, the performance of the straightforward norm calculation algorithm is twice that of the Two-Accumulator method. However, the performance of the implementations combined with QuadAdd and CPairSum is almost identical. This is because the Euclidean norm calculation is memory-intensive at double precision, but computation-intensive at the DD-type quadruple-precision level. The implementation combined with double precision is faster in the order of the straightforward norm calculation algorithm, the

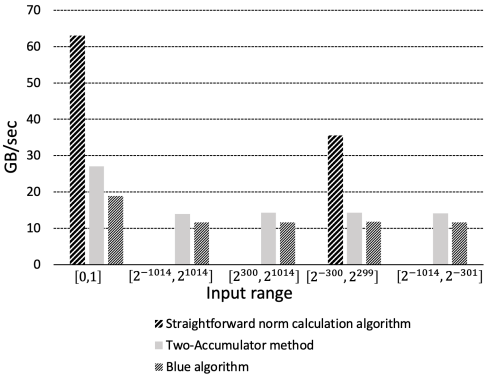


Fig. 2. Performance of implementations combined with double-precision arithmetic

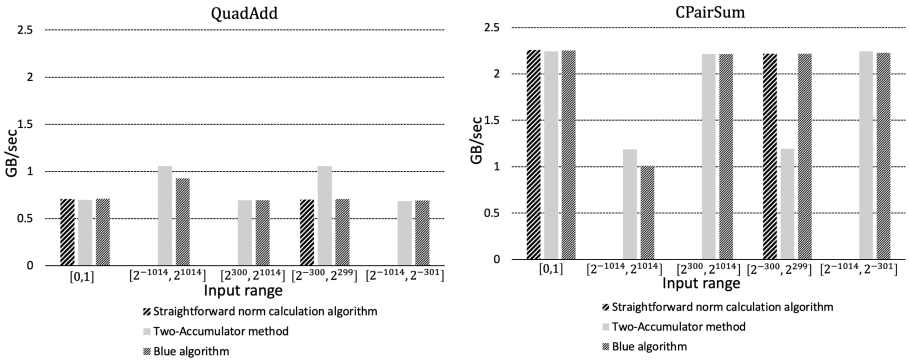


Fig. 3. Performance evaluation when combined with QuadAdd and CPairSum

Two-Accumulator method, and the Blue algorithm. The reason for this is that each method is vectorized by the compiler's automatic vectorization. Implementations that use double-precision arithmetic are vectorized by the compiler, but implementations that use DD-type quadruple-precision arithmetic are not vectorized, because there is a dependency in the process of adding the squares together. Further, the vectorization rate is higher for the straightforward norm calculation algorithm, the Two-Accumulator method, and the Blue algorithm, in that order. Therefore, the Two-Accumulator method is easier to speed up than the Blue algorithm because the number of conditional branches greatly affects the vectorization rate.

In the implementation combined with QuadAdd, the Two-Accumulator method is faster than or nearly as fast as the Blue algorithm for all input ranges. The Two-Accumulator method is faster than the Blue algorithm when all the accumulators are stored, such as $[2^{-1014}, 2^{1014}]$. For the other input ranges, the Two-Accumulator method and the Blue method are nearly identical in performance.

The performance of the implementation combined with CPairSum is up to 3.3 times faster than the implementation combined with QuadAdd. However, the Blue algorithm is faster than the Two-Accumulator method in the input range of $[2^{-300}, 2^{299}]$ only implemented using CPairSum. One possible reason for this is the relationship between the number of unpredictable conditional branches and the number of operations. The Blue algorithm processes this range of input as a single region, while the Two-Accumulator method processes it as two regions. Therefore, in this range, the Two-Accumulator method is being used under conditions such that the CPU is likely to fail in branch prediction. Therefore, the implementation combined with QuadAdd is less likely to show this difference than the implementation CPairSum because more operations are required for DD addition. On the other hand, in the implementation combined with CPairSum, the penalty for branch prediction failure is more likely to be reflected in the result, and the Blue algorithm is considered to be faster than the Two-Accumulator method.

6 Summary and Future Work

In this paper, we propose the Two-Accumulator method as an algorithm for calculating the Euclidean norm with less overflow and underflow. We combine this novel method with DD arithmetic to achieve higher accuracy. Further, we use CPairSum, which is an algorithm to speed up DD addition. The algorithms for calculating the Euclidean norm are compared in terms of input range, vector size, relative error, and performance. In terms of ease of overflow and underflow, the Two-Accumulator method is inferior to the Blue algorithm. However, in terms of accuracy, the Two-Accumulator method and the Blue algorithm are nearly identical in performance, even when combined with an accuracy expansion method. The Two-Accumulator method performs better than the Blue algorithm in the input range of $[2^{-1014}, 2^{1014}]$ combined with QuadAdd. However, the Two-Accumulator method combined with CPairSum is significantly slower than the Blue algorithm, depending on the input range. Vectorization of the Two-Accumulator method represents fruitful terrain for future research. In vectorization, the branching process changes significantly, and the number of branches has a large impact on performance. The advantage of the Two-Accumulator method with fewer conditional branches is emphasized, and the Two-Accumulator method has the potential to be faster in all ranges, as in the performance of implementations combined with double-precision arithmetic.

Acknowledgment. This research was supported in part by the Multidisciplinary Cooperative Research Program (Cygnus) in the Center for Computational Sciences (CCS), University of Tsukuba.

References

1. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org
2. Bailey, D.H.: QD: A double-double and quad-double package for Fortran and C++. https://www.davidhbailey.com/dhbsoftware/
3. Blanchard, P., Higham, N.J., Mary, T.: A class of fast and accurate summation algorithms. SIAM J. Sci. Comput. **42**, 1541–1557 (2020)
4. Blue, J.L.: A portable Fortran program to find the Euclidean norm of a vector. ACM Trans. Math. Softw. **4**(1), 15–23 (1978)
5. Dekker, T.J.: A floating-point technique for extending the available precision. Numer. Math. **18**(3), 224–242 (1971)
6. Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P., Zimmermann, P.: The GNU MPFR Library. https://www.mpfr.org/
7. Hanson, R.J., Hopkins, T.: Remark on algorithm 539: a modern Fortran reference implementation for carefully computing the Euclidean norm. ACM Trans. Math. Softw. **44**(3), Article 24 (2017)
8. Hida, Y., Li, X.S., Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic. In: Proceedings of the 15th IEEE Symposium on Computer Arithmetic, pp. 155–162 (2001)
9. Karp, A.H., Markstein, P.: High-precision division and square root. ACM Trans. Math. Softw. **23**(4), 561–589 (1997)
10. Knuth, D.E.: The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd edn. Addison-Wesley, Boston (1997)
11. Lange, M., Rump, S.M.: Faithfully rounded floating-point computations. ACM Trans. Math. Softw. **46**(3), Article 21 (2020)
12. Li, X.S., et al.: Design, implementation and testing of extended and mixed precision BLAS. ACM Trans. Math. Softw. **28**(2), 152–205 (2002)
13. Nakata, M.: The MPACK (MBLAS/MLAPACK); a multiple precision arithmetic version of BLAS and LAPACK (2010). http://mplapack.sourceforge.net/
14. Rump, S.M.: Error bounds for computer arithmetics. In: 2019 IEEE 26th Symposium on Computer Arithmetic (ARITH), pp. 1–14 (2019)
15. Xianyi, Z., Qian, W., Chothia, Z.: OpenBLAS. http://www.openblas.net