Journal of Information Processing Vol.21 No.2 315–319 (Apr. 2013)

[DOI: 10.2197/ipsjjip.21.315]

Technical Note

Fast Computation of the n-th Root in Quad-double Arithmetic Using a Fourth-order Iterative Scheme

Tsubasa Saito[1,a)]

Received: August 22, 2012, Accepted: December 7, 2012

Abstract: We propose new algorithms for computing the n-th root of a quad-double number. We construct an iterative scheme that has quartic convergence and propose algorithms that require only about 50% to 60% of the double-precision arithmetic operations of the existing algorithms. The proposed algorithms perform about 1.7 times faster than the existing algorithms, yet maintain the same accuracy. They are sufficiently effective and efficient to replace the existing algorithms.

Keywords: octuple-precision, quad-double arithmetic, square root, n-th root, modified Newton's method

1. Introduction

High-precision arithmetic is needed to verify the numerical results computed by double-precision arithmetic, and quad-double arithmetic has been proposed for quasi-octuple-precision arithmetic [8]. These arithmetic operators can be achieved by combining double-precision arithmetic operations. However, computing the n-th root of a quad-double number requires many more double-precision arithmetic operations, because the computation is based on Newton's method.

In this paper, we propose two new algorithms, one each for computing the square root and the n-th root of a quad-double number. We reconsidered the existing algorithms, which are based on the second-order Newton iterative scheme, and constructed a fourth-order iterative scheme. To reduce the number of double-precision arithmetic operations, we considered the following strategies: (i) Rearranging terms to reuse intermediate results; (ii) Using multiplication by the power of 2; and (iii) Replacing quad-double multiplication by addition.

We compared the number of double-precision arithmetic operations, the computation time, and the accuracy of both new algorithms with the existing algorithms. The proposed algorithms require only 50% to 60% of the double-precision arithmetic operations required by the existing algorithms in the QD package [1]. The proposed algorithms can be executed much faster than the existing algorithms, and have almost the same results as the existing algorithm in 63 decimal digits. The proposed algorithms are thus effective for computing the n-th root of a quad-double number.

2. Double-double and Quad-double

Double-double and quad-double arithmetic were proposed for quasi-quadruple and -octuple-precision arithmetic by Hida et al. [8]. A double-double number is represented by two, and a quad-double number is represented by four, double-precision numbers. Throughout this paper, the variables with a subscript as $x_{(dd)}$ and $y_{(qd)}$ mean a double-double number and a quad-double number respectively. The symbols $\cdot$ ($\cdot \in \{+, -, \times, /\}$) represent the four arithmetic operators based on mathematics, and $\odot$ ($\odot \in \{\oplus, \ominus, \otimes, \oslash\}$) represent the double-precision operators performed on the computer. A double-double number $x_{(dd)}$ and a quad-double number $y_{(qd)}$ are represented by double-precision numbers $x_0, x_1, y_0, y_1, y_2, y_3$ as follows:

$$x_{(dd)} = x_0 + x_1, \ \ y_{(qd)} = y_0 + y_1 + y_2 + y_3.$$

The components x_0, x_1, y_0, y_1, y_2 and y_3 satisfy the following inequalities:

$$|x_1| \le \frac{1}{2}\mathrm{ulp}(x_0), \ \ |y_{i+1}| \le \frac{1}{2}\mathrm{ulp}(y_i), \ \ i = 0, 1, 2,$$

where ulp stands for 'units in the last place'. A double-double (quad-double) number has 31(63) significant decimal digits. These can be computed by using only double-precision arithmetic operations. Addition, subtraction, and multiplication of quad-double numbers are based on error-free floating-point arithmetic algorithms such as Two-Sum [9] and Two-Prod [3]. There are no error-free algorithms for division or for finding the square root, so computing these are based on Newton's method. As a result, for quad-double numbers, these operations need more double-precision arithmetic operations than do addition and multiplication. The details of the algorithms for double-double and quad-double arithmetic are shown in Refs. [10], [11].

3. Existing Algorithms for the n-th Root

Algorithm 1 shows the procedure for computing the square root of a quad-double number using the QD package [1], which is one of the most popular libraries using quad-double arithmetic

[1] Graduate School of Science, Tokyo University of Science, Shinjuku, Tokyo 162–8601, Japan

a) t.saito0106@gmail.com

written in C++. It is based on division-free Newton-Raphson iteration:

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^2)}{2}, \ i = 0, 1, \ldots, \tag{1}$$

which converges to $\frac{1}{\sqrt{a}}$, starting with the double-precision approximation to $\frac{1}{\sqrt{a}}$. To get the square root $\sqrt{a}$, we need to multiply the result by the input argument a. This algorithm does not require division of quad-double numbers. Note that round-off errors do not occur in multiplication or division by the power of 2, and in this case, multiplication by 0.5 can be done componentwise in line 3.

Algorithm 2 for an n-th root ($n > 2$) can be designed in the same way as Algorithm 1. It is based on the following second-order iterative scheme:

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^n)}{n}, i = 0, 1, \ldots. \tag{2}$$

A round-off error may occur with division by n. Unlike computation of the square root, Algorithm 2 requires quad-double division in lines 8 and 11. Also, Algorithm 2 requires computation of an n-th power of a quad-double number in line 4, which requires more double-precision arithmetic operations than computing a square.

Algorithm 1 Quad-double square root in the QD package

```
 1: $app \leftarrow$ sqrt($a_0$)
 2: $r_{(qd)} \leftarrow$ qd(1.0 $\oslash$ $app$)
 3: $h_{(qd)} \leftarrow$ mul_pwr_qd($a_{(qd)}$, 0.5)
 4: for $i$ = 1 to 3 do
 5:     $x_{(qd)} \leftarrow$ qd_sqr($r_{(qd)}$)
 6:     $y_{(qd)} \leftarrow$ qd_qd_mul($h_{(qd)}, x_{(qd)}$)
 7:     $w_{(qd)} \leftarrow$ d_qd_sub(0.5, $y_{(qd)}$)
 8:     $z_{(qd)} \leftarrow$ qd_qd_mul($w_{(qd)}, r_{(qd)}$)
 9:     $r_{(qd)} \leftarrow$ qd_qd_add($r_{(qd)}, z_{(qd)}$)
10: end for
11: $res_{(qd)} \leftarrow$ qd_qd_mul($r_{(qd)}, a_{(qd)}$)
12: return $res_{(qd)}$
```

Algorithm 2 Quad-double n-th root in the QD package

```
 1: $app \leftarrow$ pow($a_0$, −1.0 $\oslash$ $n$)
 2: $r_{(qd)} \leftarrow$ qd($app$)
 3: for $i$ = 1 to 3 do
 4:     $x_{(qd)} \leftarrow$ qd_pow($r_{(qd)}, n$)
 5:     $y_{(qd)} \leftarrow$ qd_qd_mul($a_{(qd)}s, x_{(qd)}$)
 6:     $w_{(qd)} \leftarrow$ d_qd_sub(1.0, $y_{(qd)}$)
 7:     $z_{(qd)} \leftarrow$ qd_qd_mul($w_{(qd)}, r_{(qd)}$)
 8:     $v_{(qd)} \leftarrow$ qd_d_div($z_{(qd)}, n$)
 9:     $r_{(qd)} \leftarrow$ qd_qd_add($r_{(qd)}, v_{(qd)}$)
10: end for
11: $res_{(qd)} \leftarrow$ d_qd_div(1.0, $r_{(qd)}$)
12: return $res_{(qd)}$
```

4. Proposal of a New Algorithm

We consider computing by a modified Newton's method. Recently, high-order iterative schemes focused on the square root or the n-th root have been proposed (Refs. [4], [7]). However, such schemes are often complicated to compute with quad-double arithmetic. In contrast, a simple approach was proposed by Gerlach [6]. Gerlach focused on a class of functions that converges well with Newton's method, and then modified Newton's method for more rapid convergence. Ford and Pennline [5] improved Gerlach's scheme to obtain the same result but can be implemented more easily.

We propose a new fourth-order iterative algorithm that is based on the method of Ford and Pennline [5]. The mantissa of a quad-double number is four times longer than that of a double-precision number. If the scheme has quartic convergence, the accuracy of an approximate solution increases four times with each iteration. Using a square root with double-precision arithmetic as an initial value, we can obtain the n-th root of a quad-double number with one iteration. First we present a theorem for the high-order iterative scheme in Ref. [5]. Using this, a fourth-order iterative scheme for computing the n-th root is constructed.

Theorem. (Ford and Pennline [5]) *If Newton's method has M-th-order convergence for a given function f, then the solution to $f(x) = 0$ may be obtained by applying the modified Newton's method*

$$x_{i+1} = x_i - \frac{f(x_i)Q_N(x_i)}{Q_{N+1}(x_i)}, \quad i = 0, 1, 2, \ldots, \tag{3}$$

where the function Q_N is defined for $N \geq M$ by

$$\begin{cases} Q_M(x) = 1, & (4)\\ Q_{N+1}(x) = f'(x)Q_N(x) - \dfrac{1}{N-1}f(x)Q'_N(x), \ N \geq M, & (5)\end{cases}$$

the scheme having N-th-order convergence, i.e.,

$$|x_{i+1} - a| \leq C|x_i - a|^N,$$

for some constant C and the solution a.

From this theorem, an N-th-order iterative scheme can be obtained by calculating $Q_M, Q_{M+1}, \ldots, Q_{N+1}$ according to Eqs. (4) and (5), and substituting them into Eq. (3). We can deductively generate an iterative scheme that has more than M-th-order convergence.

Let a function f be $f(x) = \frac{1}{x^n} - \frac{1}{a}$. Newton's method has quadratic convergence, i.e., $M = 2$. We calculate Q_3, Q_4, and Q_5 and substitute them into Eq. (3). Then the iterative scheme becomes

$$x_{i+1} = x_i - \frac{3x_i(x_i^n - a)\big((n+1)x_i^n + (n-1)a\big)}{(n^2 + 3n + 2)x_i^{2n} + 4(n^2 - 1)ax_i^n + (n^2 - 3n + 2)a^2}. \tag{6}$$

This is the fourth-order iterative scheme for computing the n-th root of a quad-double number. Let x_i be the n-th root computed by double-precision; then x_{i+1} becomes the n-th root of a quad-double number. This is faster than the existing algorithm.

4.1 Algorithm for the Square Root

When $n = 2$ in Eq. (6), we obtain a simple scheme for computing the square root:

$$x_{i+1} = x_i - \frac{(x_i^2 - a)(3x_i^2 + a)}{4x_i(x_i^2 + a)}. \tag{7}$$

This iterative scheme is known as the Bakhshali square root formula from ancient times [2]. Although division-free Newton-Raphson iteration does not require quad-double division, Eq. (7) does. To reduce the number of double-precision arithmetic operations, we consider the following: (i) Rearranging terms to reuse intermediate computation results; (ii) Using multiplication by the power of 2; and (iii) Replacing quad-double multiplication with addition. The reason for (iii) is that quad-double addition needs fewer double-precision arithmetic operations than does quad-double multiplication.

Algorithm 3 shows the procedure based on Eq. (7). In order to compute $3x_i^2 + a$, two quad-double operations are required. Then we compute $3x_i^2 + a$ by transforming it into $(x_i^2 + a) + 2x_i^2$. The first term $x_i^2 + a$ was computed in line 3, and the second term $2x_i^2$ does not have round-off error. Only two double-precision multiplications are needed for computing $2x_i^2$ in line 5. Thus $3x_i^2 + a$ can be computed by two double-precision multiplications and one mixed-precision addition of double-double and quad-double in line 6. $4x_i$ is also computed with no round-off errors in line 8. As a result, the proposed algorithm requires only about half the double-precision arithmetic operations required by the algorithm in QD package.

4.2 Algorithm for the *n*-th Root

We reconstruct the iterative scheme Eq. (6) to reduce the number of double-precision arithmetic operations as follows:

$$\begin{aligned} x_{i+1} &= x_i - \frac{3x_i(x_i^n - a)\big((n+1)x_i^n + (n-1)a\big)}{(n^2+3n+2)x_i^{2n} + 4(n^2-1)ax_i^n + (n^2-3n+2)a^2} \\ &= x_i - \frac{F(x_i)}{G(x_i)}, \end{aligned} \tag{8}$$

Algorithm 3 Proposed algorithm for computing the square root of a quad-double number

```
 1: $app \leftarrow \mathrm{sqrt}(a_0)$
 2: $x_{(dd)} \leftarrow \text{Two-Sqr}(app)$
 3: $s_{(qd)} \leftarrow \text{dd\_qd\_add}(x_{(dd)}, a_{(qd)})$
 4: $t_{(qd)} \leftarrow \text{dd\_qd\_sub}(x_{(dd)}, a_{(qd)})$
 5: $p_{(dd)} \leftarrow \text{mul\_pwr\_dd}(x_{(dd)}, 2)$
 6: $u_{(qd)} \leftarrow \text{dd\_qd\_add}(p_{(dd)}, s_{(qd)})$
 7: $v_{(qd)} \leftarrow \text{qd\_qd\_mul}(u_{(qd)}, t_{(qd)})$
 8: $q \leftarrow 4 \otimes app$
 9: $w_{(qd)} \leftarrow \text{d\_qd\_mul}(q, s_{(qd)})$
10: $z_{(qd)} \leftarrow \text{qd\_qd\_div}(v_{(qd)}, w_{(qd)})$
11: $res_{(qd)} \leftarrow \text{qd}(app, -z_0, -z_1, -z_2)$
12: return $res_{(qd)}$
```

Algorithm 4 Proposed algorithm for computing the *n*-th root of a quad-double number

```
 1: $app \leftarrow \mathrm{pow}(a_0, 1.0 \oslash n)$
 2: $x_{(qd)} \leftarrow \text{qd\_pow}(\text{qd}(app), n)$
 3: $s_{(qd)} \leftarrow \text{qd\_qd\_add}(x_{(qd)}, a_{(qd)})$
 4: $t_{(qd)} \leftarrow \text{qd\_qd\_sub}(x_{(qd)}, a_{(qd)})$
 5: $u_{(qd)} \leftarrow \text{d\_qd\_mul}(n, s_{(qd)})$
 6: $v_{(qd)} \leftarrow \text{qd\_qd\_add}(u_{(qd)}, t_{(qd)})$
 7: $w_{(qd)} \leftarrow \text{qd\_qd\_mul}(t_{(qd)}, v_{(qd)})$
 8: $y_{(qd)} \leftarrow \text{d\_qd\_mul}(app, w_{(qd)})$
 9: $z_{(qd)} \leftarrow \text{mul\_pwr\_qd}(y_{(qd)}, 2)$
10: $b_{(qd)} \leftarrow \text{qd\_qd\_add}(z_{(qd)}, y_{(qd)})$
11: $nn \leftarrow n \otimes n$
12: $p_{(qd)} \leftarrow \text{d\_qd\_mul}(nn \oplus 3 \otimes n \oplus 2, s_{(qd)})$
13: $q_{(qd)} \leftarrow \text{d\_qd\_mul}(6 \otimes n, a_{(qd)})$
14: $r_{(qd)} \leftarrow \text{qd\_qd\_sub}(p_{(qd)}, q_{(qd)})$
15: $c_{(qd)} \leftarrow \text{qd\_qd\_mul}(s_{(qd)}, r_{(qd)})$
16: $d_{(qd)} \leftarrow \text{d\_qd\_mul}(2 \otimes (nn \ominus 4), a_{(qd)})$
17: $e_{(qd)} \leftarrow \text{qd\_qd\_mul}(d_{(qd)}, x_{(qd)})$
18: $g_{(qd)} \leftarrow \text{qd\_qd\_add}(c_{(qd)}, e_{(qd)})$
19: $h_{(qd)} \leftarrow \text{qd\_qd\_div}(b_{(qd)}, g_{(qd)})$
20: $res_{(qd)} \leftarrow \text{qd}(app, -h_0, -h_1, -h_2)$
21: return $res_{(qd)}$
```

where

$$\begin{cases} F(x_i) = x_i(x_i^n - a)\Big(n(x_i^n + a) + (x_i^n - a)\Big) \\ \qquad\qquad + 2x_i(x_i^n - a)\Big(n(x_i^n + a) + (x_i^n - a)\Big), \\ G(x_i) = (x_i^n + a)\Big((n^2+3n+2)(x_i^n + a) - 6na\Big) + 2(n^2-4)ax_i^n. \end{cases}$$

Algorithm 4 shows the procedure based on Eq. (8). As for the square root, the intermediate results $(x^n \pm a)$ are used repeatedly in lines 5, 6, 7, 12, and 16. We transform $3x_i(x_i^n - a)\Big((n+1)x_i^n + (n-1)a\Big)$ into $F(x_i)$ to apply multiplication by 2 and replace the quad-double multiplication with addition. The proposed algorithm requires about 60% the number of double-precision arithmetic operations of the algorithm in the QD package.

5. Comparison

We compared the number of double-precision arithmetic operations, the computation time, and the accuracy. Experiments were carried out on a PC with an Intel Core i5 1.7 GHz CPU, 4 GB memory, and Scilab version 5.3.3 running on a Mac OS X Lion. To execute the algorithms, we use our MuPAT [10], which is implemented in double-double and quad-double arithmetic based on the QD package on Scilab.

5.1 Number of Double-precision Arithmetic Operations

Table 1 shows the number of double-precision arithmetic operations for both algorithms. 'Double' is the number of double-precision arithmetic operations needed for each function. The numbers in the 3rd to 6th column show the number of times functions appear in each algorithm. 'Number of double' is the total number of double-precision arithmetic operations for each algorithm. The `sqrt` and `pow` functions do not count in the total number of double-precision arithmetic operations. For computing the square root, the existing algorithm in the QD package needs 2,445 double-precision arithmetic operations, but the proposed algorithm only needs 1,212, about half as many.

The number of double-precision arithmetic operations for the *n*-th power depends on the exponent *n*. **Table 2** shows the num-

Table 1 The count of functions in each function, and the number of double-precision arithmetic operations for Algorithm 1 to 4.

		Square root		*n*-th root	
'Function'	'Double'	QD	Proposed	QD	Proposed
Two-Sqr	12	-	1	-	-
dd_qd_add	71	-	2	-	-
qd_qd_add	91	3	-	3	4
d_qd_sub	52	3	-	3	-
dd_qd_sub	71	-	1	-	-
qd_qd_sub	91	-	-	-	2
d_qd_mul	118	-	1	-	5
qd_qd_mul	217	7	1	6	3
d_qd_div	610	-	-	1	-
qd_d_div	286	-	-	3	-
qd_qd_div	649	-	1	-	1
qd_sqr	164	3	-	-	-
mul_pwr_dd	2	-	1	-	-
mul_pwr_qd	4	1	-	-	1
$\oplus$, $\ominus$, $\otimes$, $\oslash$	1	1	1	1	8
`sqrt`	(+1)	1	1	-	-
`pow`	(+1)	-	-	1	1
qd_pow	$X(n)$	-	-	3	1
'Number of double'		2,445	1,212	$3{,}200+3X(n)$	$2{,}458+X(n)$

$X(n)$: Depends on the exponent n (see Table 2)

Table 2 Number of double-precision arithmetic operations for the n-th power and the total number of double-precision arithmetic operations for n-th root ($3 \leq n \leq 10$).

	n-th power	n-th root		
n	$X(n)$	QD	Proposed	Ratio(%)
3	600	5,000	3,058	61.2
4	548	4,844	3,006	62.1
5	765	5,495	3,223	58.7
6	765	5,495	3,223	58.7
7	982	6,146	3,440	56.0
8	713	5,339	3,171	59.4
9	930	5,990	3,388	56.6
10	930	5,990	3,388	56.6

Table 3 Comparison of the computation time in seconds.

n	QD (sec.)	Proposed (sec.)	Speed-up ratio
2	0.982	0.561	1.75
3	2.315	1.414	1.64
4	2.257	1.415	1.59
5	2.530	1.484	1.70
6	2.534	1.484	1.71
7	2.739	1.551	1.77
8	2.477	1.480	1.67
9	2.733	1.554	1.76
10	2.740	1.550	1.77

ber of double-precision arithmetic operations for computing the n-th power and the n-th root ($3 \leq n \leq 10$) of a quad-double number. The algorithm in the QD package requires about 4,800 to 6,100 double-precision arithmetic operations, but the proposed algorithm needs only about 3,000 to 3,400, reducing the number of double-precision arithmetic operations by 56% to 62%. The proposed algorithms are thus faster than the existing algorithms.

5.2 Computation Time

We generated one million quad-double random numbers between 0 and 1 with the function `qdrand` in MuPAT; it is a quad-double random value generator based on the Scilab function `rand`. **Table 3** shows the computation times (each one is the average of ten trials) and the speed-up ratios. The computation time includes calling the functions to execute the algorithms on Scilab. In the case of the square root, the computation times using the existing algorithm and the proposed algorithm are 0.982 seconds and 0.561 seconds, respectively. The proposed algorithm is 1.75 times faster than the existing algorithm. In every case where $3 \leq n \leq 10$, the proposed algorithm is faster than the existing algorithm.

The existing algorithm is a sequential algorithm and lines 5 to 9 in Algorithm 1 use the previous computation results. On the other hand, the proposed algorithm can compute in parallel. For example, the numerator and denominator of the iterative scheme Eqs. (7) and (8) can be computed separately, which can lead to further acceleration.

5.3 Accuracy

We generated another one million quad-double random numbers and computed the square root and the n-th root of a quad-double number with each algorithm. We then compared them to 63 decimal digits, which is the number of significant digits of a quad-double number. **Table 4** shows the concordance rate of the results of both algorithms. In case of the square root, 97.3% of the results by the proposed algorithm correspond exactly to the existing algorithm. If we represent all the results by the algorithm in the QD package as the vector $\boldsymbol{q}_{(qd)}$ and those by the proposed algorithm as the vector $\boldsymbol{p}_{(qd)}$, then $\|\boldsymbol{q}_{(qd)} - \boldsymbol{p}_{(qd)}\|_\infty$ is 1.396×10^{-63}. That is, even if the results differ, the difference occurs at most in the 63rd digit in decimal. For the n-th root, more than 95% of the results are the same ($3 \leq n \leq 10$). These results confirm that the proposed algorithms lead to an approximation that has almost the same accuracy as the existing algorithms.

Table 4 Concordance rate of both results to 63 decimal digits.

n	Ratio
2	97.3%
3	96.5%
4	96.8%
5	96.2%
6	95.9%
7	95.6%
8	95.9%
9	95.4%
10	95.1%

6. Conclusion

We proposed two new algorithms, one each for computing the square root and the n-th root of a quad-double number using a fourth-order iterative scheme. The proposed algorithm for the square root requires about half the number of double-precision arithmetic operations as for the existing algorithm in the QD package, and that for the n-th root, 56% to 62%.

We compared the computation time and the accuracy between the algorithms. The proposed algorithms for the square root and the n-th root are 1.59 to 1.77 times faster than the existing algorithm. Of the results of the square root, 97.3% are the same to 63 significant decimal digits. In case of the n-th root ($3 \leq n \leq 10$), more than 95% of the computation results of the proposed algorithm are the same as from the existing algorithm. Thus the proposed algorithm has almost the same accuracy as does the algorithm in the QD package.

The proposed algorithms are fast and accurate, and we conclude that they are an improvement over the existing algorithm in the QD package. Quad-double division is also based on the Newton's method, so a similar faster algorithm is expected. This and executing the current proposed algorithms in parallel to accelerate them, are future topics of study. Acceleration by specific hardware such as FPGA is also a future topic of study.

Acknowledgments The author would like to thank the anonymous reviewers for their useful comments.

References

[1] Bailey, D.H.: QD (C++ / Fortran-90 double-double and quad-double package), available from ⟨http://crd.lbl.gov/~dhbailey/mpdist/⟩

[2] Bailey, D.H. and Borwein, J.M.: Ancient Indian square roots: An exercise in forensic paleo-mathematics, *Amer. Math. Monthly*, Vol.119, No.8, pp.646–657 (2012).

[3] Dekker, T.J.: A floating-point technique for extending the available precision, *Numer. Math.*, Vol.18, pp.224–242 (1971).

[4] Dubeau, F.: Newton's method and high-order algorithms for the nth root computation, *J. Comput. Appl. Math.*, Vol.224, No.1, pp.66–76 (2009).

[5] Ford, W.F. and Pennline, J.A.: Accelerated convergence in Newton's method, *SIAM* Review, Vol.38, pp.658–659 (1996).

[6] Gerlach, J.: Accelerated convergence in Newton's method, *SIAM* Review, Vol.36, pp.272–276 (1994).

[7] Hernández, M.A. and Romero, N.: Accelerated convergence in New-

ton's method for approximating square roots, *J. Comput. Appl. Math.*, Vol.177, No.1, pp.225–229 (2005).

[8] Hida, Y., Li, X.S. and Bailey, D.H.: Quad-double arithmetic: Algorithms, implementation, and application, Technical Report LBNL-46996, LBNL, Berkeley, CA 94720 (2000).

[9] Knuth, D.E.: *The Art of Computer Programming*, Vol.2, Addison Wesley (1969).

[10] Saito, T.: MuPAT (Multiple Precision Arithmetic Toolbox), available from ⟨http://www.mi.kagu.tus.ac.jp/qupat.html⟩

[11] Saito, T., Ishiwata, E. and Hasegawa, H.: Development of quadruple precision arithmetic toolbox QuPAT on Scilab, *Proc. ICCSA 2010*, Part II, LNCS 6017, pp.60–70 (2010).

Tsubasa Saito received his M.S. degree from Tokyo University of Science in 2011. His research interest is high-precision arithmetic for numerical linear algebra. He implemented a quadruple precision arithmetic environment QuPAT and a multiple precision arithmetic environment MuPAT as toolboxes on Scilab. He is the Grand Prize winner of Scilab Toolbox Japan Contest 2009 for the student section.