Development of quadruple precision arithmetic toolbox QuPAT on Scilab

Tsubasa Saito[1], Emiko Ishiwata[2], and Hidehiko Hasegawa[3]

[1] Graduate School of Science, Tokyo University of Science, Japan
[2] Tokyo University of Science, Japan
[3] University of Tsukuba, Japan

Abstract. When floating point arithmetic is used in numerical computation, cancellation of significant digits, round-off errors and information loss cannot be avoided. In some cases it becomes necessary to use multiple precision arithmetic; however some operations of this arithmetic are difficult to implement within conventional computing environments. In this paper we consider implementation of a quadruple precision arithmetic environment QuPAT(Quadruple Precision Arithmetic Toolbox) using the interactive numerical software package Scilab as a toolbox. Based on Double-Double (DD) arithmetic, QuPAT uses only a combination of double precision arithmetic operations. QuPAT has three main characteristics: (1) the same operator is used for both double and quadruple precision arithmetic; (2) both double and quadruple precision arithmetic can be used at the same time, and also mixed precision arithmetic is available; (3) QuPAT is independent of which hardware and operating systems are used. Finally we show the effectiveness of QuPAT in the case of analyzing a convergence property of the GCR(*m*) method for a system of linear equations.

Keywords: quadruple precision arithmetic, mixed precision, Scilab

1 Introduction

Floating point arithmetic operations governed by IEEE754 are mainstream on conventional computers. But in floating point arithmetic, we cannot avoid cancellation of significant digits, round-off errors and information loss. In the case of double precision floating point numbers, there are approximately 16 (decimal) significant digits. Therefore, when a system of linear equations is solved by some iterative method, it is known that stagnation of the residual norm may occur or that the numerical solution may not converge. To reduce these errors, we need to use multiple precision arithmetic. However, it is difficult to implement multiple precision arithmetic in ordinary computing environments without any special hardware.

We consider quadruple precision arithmetic as multiple precision arithmetic. There is a proposal called Double-Double (DD) for quadruple precision arithmetic. DD is based on the algorithm for error-free floating point arithmetic by Dekker[2] and Knuth[5]. A DD number is represented by two double precision floating point numbers. QD[1] and Lis[6] are implementations of DD arithmetic in C and C++. In addition, although it is not standard, many Fortran compilers have a quadruple precision real number type 'REAL(KIND=16)' together with operations on such. The computational cost is high, however, if we cannot use special hardware. On the other hand, if we do use it, then code should be rewritten to use quadruple precision arithmetic hardware or library (except for Fortran). The rewritten code does not execute in the previous environment (i.e. without any special hardware or library), and rewritten code is difficult to debug.

However, there does exist an interactive numerical software package Scilab[10]. Scilab is similar to MATLAB, and, moreover, this is free and open source software. In this paper we implement a quadruple precision arithmetic environment QuPAT (Quadruple Precision Arithmetic Toolbox) using the interactive numerical software package Scilab as a toolbox.

In Scilab, we can define a new data type and apply operator overloading. Thus, we define a new data type representing a DD number to expand the double precision arithmetic environment. Using operator overloading, we can use the same operator for both double precision arithmetic and quadruple precision arithmetic. We can also use both double precision arithmetic and quadruple precision arithmetic at the same time, and thus we can make use of mixed precision arithmetic. QuPAT is implemented only using Scilab functions, so that QuPAT is independent of hardware and operating systems. Therefore, Scilab users can easily make use of QuPAT anywhere that it is required.

This paper is organized as follows. Section 2 presents some algorithms for error-free double precision floating point arithmetic and DD arithmetic. In Section 3, we describe the way to construct a DD environment on Scilab, the characteristics of QuPAT, and the computational time for DD arithmetic. In Section 4 we show the effectiveness of QuPAT for analyzing a convergence property of the GCR(m) method for a system of linear equations. Section 5 presents a summary and discusses future work.

2 DD arithmetic

We first explain DD arithmetic, which is based on representation using two double precision floating point numbers and defining four error-free floating point arithmetic algorithms [2, 5]. Specifically we explain the characteristics and four arithmetic operations of DD.

2.1 Characteristics of DD arithmetic

A DD number is represented using two double precision floating point numbers. A real number α is represented as the DD number $A = (Ahi, Alo)$, which is defined below:

$$Ahi = (\ \alpha \text{ rounded to a double precision number})$$
$$Alo = (\ (\alpha - Ahi) \text{ rounded to a double precision number})$$

Figure 1 shows the constitution of each DD number in bits and an IEEE754 quadruple precision floating point number. A DD thus contains two sign bits and a pair of 11 bit sequences for the exponent parts. But the sign and the exponent part of a DD number depend only on Ahi. The mantissa of a DD number contains in total 104 bits. This is 8 bits less than is required for an IEEE quadruple precision number. A double precision number d can be transformed to a DD number by setting $Ahi = d$, $Alo = 0$. For arithmetical operations on DD numbers, if the computational result doesn't fit into a DD number, then it is rounded to a DD number.

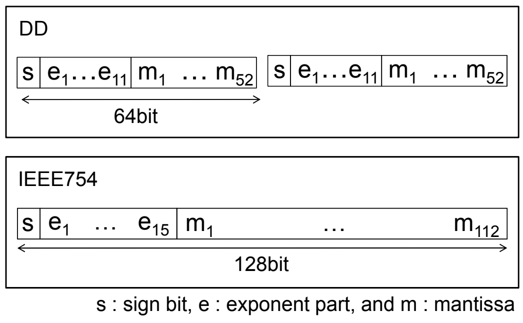


Figure 1 : Constitution of bits (DD number and IEEE754 quadruple precision number)

2.2 Algorithm for DD arithmetic

For implementation of DD arithmetic, double precision arithmetic should be computed exactly. However, a result of double precision arithmetic does not always fit into a double precision number. Therefore, we need to express such a number correctly using two double precision numbers. $f(\cdot)$ denotes a computation of double precision arithmetic, variables of small letter are double precision numbers, variables of capital letter are DD numbers. We present four algorithms (D1)～(D4) for error-free double precision arithmetic below (see [3, 4] for details).

Algorithm (D1) - two_sum.
The following algorithm computes double precision addition $s = f(a + b)$ and error $e = (a + b) - s$. In this way, the sum of two double precision floating point numbers is represented strictly: $a + b = s + e$.

two_ sum

```
$[s, e] = two\_sum\ (a, b)$
    $s = a + b;$
    $v = s - a;$
    $e = (a - (s - v)) + (b - v);$
$end$
```

Algorithm (D2) - fast_two_sum (for $|a| \geq |b|$).
The following algorithm is almost the same as (D1) except for the assumption that $|a| \geq |b|$.

fast_two_sum

```
$[s, e] = fast\_two\_sum\ (a, b)$
    $s = a + b;$
    $e = b - (s - a);$
$end$
```

Algorithm (D3) - split.
The following algorithm splits a double precision floating point number a into h and l where h contains the higher 26 bits of the mantissa of a, and l contains the lower 26 bits.

split

```
$[h, l] = split\ (a)$
    $t = 134217729 * a;$
    $h = t - (t - a);$
    $l = a - h;$
$end$
```

Algorithm (D4) - two_prod.
The following algorithm computes double precision multiplication $p = f(a \times b)$ and error $e = (a \times b) - p$. Using (D3), we get the following equality and algorithm :

$$a \times b = f(ah \times bh) + f(ah \times bl) + f(al \times bh) + f(al \times bl)$$

two_ prod

```
$[p, e] = two\_prod\ (a, b)$
    $p = a * b;$
    $(ah, al) = split\ (a);$
    $(bh, bl) = split\ (b);$
    $e = ((ah * bh - p) + ah * bl + al * bh) + al * bl;$
$end$
```

The four arithmetic operations of DD are defined only using double precision arithmetic numbers and their computation algorithms (D1)〜(D4). Let DD numbers A, B and C be (Ahi,Alo), (Bhi,Blo) and (Chi,Clo) , respectively.

Algorithm (DD1) - addition.
The following algorithm computes DD addition $A + B$.

addition

```
$C = add\ (A, B)$
    $[sh, eh] = two\_sum\ (Ahi, Bhi);$
    $[sl, el] = two\_sum\ (Alo, Blo);$
    $se = eh + sl;$
    $[sh', se'] = fast\_two\_sum\ (sh, se);$
    $see = se' + el;$
    $[Chi, Clo] = fast\_two\_sum\ (sh', see);$
$end$
```

Algorithm (DD2) - subtraction.
The following algorithm computes DD subtraction $A - B$ using DD addition.

subtraction

```
$C = sub\ (A, B)$
    $B^{*}hi = -Bhi;$
    $B^{*}lo = -Blo;$
    $C = add\ (A, B^{*});$
$end$
```

Algorithm (DD3) - multiplication.
The following algorithm computes DD multiplication $A \times B$ using the equality:

$$A \times B = Ahi \times Bhi + Ahi \times Blo + Alo \times Bhi + Alo \times Blo$$

However, we omit $Alo \times Blo$ from the computation to reduce the computation cost.

multiplication

```
$C = mul\ (A, B)$
    $[p1, p2] = two\_prod\ (Ahi, Ahi);$
    $p2 = p2 + Ahi * Blo;$
    $p2 = p2 + Alo * Bhi;$
    $[Chi, Clo] = fast\_two\_sum\ (p1, p2);$
$end$
```

Algorithm (DD4) - division.
The following algorithm computes DD division $(A \div B)$, assuming $B \neq 0$. DD division is based on Newton's method with an initial value $f(Ahi/Bhi)$.

division

```
$C = div\ (A, B)$
    $c = Ahi/Bhi;$
    $[p, e] = two\_prod\ (c, Bhi);$
    $cc = (Ahi - p - e + Alo - c * Blo)/Bhi;$
    $[Chi, Clo] = fast\_two\_sum\ (c, cc);$
$end$
```

Table 1 shows the number of double precision operations in the above algorithms, and Figure 2 shows the relationships among these algorithms.

3 Construction of DD environment on Scilab

Matlab and Scilab[10] are popular software packages for interactive numerical computation. They use double precision arithmetic, which can result in cancellation of sig-

Table 1 : Number of double precision operations

	number of each operation			
	add & sub	mul	div	total
fast_two_sum	3	0	0	3
two_sum	6	0	0	6
split	3	1	0	4
two_prod	10	7	0	17
add	20	0	0	20
sub	20	0	0	20
mul	15	9	0	24
div	17	8	2	27

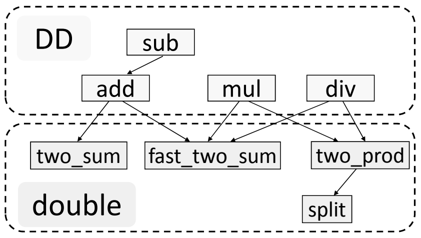


Figure 2 : Relationships among exact double precision arithmetic algorithms and DD algorithms

nificant digits, round-off errors and information loss. Scilab has almost the same capability as Matlab; however Scilab is open source and free. Scilab has been developed at Institut National de Recherche en Informatique et en Automatique (INRIA) in France.

In this paper, using DD, we construct a new environment for quadruple precision arithmetic QuPAT to reduce numerical errors, using Scilab as a toolbox.

3.1 Definition of data type for DD

Using Scilab, we can define a new data type using the Scilab function ‘tlist’. The function tlist creates a Scilab object and describes it as ‘tlist(typ, a1, ..., an)’, where ‘typ’ is our chosen name for the data type, and ‘a1, ..., an’ are elements. The values of a new data type are classified by its name. In this way we treat a data type using a combination of some data as a class in C++.

In Scilab, double precision numbers are defined by the data type named ‘constant’. Then, we define a new data type named ‘dd’ to contain DD numbers. To generate a value *a* as an element of the data type ‘dd’, we use the following code:

$$a = \text{tlist}\ ([\text{`dd'},\text{`hi'},\text{`lo'}],\ \text{ahi},\ \text{alo}\ ).$$

Thus, ‘[‘dd’,‘hi’,‘lo’]’ represents the name of the data type and its elements, and ‘ahi’ and ‘alo’ are double precision values in the constant data type. The name of the new data type is ‘dd’. To refer to the value of the higher (resp. lower) part ahi (resp. alo), we simply type ‘a.hi’ (resp. ‘a.lo’).

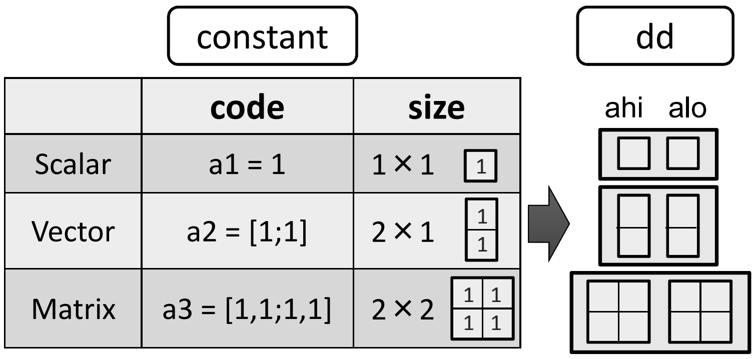
In QuPAT, we define the following function to generate a DD number :

```
function a = dd(ahi,alo)
  a = tlist(['dd','hi','lo'],ahi,alo);
endfunction
```

For example, we define ‘a = dd(1,0)’, then the valuable of dd type *a* becomes 1.

If we transform a value *a* from ‘constant’ into ‘dd’, we may use the function ‘d2dd’. On the other hand, if we transform a value *a* from ‘dd’ into ‘constant’, we may refer the variable ‘a.hi’.

In Scilab, scalars, vectors and matrices are treated in the same way as the data type ‘constant’ (Figure 3). Then, defining only the data type ‘dd’ enables elements to be expanded into DD numbers naturally.



	code	size
Scalar	a1 = 1	1 × 1
Vector	a2 = [1;1]	2 × 1
Matrix	a3 = [1,1;1,1]	2 × 2

Figure 3 : Relationship between ‘constant’ and ‘dd’

3.2 Definition of operators for DD

On Scilab, we can make use of operator overloading for a new data type defined by tlist. For computing values of a new data type, Scilab calls the preliminarily defined function. Allowing operator overloading for computing a value of ‘dd’ using (DD1)∼(DD4), we can use the same operator for quadruple precision arithmetic as double precision arithmetic. In particular, we only redefine functions named ‘%<first_ operand_ type>_<op_code>_<second_operand_type>’ or ‘ %<operand_type>_<op_code>’. We need to write the sequence of characters associated with each data type to ‘<first_ operand_ type>’ and ‘<second_ operand_type>’. Similarly, we should write a single character associated with each operator to ‘<op_code>’.

For example, a single character ‘a’ can be assigned to the operator ‘+’. For computing the sum of DD numbers *A* and *B* using the operator ‘+’, we only define the function named %dd_a_dd. In the same way, computing the sum of a DD number *A* and a double precision number *b* using the operator ‘+’, we define the function named %dd_a_s, where ‘s’ is the character associated with the data type constant. For computation of mixed precision arithmetic with ‘dd’ and ‘constant’, the value of ‘constant’ is expanded to ‘dd’ in the computation.

3.3 Definition of a function for DD

If a Scilab function is applied to an argument whose data type is ‘dd’ , then an error occurs. Hence, we define new functions named ‘dd<function_name>’ for calling with the type dd. For example, we create a new function ‘ddsqrt’ to be applied to a variable of type ‘dd’, corresponding to the Scilab function ‘sqrt’ to compute a square root.

3.4 Characteristics of QuPAT

In QuPAT (Quadruple Precision Arithmetic Toolbox), the data types ‘dd‘ and ‘constant’ are defined separately. Thus we can use both double precision arithmetic and quadruple precision arithmetic in the same code. In addition, we can apply mixed precision arithmetic using the same operator (+, −, ∗, /) with QuPAT. To convert a value of type ‘constant’ into one of type ‘dd’, we assign 0 to the lower part of dd. In contrast, to convert a value of ‘dd’ into ‘constant’, we extract the higher part of dd. As a result, users of Scilab can use quadruple precision arithmetic with QuPAT in the same way as ordinary Scilab double precision arithmetic. Figure 4 shows the relationship between types ‘constant’ and ‘dd’ in Scilab. QuPAT is implemented using pure Scilab functions, hence QuPAT is independent of hardware and operating systems.

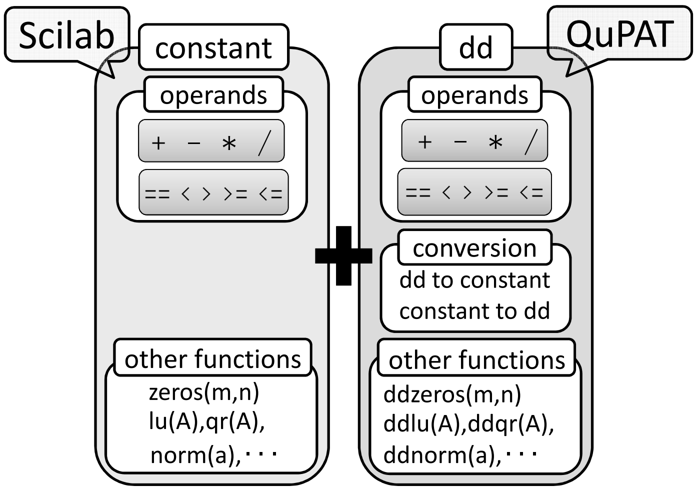


Figure 4 : Relationship between Scilab and QuPAT

3.5 Computational time for DD arithmetic

Computation was carried out on a PC with an AMD Turion(tm) X2 Dual-Core 2.00GHz and Scilab version 5.1.1. Table 2 shows the computation time in seconds and the ratio of time required for DD arithmetic to time required for double precision arithmetic. Each result is the average over five trials, and N is the number of repetitions in the loop.

Computation time for DD arithmetic is about 9 to 15 times greater than that for double precision arithmetic.

4 Effectiveness of DD arithmetic for analyzing the GCR(m) method

The GCR (Generalized Conjugate Residual) method is one of the the Krylov subspace methods to solve a nonsymmetric linear system $A\boldsymbol{x} = \boldsymbol{b}$. The GCR method is based on Arnoldi process and the minimal residual approach. In addition, the GCR method has the theoretical property that the residual norm decreases at each iteration and converges

Table 2 : Computation time in seconds; the ratio is in parentheses

		computation time and ratio			
	N	addition	subtraction	multiplication	division
double	50,000	1.02	1.21	1.13	1.28
	100,000	2.08	1.92	1.98	1.95
	500,000	9.86	9.78	10.72	9.88
	1,000,000	19.24	20.11	19.69	19.25
DD	50,000	9.80 (9.62)	10.72 (8.87)	13.25 (11.74)	13.41 (10.49)
	100,000	19.91 (9.58)	21.12 (10.97)	28.00 (14.11)	29.00 (14.85)
	500,000	104.77 (10.63)	108.42 (11.08)	137.38 (12.82)	141.06 (14.28)
	1,000,000	207.74 (10.80)	218.23 (10.85)	278.83 (14.16)	280.11 (14.55)

after at most n iterations, where n is the dimension of the matrix A. But using floating point arithmetic, it is known that stagnation of the residual norm may occur and that the numerical solution may not converge.

In this section, we investigate numerical solution by the GCR(m) method where m is the restart cycle, comparing the results of using double precision arithmetic versus DD arithmetic. The difference in the code between double precision arithmetic and DD arithmetic is only in the definition of the variables and the name of the functions to compute a norm and an inner product. All experiments were carried out in the same computational environment as in Section 3.5.

We consider 'arc30' for the matrix A from the MatrixMarket [7]. The dimension of this matrix is $n = 130$, and its condition number is 6.05×10^{10}, obtained using the Scilab function 'cond'. We set the restart cycle at $m = 50$, and GCR(m) in double and DD arithmetic was terminated at 1000 iterations if convergence did not occur. The iteration was started with $\boldsymbol{x}_0 = \boldsymbol{0}$ and the right-hand side vector $\boldsymbol{b}$ was given by substituting the solution vector $\boldsymbol{x}^* = (1, 1, ..., 1)^T$ into $\boldsymbol{b} = A\boldsymbol{x}^*$. The stopping criteria are given below:

$$||\boldsymbol{r}_k||_2 \le 10^{-12}\ ||\boldsymbol{r}_0||_2 \qquad \text{(for double)}$$
$$||\boldsymbol{r}_k||_2 \le 10^{-18}\ ||\boldsymbol{r}_0||_2 \qquad \text{(for DD)}$$

Table 3 and Figure 5 show the numerical results. In the case of double precision arithmetic, the relative residual norm stagnated at about 1.0×10^{-10} and the solution did not converge. In addtion, the error norm was 9.27×10^0 at 1000 iterations. Because $n = 130$, the residual norm should theoretically converge after 130 iterations. On the other hand, using DD arithmetic that has about twice the number of significant digits, the relative residual norm became 9.89×10^{-19} and the error norm became 2.74×10^{-8} in 18 iterations. This is a great improvement.

Table 3 : Iteration counts, relative residualnorm and relative error norm.

	Iteration counts	$\lVert\boldsymbol{r}\rVert_2/\lVert\boldsymbol{r}_0\rVert_2$	$\lVert\boldsymbol{x}-\boldsymbol{x}^*\rVert_\infty/\lVert\boldsymbol{x}^*\rVert_\infty$
double	1000	6.28e-11	9.27e+00
DD	18	9.89e-19	2.74e-08

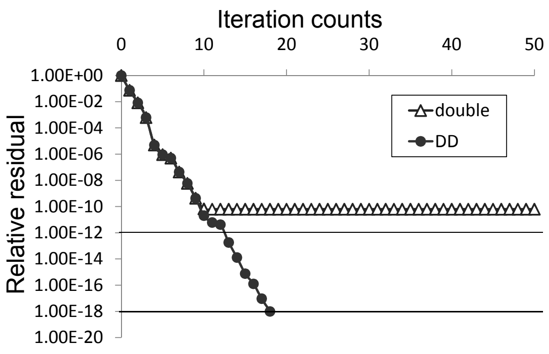


Figure 5 : Convergence history

It is clear that the reason for the differences is computational error. The error norm did not decrease using double precision arithmetic. Because a double precision floating point number has about 16 (decimal) significant digits, it is difficult to obtain a solution with sufficient accuracy for a system whose condition number is 6.05×10^{10}. In addition, the residual norm stagnates due to computational error when using double precision arithmetic. However, there are situations where the theory-based result for reducing computational error using DD arithmetic does hold. We also intend to analyze the implementation of these iterative algorithms using QuPAT.

5 Conclusion

To examine certain computational results of double precision arithmetic, we need a higher precision arithmetic environment. For example, in Section 4, in solving a system of linear equations by the GCR(m) method, the relative residual norm may stagnate and may not converge in double precision arithmetic; i.e., it is not possible to obtain the solution with sufficient accuracy. As a useful way to employ quadruple precision arithmetic, Double-Double(DD) is proposed. However, in programming languages such as C, we cannot set up quadruple precision arithmetic easily.

In this paper, we constructed a convenient quadruple precision arithmetic environment QuPAT(Quadruple Precision Arithmetic Toolbox) using Scilab as a toolbox. As a consequence, we were able to use quadruple precision arithmetic without rewriting the code in Scilab, and also utilize mixed precision arithmetic.

As in section 3.1, we defined a new data type ‘dd’ for quadruple precision numbers using the Scilab function ‘tlist’. In QuPAT, a new data type ‘dd’ and the existing data type ‘constant’ were defined separately. Thus it became possible to utilize both double and quadruple precision arithmetic in the same code.

In addition, we applied operator overloading to the type ‘dd’ for the four fundamental rules of DD arithmetic. Thus it became possible to use all of the double, quadruple and mixed precision arithmetic with the same operators ($+, -, *, /$). The Scilab environment was naturally extended to use DD arithmetic with QuPAT. The computation time for DD arithmetic was about 9 to 15 times that for double precision arithmetic.

QuPAT was implemented using pure Scilab functions. Therefore, if Scilab is available, we can utilize QuPAT independently of underlying hardware and operating systems. QuPAT is downloadable provisionally from the web [9].

Until now, some other important functions, such as sine or cosine were not implemented. In addition, if we use Scilab capability link to functions written in C or Fortran, QuPAT will execute faster. However, if we do use this capability, the code will depends on the computing environments and programming languages. These points will be discussed in future work.

References

[1] D. H. Bailey, QD (C++ / Fortran-90 double-double and quad-double package), Available at http://crd.lbl.gov/˜dhbailey/mpdist/

[2] T. J. Dekker, A Floating-Point Technique for Extending the Available Precision, Numer. Math. Vol. 18, 224-242(1971) .

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

[4] Y. Hida, X. S. Li and D. H. Bailey, Algorithms for Quad-Double Precision Floating Point Arithmetic, Proceedings of the 15th IEEE Symposium on Computer Arithmetic, 155-162(2001).

[5] D. E. Knuth, The Art of Computer Programming, vol. 2. Addison Wesley (1969).

[6] H. Kotakemori, A. Fujii, H. Hasegawa and A. Nishida, Stabilization of Krylov subspace methods using fast quadruple-precision operation, Transactions of JSCES Vol. 12, 631-634(2007), in Japanese.

[7] MatrixMarket, http://math.nist.gov/MatrixMarket/

[8] N. Nakasato, T. Ishikawa, J. Makino and F. Yuasa, Fast Quad-Precision Operations On Many-core Accelerators, IPSJ SIG Technical Report(2009), in Japanese.

[9] QuPAT, http://www.mi.kagu.tus.ac.jp/qupat.html

[10] Scilab, http://www.scilab.org/