Scilabにおける高精度演算環境MuPATの実装

吉川慧子 *，齊藤翼 *，石渡恵美子 **，長谷川秀彦 ***

Implementation of Multiple Precision Arithmetic Toolbox on Scilab

Satoko Kikkawa*, Tsubasa Saito*, Emiko Ishiwata** and Hidehiko Hasegawa***

抄録

浮動小数点演算では，丸め誤差の影響は避けられず，演算結果の検証などには現在主流の倍精度演算よりも高精度な演算が必要となる．そこでわれわれは，高精度演算環境MuPAT(Multiple Precision Arithmetic Toolbox)を対話型数値計算ソフトウェアScilab上に実装した．MuPATでは，倍精度，4倍精度，8倍精度演算で共通の演算子や関数を使用できる．プログラムの変更が最小限で済むため高精度演算への切り替えが簡単にでき，部分的な高精度化や混合精度演算も可能である．また，C言語の外部関数を利用することで，演算の高速化を達成している．外部関数を利用したMuPATの高精度演算は，Scilabの機能のみの実装よりも180～1200倍高速である．

今回，新たに疎行列形式を導入し，より少ないメモリと実行時間で高精度演算が利用できるようになった．8倍精度演算の場合，行列の非零要素率が80%以下の場合，使用メモリ量は密行列データ型よりも小さくなる．行列ベクトル積では密行列データ型の1～2倍，行列加算では7～21倍，行列積では83～900倍高速に実行できる．本稿では，8倍精度演算を中心に，Scilabにおける高精度演算の具体的な実装方法，外部関数を利用した高速化，疎行列形式への対応について詳細を報告する．

Abstract

To analyze errors in numerical computation, easily usable high precision arithmetic is important. We have developed a quadruple and octuple precision arithmetic toolbox named MuPAT(Multiple Precision Arithmetic Toolbox) on Scilab. Using MuPAT, the users can use the same operators and functions to double, quadruple and octuple precision numbers and mixed precision arithmetic is also available. We have also developed a high speed implementation with external C routines. MuPAT with external routines can perform 180～1200 times faster than MuPAT without external routine.

MuPAT has sparse data structure for quadruple and octuple precision and the users can treat large matrices with lower memory consumption and small computation time. If the sparsity is less than 80%, the memory consumption of sparse data structure for octuple precision is smaller than that of dense data structure. Compared with dense data structure, the computation speed of octuple precision arithmetic with sparse data structure is 1～2 times faster at matrix vector product, 7～21 times faster at matrix addition and 83～900 times faster at matrix multiplication. In this paper, we describe the detail of the basic implementation method and acceleration of MuPAT, and the implementation method of sparse data structures on Scilab.

* 東京理科大学大学院理学研究科
Graduate School of Science, Tokyo University of Science
** 東京理科大学理学部
Faculty of Science, Tokyo University of Science
*** 筑波大学図書館情報メディア系
Faculty of Library, Information and Media Science, University of Tsukuba

1　はじめに

科学技術計算では，表現できる数の範囲が広い浮動小数点演算を用いることが多く，現在では倍精度演算が主流である．しかし，浮動小数点演算では丸め誤差などの計算誤差が避けられない．誤差による計算結果への影響を検証したり，倍精度演算では十分な精度の解が得られない問題を解くためには，倍精度演算よりも高精度な演算が必要である．倍精度演算のみを用いて高精度演算を実現する手法である DD(擬似 4 倍精度)[3] と QD(擬似 8 倍精度)[4] を利用した 4 倍精度，8 倍精度演算環境 MuPAT (Multiple Precision Arithmetic Toolbox) をフリーの対話型数値計算ソフトウェア Scilab[13] 上に実装した [8, 11]．MuPAT では，倍精度，4 倍精度，8 倍精度演算に共通の演算子 (+, -, *, /) が利用でき，高精度演算のみでなく混合精度演算も行える．

一方，QD は実行に必要な倍精度演算回数が多く，さらに倍精度数の約 4 倍ものメモリを消費する．そのため，大規模行列に対しては，実行時間やメモリ使用量の観点から QD 演算の適用は困難だった．そこで今回，MuPAT の特徴は維持したまま疎行列形式を導入し，大規模行列に対しても高精度演算が適用できるようにした．

本稿では，MuPAT の具体的な実装方法を述べる．はじめに，DD 演算と QD 演算を紹介し，その後，MuPAT の基本的な実装方法，外部関数を利用した MuPAT の高速化，疎行列向け高精度演算の実装方法について，順に述べる．

2　Double-Double と Quad-Double

DD(Double-Double) は 2 つの倍精度数を組み合わせて 4 倍精度数を，QD(Quad-Double) は 4 つの倍精度数を組み合わせて 8 倍精度数を表現する手法のことである．例えば，QD の数 A は 4 つの倍精度数 a_0, a_1, a_2, a_3 を用いて以下のように表現される．

$$A = a_0 + a_1 + a_2 + a_3$$

ただし，a_0, a_1, a_2, a_3 の間には次の関係が成り立つ．

$$|a_{i+1}| \leq \frac{1}{2}\mathrm{ulp}(a_i), \quad i = 0, 1, 2,$$

ここで ulp は units in the last place の略で，その浮動小数点数で表現可能な最小単位のことである．最上位項の a_0 は QD の数 A の倍精度数への近似であり，その誤差は $\mathrm{ulp}(a_0)$ の半分以下である．DD の数は 10 進で 31 桁，QD の数は 10 進で 63 桁の有効桁数を持つ．DD 演算と QD 演算は倍精度演算のみを利用した，エラーフリーな浮動小数点アルゴリズムの組み合わせで行われる．そのため DD 演算と QD 演算は，倍精度演算環境さえあれば実装できる．DD 演算と QD 演算の詳細は Appendix A に示す．以降，QD の数 A を $A = (a_0, a_1, a_2, a_3) = a_0 + a_1 + a_2 + a_3$ として表記する．

3　MuPAT

MuPAT(Multiple Precision Arithmetic Toolbox) [8, 11] は，DD 演算と QD 演算を利用した 4 倍精度と 8 倍精度からなる高精度演算環境であり，Scilab[13] のツールボックスとして実装されている．

Scilab[13] とは，フランスの研究機関 INRIA が開発したオープンソースの対話型数値計算ソフトウェアである．Matlab とほぼ同等の機能を持ち，線形代数演算が簡単に記述できるという特徴を持つ．例えば，行列 A と B の加算は $A+B$ のように記述するだけで実行できる．また，Scilab では型や関数を独自に定義でき，同一の演算子記号や関数名を複数定義するオーバーロード機能があるため，独自に定義した型に対する演算や関数を組み込みの演算や関数と同等に扱うことができる．MuPAT の実装では，Scilab の簡便さを維持したまま高精度演算を実現するために，これらの Scilab の特徴を利用することとした．

MuPAT では，演算子や関数をオーバーロードしているため，倍精度，4 倍精度，8 倍精度演算で共通の演算子や関数が使える．そのため，ユーザーが高精度演算を使用する際に，変数の定義以外はプログラムをほとんど書き換える必要がなく，Scilab の特徴であるプログラムの書きやすさを維持している．さらに，すべての精度の演算を同時に扱えるため，部分的な高精度化や混合精度演算もできる．まとめると MuPAT は以下の特徴を持つ．

- 倍精度，4 倍精度，8 倍精度演算，それらの混合精度演算が同時に利用できる．
- 共通の演算子 (+, -, *) を使える．

本節では，QD 演算の Scilab への実装方法を中心に述べる．DD 演算の実装については，文献 [10, 12] に詳細がある．

3.1　Scilab における倍精度変数

Scilab では，実数の定数はすべて `constant` 型として扱われる．`constant` 型は通常のプログラミング言語の倍精度数 'double' に対応する Scilab のデータ型だが，可変長の配列のようなデータ型であるため，ベクトルや行列もスカラーと同様に扱われるという特徴を持つ．行列やベクトルを扱う場合には角括弧 `[ ]` で囲んで以下のように入力する．データ型の宣言は必要ない．

```
-->A1 = 1           //スカラー
 A1  =
     1.
-->A2 = [1;2]       //ベクトル
 A2  =
     1.
     2.
-->A3 = [1,2;3,4] //行列
 A3  =
     1.    2.
     3.    4.
```

このように，スカラーはサイズが 1×1，m 次元ベクトルはサイズが $m \times 1$，$m \times n$ 行列は $m \times n$ の `constant` 型の変数として定義される．

3.2　QD の数の定義

本節では，QD の数に相当する新しいデータ型の定義について述べる．

Scilab では tlist という関数を利用することで，新しいデータ型を定義できる．Scilab のヘルプによると，tlist 関数は，

```
a = tlist(["listtype","field1","field2",
     ... , "fieldn"], a1, a2, ..., an);
```

のように記述する．`"　"` で囲まれる変数には，文字列を代入する．ここで 1 番目の引数は文字列を要素に持つベクトルであり，`"listtype"` はユーザーが定義するデータ型の名前となる．`"field1"`, `"field2"`, ... , `"fieldn"` では，2 番目以降の要素 `a1`, ... ,`an` に独自の名前を指定できる．`a1`, ... , `an` は新しいデータ型を構成する要素であり，行列やリスト，文字列などを格納できる．`"fieldn"` を記述した場合，`a.fieldn` と入力することで `an` を参照できる．

MuPAT では，tlist 関数を利用し，Scilab の倍精度データ型 '`constant`' を組み合わせることで，2 節で述べた QD の数の構成に基づいた新しいデータ型 '`qd`' を定義している．具体的には以下のように記述している．

```
a = tlist(['qd','d'], a0,a1,a2,a3);
```

ここで，`a0,a1,a2,a3` は `constant` 型の変数であり，2 節の a_0, a_1, a_2, a_3 に対応している．Scilab では文字列を `'　'` で囲んで定義しており，`'qd'` は新たなデータ型の名前，`'d'` は `a0` の名前を示している．`a0` の値を参照したい場合には，`a.d` と入力する．`a(i)` と入力すると i 番目の要素を参照できる．MuPAT では，`qd` 型の変数を生成するための以下のような関数を用意している．

```
function a = qd(a0,a1,a2,a3)
   a = tlist(['qd','d'],a0,a1,a2,a3);
endfunction
```

これにより，`a = qd(a0, a1, a2, a3)` と入力するだけで `qd` 型の変数 `a` を生成できる．実際は，4 つの引数すべてを入力しなくても `qd` 型の変数が生成されるように実装している．例えば，`qd(2)` のように引数を 1 つとした場合，`a0` に 2 が代入され，残りの `a1`, `a2`, `a3` には自動的に 0 が代入される．3.1 節で述べたように，`qd` 型を構成する `constant` 型はスカラーだけでなく行列やベクトルを扱えるため，`qd` 型もスカラー，ベクトル，行列を扱うことができる．$m \times n$ 行列の `qd` 型変数を定義したい場合は，`a0`, `a1`, `a2`, `a3` それぞれに $m \times n$ 行列を代入する．図 1 に QD の数を成分に持つ 4×4 行列 $A = A0 + A1 + A2 + A3$ を `qd` 型で定義した例を示す．$A0$ には，A の各成分 a, b, c, d の QD の数の最上位項から成る 4×4 行列が入り，$A1, A2, A3$ にも QD の数の第 2 項，第 3 項，第 4 項の行列がそれぞれに入る．

3.3　QD 演算の定義

Scilab では tlist で作成した新規のデータ型に対し，演算子のオーバーロードができる．`qd` 型に関

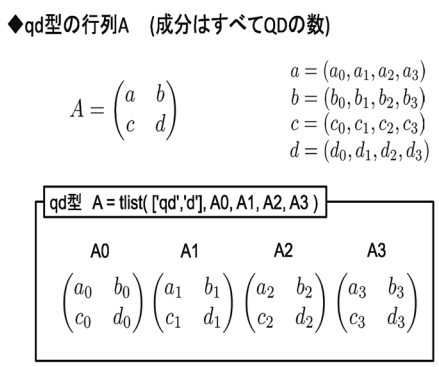

図 1: 2×2 行列を qd 型で定義した場合

する演算に，演算子 (+, -, *, /) が使えるようにそれぞれの演算子をオーバーロードし，QD 演算アルゴリズムが実行されるように実装している．演算子のオーバーロードは，二項演算の場合は，

```
%<1st_op_type>_<op_code>_<2nd_op_type>,
```

単項演算の場合は，

```
%<op_type>_<op_code>
```

と関数名を定めることで実現される．<1st_ op_ type>, <2nd_op_type>, <op_type> にはその演算に関係するデータ型の名前を記述する．constant 型のときのみ Scilab で指定された文字コード‘s’を使用する．<op_code> には，Scilab で設定された演算子を示す文字コードを記述する．表 1 に<op_code> と演算子の対応を示す．

表 1: 演算子に対応する文字コード

演算子	<op_code>
+	a
-	s
*	m
/	r

オーバーロードの具体的な例として，qd 型変数同士の加算に演算子‘+’を適用する．<1st_op_type> と<2nd_op_type> は qd である．<op_code> は表 1 より a である．よって，関数名を%qd_a_qd とし，関数の中に QD の加算アルゴリズムを記述すればよい．MuPAT では，以下のように qd 型変数同士の加算を実装している．

```
function c = %qd_a_qd(a,b)
  [s1,t1] = two_sum(a(2),b(2));
  [s2,t2] = two_sum(a(3),b(3));
  [s3,t3] = two_sum(a(4),b(4));
  [s4,t4] = two_sum(a(5),b(5));
  [s2,t1] = two_sum(s2,t1);
  [s3,t1,t2] = three_sum(s3,t2,t1);
  [s4,t1] = three_sum2(s4,t3,t1);
  s5 =t1+t2+t4;
  [t1,t2,t3,t4]
      = renormalize(s1,s2,s3,s4,s5);
  c = tlist(['qd','d'],t1,t2,t3,t4);
endfunction
```

このように記述した関数を，関数名と同一のファイル名で拡張子が.sci の.sci ファイルに格納する．演算実行時には.sci ファイルが呼び出され演算が実行される．これにより，qd 型の変数 a と b の加算は，演算子‘+’を使って $a+b$ と入力するだけで実行できる．

表 2 に加算に関する関数を示す．MuPAT では，dd 型，qd 型の四則演算や倍精度，4 倍精度，8 倍精度の混合精度演算に関し，このような関数を 32 個実装している．これにより，二項演算に関する変数 a, b がどのデータ型であっても，四則演算子 (+, -, *, /) を使って実行できる．

$$a \bullet b \quad \left(\begin{array}{l} a, b \in \{\texttt{constant}, \texttt{dd}, \texttt{qd}\} \\ \bullet \in \{+, -, *, /\} \end{array} \right)$$

混合精度演算の結果は，二項のうち，より精度の良い型へ拡張される．同様にして，関係演算子についてもオーバーロードをしている．dd 型と qd 型の関

表 2: 加算に関する関数

関数名	オーバーロードされる演算		
%s_a_dd	constant 型	+	dd 型
%dd_a_s	dd 型	+	constant 型
%s_a_qd	constant 型	+	qd 型
%qd_a_s	qd 型	+	constant 型
%dd_a_dd	dd 型	+	dd 型
%dd_a_qd	dd 型	+	qd 型
%qd_a_dd	qd 型	+	dd 型
%qd_a_qd	qd 型	+	qd 型

係演算子には，‘==, <, >, <=, >=, ～=’が使用できる．結果は%T または%F で返される．

3.4　QD の数に対する関数の定義

Scilab には，constant 型用に初等関数が多く用意されている．これらはそのままでは，独自に定義した qd 型を引数に持つことはできない．そこで，演算子と同様に関数に対してもオーバーロードをする．関数をオーバーロードするには，関数名を

```
%<op_type>_<function_name>
```

と定めた関数を実装する．<op_type> にはその関数の引数となるデータ型の名前を記述し，<function_name> にはオーバーロードしたい関数の名前を記述する．例えば，constant 型を引数に持つ平方根を求める関数‘sqrt’を qd 型でも扱えるようにするため，%qd_sqrt(a) という関数名の関数を実装している．このような実装により，変数 a の平方根を求めたい場合，a が constant 型でも qd 型でも sqrt(a) と記述すればよい．

Scilab には，この他に行列を生成する関数なども用意されている．例えば，constant 型の $n \times n$ 単位行列を生成する関数 eye(n,n) があるが，このような関数は精度を決定する変数が引数にないためオーバーロードできない．MuPAT では，オーバーロードできない関数に対し，qd<function _name> という名前を持つ関数を用意している．表 3 に，MuPAT で実装した dd 型，qd 型の変数のための関数のリストを示す．これらは標準的なアルゴリズムを高精度数を用いて実装したものである．

4　外部関数を利用した高速化

表 4 に DD 演算と QD 演算を 1 回実行するために必要な倍精度演算回数を示す．QD 演算は多くの倍精度演算を必要とする．特に，QD の除算は 649 回もの倍精度演算が必要であり，Scilab で実行すると倍精度除算の数千倍の実行時間がかかる．高精度演算の高速化のため，MuPAT では外部関数を利用した高速な実装を用意した．

表 4: DD，QD 演算に必要な倍精度演算回数

		double precision			
		add & sub	mul	div	total
DD	add & sub	11	0	0	11
	mul	15	9	0	24
	div	17	8	2	27
QD	add & sub	91	0	0	91
	mul	171	46	0	217
	div	579	66	4	649

4.1　実装方法

Scilab では，link 関数と call 関数により C 言語や Fortran で書かれた外部関数を Scilab から呼び出して実行できる．MuPAT のユーザーインターフェースを変えずに高速な演算が利用できるようにするため，.sci ファイル中の演算の処理のみを抜き出し，C 言語で書かれた外部関数において QD 演算の高速化を図った．

外部関数を利用した MuPAT は，Windows，Mac OS，Linux に対応しているが，それぞれの OS により必要となるファイルが異なる．以降，qd 型変数 a, b の加算 $a + b$ の実装を例に，外部関数の実装方法について述べる．

4.1.1　Windows の場合

外部関数を利用した流れを図 2 に示す．.sci ファイルは call という関数を使い外部関数を呼び出すように書き換えられており，QD 演算は外部関数内で実行される．

Scilab では基本的に，データは行列として扱われる．外部関数を利用した実装では，データの先頭アドレスとサイズを渡して，外部関数にできるだけ多くの演算をさせるように設計している．例えば，qd 型変数同士の加算を実行する C 言語の関数 qd_a_qd は以下のように実装している．外部関数の作成には Microsoft Visual C++2010 を使用した．

```
extern "C"
_declspec(dllexport)
int qd_a_qd(double *c0, double *c1,
            double *c2, double *c3,
```

表 3: MuPAT の関数

	関数名	機能	引数	戻り値
定義と変換	dd(a0,a1)	dd 型の生成	constant	dd
	qd(a0,a1,a2,a3)	qd 型の生成	constant	qd
	d2dd(a)	constant 型の変換	constant	dd
	d2qd(a)	constant 型の変換	constant	qd
	dd2qd(A)	qd 型への変換	dd	qd
	qd2dd(A)	dd 型への変換	qd	dd
	getHi(A)	倍精度への丸め	dd, qd	constant
全精度共通の関数	abs(A)	絶対値	constant, dd, qd	constant, dd, qd
	norm(X,N)	X の N-ノルム		
	nrt(A,N)	N 乗根		
	sqrt(A)	平方根		
	ceil(A)	切り上げ		
	floor(A)	切り下げ		
	sin(A)	正弦		
	cos(A)	余弦		
	tan(A)	正接		
	lu(A)	LU 分解		
	qr(A)	QR 分解		
	A(i,j)	成分の取出・挿入		
	A’	転置		
その他の関数	ddexp()	ネイピアの数	-	dd
	qdexp()	ネイピアの数	-	qd
	ddeye(n,n)	単位行列	constant	dd
	qdeye(n,n)	単位行列	constant	qd
	ddzeros(m,n)	ゼロ行列	constant	dd
	qdzeros(m,n)	ゼロ行列	constant	qd
	ddip(X,Y)	内積	dd	dd
	qdip(X,Y)	内積	qd	qd
	ddGauss(A,B)	ガウスの消去法	dd	dd
	qdGauss(A,B)	ガウスの消去法	qd	qd
	ddprint(A)	出力	dd	-
	qdprint(A)	出力	qd	-

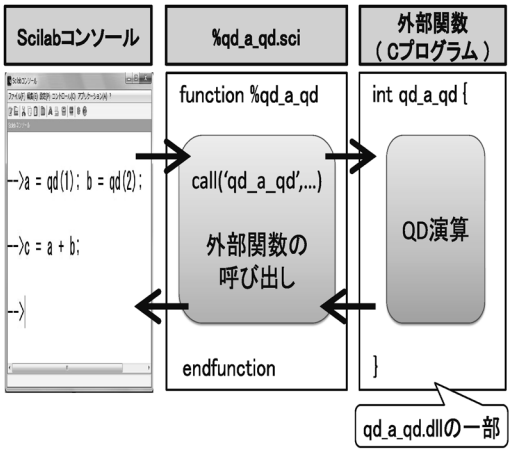

図 2: 外部関数利用の流れ

```
          double *a0, double *a1,
          double *a2, double *a3,
          double *b0, double *b1,
          double *b2, double *b3,
          int *m, int *n) {
    double e1,e2,e3,e4;
    e1 = 0.0; e2 = 0.0;
    e3 = 0.0; e4 = 0.0; int i;

    for(i=0; i<(*m)*(*n); i++) {
      c0[i] = 0.0; c1[i] = 0.0;
      c2[i] = 0.0; c3[i] = 0.0;
      two_sum(&c0[i], &e1, a0[i], b0[i]);
      two_sum(&c1[i], &e2, a1[i], b1[i]);
      two_sum(&c1[i], &e1, c1[i], e1);
      two_sum(&c2[i], &e3, a2[i], b2[i]);
      three_sum(&c2[i], &e1, &e2,
                    c2[i], e2, e1);
      two_sum(&c3[i], &e4, a3[i], b3[i]);
      three_sum2(&c3[i], &e1,
                    c3[i], e3, e1);
      e1 = e1 + e2 + e4;
      renormalize(&c0[i], &c1[i],
      &c2[i], &c3[i],
      c0[i], c1[i], c2[i], c3[i], e1);
    }
    return(0);
}
```

引数の`*a0`, `*a1`, `*a2`, `*a3`, `*b0`, `*b1`, `*b2`, `*b3` は Scilab から渡された QD の数 $a = (a0, a1, a2, a3), b = (b0, b1, b2, b3)$ のそれぞれの項であり，`*c0`, `*c1`, `*c2`, `*c3` は加算結果を返すための引数である．関数の引数はすべてポインタ渡しによって値の受け渡しをする．そのため，Scilab から渡す QD の数がベクトルや行列であっても，それぞれの項の先頭アドレスのみを渡せばよい．つまり，ベクトルや行列の成分ごとに外部関数を呼び出す必要はなく，1 回の呼び出しでベクトルや行列の演算ができるので，オーバーヘッドの削減になる．ただし，$m \times n$ 行列を Scilab から渡された場合，C 言語の関数内では $m \times n$ の一次元配列として扱わなければならない．また，Scilab のベクトルや行列のインデックスは 1 から始まるため，インデックスが 0 から始まる C 言語のプログラムに実装する際には注意が必要である．

Windows 用の MuPAT では，C 言語の外部関数をまとめたプログラムが予め.dll ファイルに変換されている．関数のヘッダ部分`_declspec(dllexport)`は，関数をエクスポートして.dll ファイルを生成するための構文である．

生成した.dll ファイルを call で呼ぶためには，事前に動的リンクを構成する関数 link により Scilab にリンクしておかなければならない．link 関数を利用するためには，

```
link("files" , "sub-names" , "flag");
```

と記述する．`"files"` にはリンクする関数が収められている.dll ファイルのファイル名，`"sub-names"` にはリンクする関数の関数名を記述する．`"flag"` には，関数が Fortran で書かれている場合は`'f'`，C 言語の場合は`'c'` と記述する．例えば，C 言語のプログラムで定義した qd_a_qd 関数が収まっている qd_a_qd.dll をリンクする場合，Scilab で

```
link('qd_a_qd.dll','qd_a_qd','c');
```

を実行する．

リンクしたファイルを Scilab で呼ぶために，call 関数を利用する．外部関数で実行した演算の結果を`[y1,...,yk]` に返すには，call 関数を

```
[y1,...,yk] =
  call("ident",x1,px1,"tx1",
          ...,xn,pxn,"txn",
       "out",[ny1,my1],py1,"ty1",
```

```
...,[nyl,myl],pyl,"tyl");
```

のように記述する．第一引数"ident" には呼び出す外部関数名を，x1,...,xn には外部関数に渡す変数をいれる．px1,...,pxn にはそれぞれの変数が外部関数の何番目の引数に渡されるのかを記す．例えば，x1 を外部関数の i 番目の引数に渡すとき，px1 には i を入れる．tx1,...,txn には外部関数における変数の型を，int の場合は i ，double の場合は d と指定する．"out" 以降は出力する変数 [y1,...,yk] のための引数である．[ny1,my1],...,[nyl,myl] は出力する変数の行列サイズ，py1,...,pyl は外部関数における位置，ty1,...,tyl で外部関数における変数の型を指定する．

外部関数を使用する際は，%qd_a_qd.sci ファイルを以下のように書き換える．

```
function c = %qd_a_qd(a,b)
  [m,n] = size(a(2));
  [c1,c2,c3,c4] =
    call('qd_a_qd',a(2),5,'d',a(3),6,'d',
           a(4),7,'d',a(5),8,'d',
           b(2),9,'d',b(3),10,'d',
           b(4),11,'d',b(5),12,'d',
           m,13,'i',n,14,'i',
           'out',[m,n],1,'d',[m,n],2,'d',
           [m,n],3,'d',[m,n],4,'d');
  c = tlist(['qd','d'],c1,c2,c3,c4);
endfunction
```

a(2), a(3), a(4), a(5) は qd 型変数 $a = (a0, a1, a2, a3)$ の 4 項である．b(2), b(3), b(4), b(5) は qd 型変数 $b = (b0, b1, b2, b3)$ の 4 項であり，m, n は a, b の行列サイズである．例えば，倍精度数で構成されるベクトル a(2) は，C 言語の外部関数 qd_a_qd の 5 番目の引数 int *a0 に値を渡すので，call 関数には a(2),5,'d' と記述する．また，$a + b$ の加算結果を qd 型変数 $c = (c0, c1, c2, c3)$ とすると，c の行列サイズは $m \times n$，$c0$ は $m \times n$ の倍精度の行列となるので，'out' の後に [m,n],1,'d' と記述する．

4.1.2　Mac OS，Linux の場合

Mac OS と Linux の場合，演算を記述した.c ファイルと call を記述した.sci ファイルが必要となる．.c ファイルは Windows の場合とほぼ同じだが，dll には変換しないため関数のヘッダ部分は記述しない．MuPAT ビルド時に C 言語のプログラムをコンパイルし，動的リンクライブラリを生成してから ilib_for_link 関数を使用して Scilab にリンクする．

4.2　外部関数利用時の実行時間

本節では外部関数による高速化の効果を示すために，DD 演算，QD 演算の実行時間を，Scilab の機能のみを使用した実装と外部関数を利用した実装で比較する．ここで [7] のデータを再掲して用いる．以下の 4 つの演算に対し，(i)〜(iii) 10^4 回，(iv) 10^3 回反復したときの実行時間を計測した．

(i) スカラーの四則演算

(ii) ベクトル加算

(iii) 内積

(iv) 行列ベクトル積

使用するベクトル，行列の次元は 10^3 とし，実行時間は計測 3 回の平均値とした．計算環境には，Intel Core i5 2.5GHz, 4GB memory, Windows 7, Scilab version 5.3.3 を用いた．実行時間を表 5, 6 に示す．*MuPAT* は Scilab の機能のみを使用した実装を表し，*MuPAT_c* は外部関数を利用した実装を表す．

最も高速化されたのは QD 除算で，*MuPAT* では倍精度演算の 2317 倍もの実行時間がかかっていたが，*MuPAT_c* では 24 倍になった．

MuPAT_c においては，内積 $\boldsymbol{x}'\boldsymbol{y}$ は DD 演算で 9859 倍から 52 倍に，QD 演算では 270273 倍から 223 倍にまで改善された．外部関数を用いた高速化の効果は，QD 演算の場合 180〜1200 倍である．*MuPAT_c* では，1 回の呼び出しで外部関数に演算を実行させる量が大きいほど，効果的に演算時間を減らせるように実装している．QD 除算や内積は他の演算に比べて必要な倍精度演算回数が多いため，演算時間が大幅に削減されたといえる．Mac OS の場合も，Windows の場合と同様の結果が得られた．

ここで，呼び出しに対するオーバーヘッドの影響を調べるために，*MuPAT_c* を使って以下の 2 つの実行時間を比較した．これらは演算量は同じであるが，(a) では加算 1 回ごとに，合計 10^6 回外部関数を呼び出すのに対し (b) ではベクトルをそのまま外部関数に渡すため，1 回だけ外部関数を呼び出す．

表 5: スカラーの四則演算の実行時間 (括弧内は比率を表す，単位：秒)

		$x \pm y$	xy	x / y
D		0.022 (1)	0.021 (1)	0.017 (1)
DD	*MuPAT*	0.22 (10.4)	0.41 (19.7)	0.43 (25.6)
	MuPAT_c	0.27 (12.1)	0.29 (13.7)	0.31 (18.2)
QD	*MuPAT*	5.31 (241.5)	6.99 (333.0)	39.39 (2317.6)
	MuPAT_c	0.32 (14.7)	0.41 (19.5)	0.42 (24.5)

表 6: ベクトル演算の実行時間 (括弧内は比率を表す，単位：秒)

		$\boldsymbol{x}+\boldsymbol{y}$	$\boldsymbol{x}'\boldsymbol{y}$	$A\boldsymbol{x}$
D		0.01 (1)	0.02 (1)	4.10 (1)
DD	*MuPAT*	0.66 (66.0)	197.18 (9859.0)	116.84 (28.5)
	MuPAT_c	0.69 (69.0)	1.05 (52.5)	79.72 (19.4)
QD	*MuPAT*	7.39 (739.0)	5405.47 (270273.5)	1154.56 (281.6)
	MuPAT_c	2.36 (236.0)	4.46 (223.0)	226.92 (55.3)

(a) スカラー x, y に対し，$(x+y)$ を 10^6 回反復する

(b) 10^6 次元ベクトル $\boldsymbol{x}, \boldsymbol{y}$ に対し，$\boldsymbol{x}+\boldsymbol{y}$ を計算する

QD 演算の場合，(a) の実行時間は 32.5 秒だったのに対し，(b) では 0.31 秒だった．よって，呼び出しに対するオーバーヘッドは 3.2×10^{-5} 秒であり，演算 104 回に相当する．呼び出しに対するオーバーヘッドを避けるためには，スカラー演算よりもベクトルや行列演算を使うことが推奨される．

5　疎行列向けの実装

MuPAT では，3つの異なる精度のデータ型，`constant`，`dd`，`qd`型が扱える．`dd`型と`qd`型では`constant` 型の 2 倍と 4 倍のメモリを消費するため，メモリ 4GB の場合，最大でも `dd` 型で 4000 次，`qd` 型で 2500 次の行列しか定義できない．そのため，メモリ使用量や計算時間の観点から大規模な行列演算は難しい．

そこで，MuPAT の特徴を維持することに加えて，

- 密行列形式，疎行列形式を区別せずに利用できる

ことを目標に，疎行列向け高精度演算を実装した [6, 7].

5.1　Scilab における倍精度疎行列データ型

Scilab の倍精度疎行列データ型 `sparse` 型は行列の非零要素の値，行番号，列番号により行列を表現する．例えば，次の行列

$$\mathtt{A} = \begin{pmatrix} 0 & 4 & 0 & 7 \\ 1 & 0 & 0 & 8 \\ 2 & 0 & 6 & 0 \\ 3 & 5 & 0 & 0 \end{pmatrix}$$

は，`sparse` 型では以下のように表される．

```
 A  =
(    4,    4) sparse matrix
(    1,    2)        4.
(    1,    4)        7.
(    2,    1)        1.
(    2,    4)        8.
(    3,    1)        2.
(    3,    3)        6.
(    4,    1)        3.
(    4,    2)        5.
```

1 行目の `(4, 4)` は行列のサイズを表している．2 行目以降に行方向に行番号，列番号，値が保存されている．行番号を格納する行ベクトル，列番号を格納する列ベクトル，値を格納する値ベクトルは，spget 関数により取得できる．

sparse 型は constant 型と同様に演算子 (+, -, *) を使って計算でき，constant 型との混合演算もできる．また，abs 等の基本的な関数や，constant 型に変換するための full 等，sparse 型向けの関数も用意されている．非零要素率 5% の行列の場合，sparse 型は constant 型の約 7.5% のメモリ使用量で行列を格納できる．

5.2　疎行列形式の QD の数の定義

3.2 節と同様に tlist を利用して，8 倍精度疎行列データ型 'qdsp' を定義した．qdsp 型の最も単純な定義方法は，qd 型の定義に倣い，4 つの sparse 型 $A0, A1, A2, A3$ を組み合わせて，

```
A = tlist(['qdsp','d'], A0,A1,A2,A3);
```

とすることである．この場合，$A0, A1, A2, A3$ はそれぞれの項で行番号，列番号，値を保持することになる．しかし，行列の非零要素の配置は QD の数の各項で変わらないため，$A0$ と $A1, A2, A3$ のインデックスの情報が重複してしまい，メモリの無駄となる．

sparse 型からインデックスや値などの行列の情報を取り出すためには，spget 関数を利用する必要がある．QD の場合，1 つの値に対して 4 回 spget 関数を使用しなければならず，外部関数を使った高速化の際にはオーバーヘッドとなる．

この点を考慮し，qdsp 型の定義には sparse 型の組み合わせは用いず，CCS(Compressed Column Storage) 形式を採用して独自に定義することとした．演算アルゴリズムは，文献 [2] にある CCS 形式向けのアルゴリズムを参考にした．

CCS 形式では，列方向に非零要素を格納する値ベクトル，行番号を格納するインデックスベクトル，各列の先頭にある要素が値ベクトルの何番目に格納されているかを示すポインタベクトルにより行列を表現する．5.1 節の行列 A を例に CCS 形式を説明する．

いま，Ax, Ai, Ap をそれぞれ CCS 形式に基づく A の値ベクトル，インデックスベクトル，ポインタベクトルとする．行列 A の第 1 列の非零要素は第 2 行の 1，第 3 行の 2，第 4 行の 3 である．そのため，Ax には先頭から $1, 2, 3$ が格納され，Ai には先頭から $2, 3, 4$ が格納される．Ap の先頭は必ず 1 が格納される．第 1 列の非零要素数は 3 で，Ax の 4 番目から第 2 列目の要素が格納されるため，Ap の 2 番目には 4 が格納される．以降，同様にして第 2 列から第 4 列まで格納していく．Ap の最後の成分には非零要素数に 1 を足した数が格納される．図 3 に格納の様子を示す．最終的に Ax, Ai, Ap は以下のようになる．

$$
\begin{aligned}
Ax &= (1,2,3,4,5,6,7,8)^T \\
Ai &= (2,3,4,1,4,3,1,2)^T \\
Ap &= (1,4,6,7,9)^T
\end{aligned}
$$

CCS 形式の値ベクトルを QD の数に拡張し，qdsp 型を以下のように定義した．

```
A = tlist(['qdsp','p','i','s','d'],
    Ap, Ai, [m,n], Ax0, Ax1, Ax2, Ax3);
```

Ap には列ポインタ，Ai には行番号，[m,n] には A の行列サイズ，Ax0，Ax1，Ax2，Ax3 には QD の数の値を，それぞれ一次元配列で保存する．tlist 関数の第 1 引数では，変数の名前を定義している．'qdsp' は新しく定義したデータ型，'p' は Ap，'i' は Ai，'s' は [m,n]，'d' は Ax0 の名前を示している．これにより，A.p, A.i, A.s, A.d と入力するだけで，それぞれの名前が示す値を参照できる．

図 4 に，QD の数を成分に持つ 4×4 行列を qdsp 型で定義したときの例を示す．4×4 行列 A が成分に QD の数 a, b, c, d, e を持つとき，Ax0 にはそれぞれの QD の数の第 1 項が格納されるため，Ax0 $= (a_0, b_0, c_0, d_0, e_0)^T$ となる．以下同様に，Ax1 には第 2 項，Ax2 には第 3 項，Ax3 には第 4 項が格納される．

本実装では qdsp 型の他に，値ベクトルを 2 本にした 4 倍精度疎行列データ型 'ddsp' を同様に定義した．このように定義した ddsp 型と qdsp 型では，メモリ 4GB のとき，非零要素率 0.01% の場合，それぞれ最大で約 12 万次と約 7 万次の行列を定義できる．

5.3　疎行列形式の QD 演算の定義

本節では，ddsp，qdsp 型に対する演算の定義を行う [アルゴリズムの詳細は Appendix B に示す]．疎行列演算にはループを多用しなければならず，Scilab の機能のみで実装した場合は計算時間が多くかかってしまう．そのため，本実装では外部関数を利用し

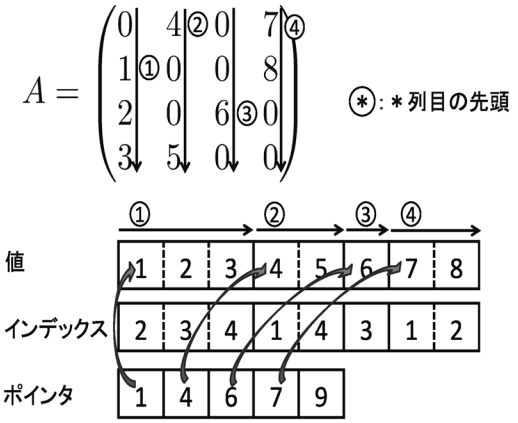

図 3: CCS 形式の例

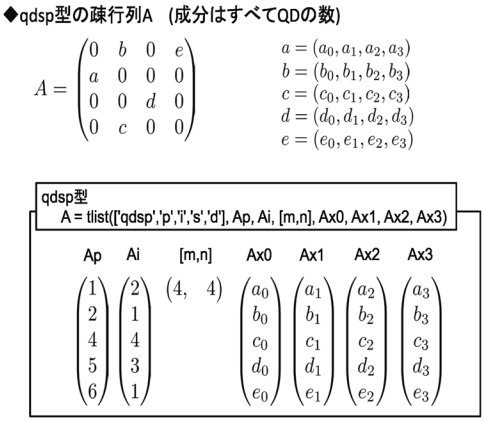

図 4: 4×4 行列を qdsp 型で定義した場合

て疎行列形式の QD 演算を定義し，高速化を図った [6].

まず，6 つのデータ型 constant，dd，qd，sparse，ddsp，qdsp に対して，共通の演算子 (+, -, *) を使って行列演算ができるようにするため，3.3 節と同様に演算子のオーバーロードをした．6 つのデータ型に対する二項演算は，同じデータ型同士の演算を含めて 36 通り存在する．そのうち，sparse 型と constant 型に対する 4 通りの二項演算は Scilab で，constant，dd，qd 型に対する 8 通りの二項演算は MuPAT で既に定義されている．よって，6 つのデータ型で相互に演算できるようにするためには，各行列演算 (+, -, *) に対し 24 種類の関数を新たに実装しなくてはならない．

表 7 に行列加算に関する関数を示す．4.1 節と同様，call と link を利用し，C 言語による外部関数を呼び出して演算を実行するように実装している．

これらの関数の実装により，二項演算に関する変数 a, b が 6 つデータ型のうちのどれであっても，以下のように演算子 (+, -, *) を使って相互に演算が実行できる．

$$a \bullet b$$

$$\begin{pmatrix} a, b \in \{\texttt{constant}, \texttt{dd}, \texttt{qd}, \texttt{sparse}, \texttt{ddsp}, \texttt{qdsp}\} \\ \bullet \in \{+, -, *\} \end{pmatrix}$$

また，Scilab の sparse 型向けの組み込み関数を ddsp，qdsp 型でも扱えるように拡張した．表 8 に MuPAT の疎行列向け関数を示す．

5.4　メモリ使用量と実行時間の比較

ddsp，qdsp 型を用いた疎行列演算の有効性を調べるため，メモリ使用量と実行時間を計測し，密行列データ型との比較を行う．計算環境は，Intel Core i5 2.5GHz，4GB，OS は Windows 7，Scilab version 5.3.3 を用いる．また密行列データ型の測定には，*MuPAT_c* を使用する．

5.4.1　メモリ使用量

非零要素率の異なる 5 つの 1000 次乱数行列を constant，dd，qd，sparse，ddsp，qdsp 型で定義したときのメモリ使用量を表 9 に示す．ddsp 型の場合では非零要素率が 66% 以下，qdsp 型の場合では非零要素率が 80% 以下のとき，密行列データ型を使用するよりもメモリ使用量が抑えられることがわかる．

5.4.2　行列演算

5.4.1 で用いた行列を使用し，次の演算を 10 回反復したときのメモリ使用量と実行時間を表 10 に示す．

(i) 行列ベクトル積：　$A\boldsymbol{x}, B\boldsymbol{x}, C\boldsymbol{x}, D\boldsymbol{x}$

(ii) 行列加算：　$A + B, B + C, C + A$

(iii) 行列乗算：　AB, BC, CA

行列ベクトル積では，非零要素率 5% のとき，ddsp 型を使用すると dd 型の 9.4 倍，qdsp 型を使

表 7: 疎行列加算に関する関数

関数名	オーバーロードされる演算		関数名	オーバーロードされる演算	
%sp_a_dd	sparse 型	＋dd 型	%ddsp_a_ddsp	ddsp 型	＋ddsp 型
%dd_a_sp	dd 型	＋sparse 型	%qdsp_a_s	qdsp 型	＋constant 型
%sp_a_qd	sparse 型	＋qd 型	%s_a_qdsp	constant 型	＋qdsp 型
%qd_a_sp	qd 型	＋sparse 型	%qdsp_a_dd	qdsp 型	＋dd 型
%ddsp_a_s	ddsp 型	＋constant 型	%dd_a_qdsp	dd 型	＋qdsp 型
%s_a_ddsp	constant 型	＋ddsp 型	%qdsp_a_qd	qdsp 型	＋qd 型
%ddsp_a_dd	ddsp 型	＋dd 型	%qd_a_qdsp	qd 型	＋qdsp 型
%dd_a_ddsp	dd 型	＋ddsp 型	%qdsp_a_sp	qdsp 型	＋sparse 型
%ddsp_a_qd	ddsp 型	＋qd 型	%sp_a_qdsp	sparse 型	＋qdsp 型
%qd_a_ddsp	qd 型	＋ddsp 型	%qdsp_a_ddsp	qdsp 型	＋ddsp 型
%ddsp_a_sp	ddsp 型	＋sp 型	%ddsp_a_qdsp	ddsp 型	＋qdsp 型
%sp_a_ddsp	sp 型	＋ddsp 型	%qdsp_a_qdsp	qdsp 型	＋qdsp 型

表 8: MuPAT の疎行列向け関数

	関数名	機能	引数	戻り値
定義と変換	ddsp(Ap,Ai,Ax,m,n)	ddsp 型の生成	dd, constant	ddsp
	qdsp(Ap,Ai,Ax,m,n)	qdsp 型の生成	qd, constant	qdsp
	ddsp2adj(A)	隣接形式への変換	ddsp	dd, constant
	qdsp2adj(A)	隣接形式への変換	qdsp	qd, constant
	adj2ddsp(xadj,adjncy,anz,mn)	隣接形式からの変換	dd, constant	ddsp
	adj2qdsp(xadj,adjncy,anz,mn)	隣接形式からの変換	qd, constant	qdsp
全精度共通の関数	full(A)	密行列型への変換	ddsp, qdsp	dd, qd
	nnz(A)	非零要素数	ddsp, qdsp	constant
	spget(A)	行列番号と値の抜出し	ddsp, qdsp	constant
	abs(A)	絶対値	ddsp, qdsp	ddsp, qdsp
	diag(A)	対角要素の抜出し	ddsp, qdsp	ddsp, qdsp
その他の関数	isddsparse(A)	型の判定	ddsp	constant
	isqdsparse(A)	型の判定	qdsp	constant
	ddspeye(n, n)	単位行列	constant	ddsp
	qdspeye(n, n)	単位行列	constant	qdsp
	ddspones(A)	成分がすべて 1 の疎行列	ddsp	ddsp
	qdspones(A)	成分がすべて 1 の疎行列	qdsp	qdsp
	ddsprand(m, n, r)	疎乱数行列	constant	ddsp
	qdsprand(m, n, r)	疎乱数行列	constant	qdsp
	ddspzeros(m, n)	疎零行列	constant	ddsp
	qdspzeros(m, n)	疎零行列	constant	qdsp

表 9: メモリ使用量 （メモリ：MB）

行列		メモリ (MB)					
	非零要素率	constant	sparse	dd	ddsp	qd	qdsp
A	1%	8.00	0.13	16.00	0.25	32.00	0.41
B	5%	8.00	0.60	16.00	1.20	32.00	2.01
C	10%	8.00	1.21	16.00	2.40	32.00	4.02
D	40%	8.00	4.86	16.00	9.71	32.00	16.18
E	66%	8.00	7.89	16.00	15.77	32.00	26.31
F	80%	8.00	9.56	16.00	19.11	32.00	31.88

表 10: 行列演算のメモリ使用量と実行時間 （メモリ：MB，実行時間：秒）

	非零要素率	メモリ使用量		実行時間					
		ddsp	qdsp	dd	ddsp	加速率	qd	qdsp	加速率
$A\boldsymbol{x}$	-	-	-	0.81	0.02	52.6	4.44	1.56	2.8
$B\boldsymbol{x}$	-	-	-	0.79	0.08	9.4	4.42	1.76	2.5
$C\boldsymbol{x}$	-	-	-	0.79	0.16	4.9	4.42	2.03	2.2
$D\boldsymbol{x}$	-	-	-	0.81	0.69	1.2	4.45	4.13	1.3
$A+B$	6%	1.4	2.4	0.83	0.07	11.3	2.73	0.12	21.9
$C+A$	11%	2.6	4.4	0.83	0.15	5.5	2.73	0.24	11.2
$B+C$	15%	3.5	5.8	0.83	0.19	4.4	2.73	0.34	7.9
AB	40%	9.4	15	567.19	0.65	879.4	4000.8	4.43	903.0
CA	64%	15	25	566.28	1.06	536.4	3964.5	8.91	444.7
BC	99%	23	39	566.41	2.82	201.0	3982.2	47.50	83.8

用すると qd 型の 2.5 倍高速に実行できた．非零要素率 10% のときは，ddsp 型を使用すると dd 型の 4.9 倍，qdsp 型を使用すると qd 型の 2.2 倍高速に実行できた．行列ベクトル積では，非零要素率 40% 以下の行列に対して疎行列データ型を用いた方が実行速度が速くなった．

行列加算では，計算結果の非零要素率が 6% のとき，ddsp 型を使用すると dd 型の 11.3 倍，qdsp 型を使用すると qd 型の 21.9 倍高速に実行できた．非零要素率 15% のときは，ddsp 型を使用すると dd 型の 4.4 倍，qdsp 型を使用すると qd 型の 7.9 倍高速に実行できた．計算結果のメモリ使用量は，密行列データ型の $\frac{1}{13}$〜$\frac{1}{5}$ に抑えられた．

行列乗算では，どの演算においても疎行列データ型を使うことで大幅な高速化が達成された．しかし，BC における計算結果のメモリ使用量は，qd 型を使用すると 32MB なのに対し，qdsp 型では 39MB となり，疎行列データ型を使用した方が上回った．これは，BC の非零要素率が 99% となり，ポインタや行番号を別に保持しなければならない疎行列データ型よりも密行列データ型の方がメモリの利用効率が良かったためである．しかし，途中の無駄な演算が省かれているため，実行速度は qdsp 型の方が 83 倍高速である．

行列乗算の加速率が大きいのは，1 演算に対して必要な倍精度演算回数が多く，オーバーヘッドが相対的に小さくなるためである．

6　まとめ

本稿では，Scilab における高精度演算環境 MuPAT の具体的な実装について述べた．MuPAT では，密行列と疎行列形式の倍精度，4 倍精度，8 倍精度演算を同時に扱え，すべての精度や格納形式の演算で，共通の演算子や関数を使用できることを目指した．

MuPAT では，Scilab の tlist 関数を利用し Scilab の倍精度密行列データ型 `constant` を組み合わせて，DD，QD の数に相当する密行列データ型‘`dd`’，‘`qd`’を定義している．それぞれのデータ型に関する演算子 (+, -, *, /) や関数をオーバーロードすることにより，すべての精度で共通の演算子や関数が使える．

また，DD 演算や QD 演算のアルゴリズムを実行する C 言語の外部関数を用意し，Scilab から外部関数を呼び出すことで，高精度演算の高速化も行っている．外部関数を利用した MuPAT の高精度演算は，Scilab のみで実装した場合よりも 180〜1200 倍高速である．

さらに，疎行列向け高精度演算のために，DD，QD の数を成分に持つ疎行列データ型‘`ddsp`’，‘`qdsp`’を定義した．`ddsp`，`qdsp` 型では，メモリを節約するため，Scilab の倍精度疎行列データ型 `sparse` を組み合わせて定義せず，CCS 形式に従い独自にデータ型を定義している．MuPAT の `constant`，`dd`，`qd` 型と同様に演算子 (+, -, *) をオーバーロードしたため，`sparse`，`ddsp`，`qdsp` 型でも共通の演算子を使って演算ができる．高速化のため，疎行列演算の実装には C 言語による外部関数を利用した．

メモリ 4GB の場合，密行列データ型の `dd` 型では最大 4000 次，`qd` 型では最大 2500 次の行列しか扱えないが，行列の非零要素率が 0.1% のとき，`ddsp` 型で最大 120000 次，`qdsp` 型で最大 70000 次の行列を扱える．`qdsp` 型を使用した場合，行列ベクトル積では `qd` 型の 1〜2 倍，行列加算では 7〜21 倍，行列積では 83〜900 倍高速に実行できる．

本実装により，MuPAT では `constant`，`dd`，`qd`，`sparse`，`ddsp`，`qdsp` の 6 つのデータ型を同時に扱え，それぞれの精度の疎行列と密行列データ型に対する混合演算が可能となった．また，MuPAT の密行列向け高精度演算と同じ演算子 (+, -, *) を使って疎行列向け高精度演算が利用できる．そのため，ユーザーが密行列向け演算を疎行列向け演算に切り替えるには，プログラムの変数の定義部分を書き換えるだけでよく，より少ないメモリ量と短い実行時間で，さらに手軽に高精度演算を扱えるようになった．

Reference

[1] D. H. Bailey, A fortran-90 double-double library, Available at http://www.nersc.gov/dhbailey/mpdist/mpdist.html

[2] T. A. Davis, Direct Methods for Sparse Linear Systems, SIAM, Philadelphia (2006).

[3] T. J. Dekker, A floating-point technique for extending the available precision, Numerische Mathematik, 18:224-242 (1971).

[4] 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 (2000).

[5] D. E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, volume 2, Addison Wesley, Reading, Massachusetts (1981).

[6] 吉川慧子, 疎行列向け高精度演算の Scilab への実装, 東京理科大学大学院修士論文 (2013).

[7] S. Kikkawa, T. Saito, E. Ishiwata and H. Hasegawa, Development and acceleration of multiple precision arithmetic toolbox MuPAT for Scilab, JSIAM Letters, 5:9-12 (2013).

[8] MuPAT, http://www.mi.kagu.tus.ac.jp/qupat.html

[9] D. M. Priest, On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations, PhD thesis, University of California, Berkeley, November 1992. Available by anonymous FTP at ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z.

[10] 齊藤翼，Scilab における 4 倍精度演算環境「QuPAT」の実装, 東京理科大学大学院修士論文 (2011).

[11] 齊藤翼，Development of a high-precision arithmetic environment on Scilab and its

applications，東京理科大学大学院学位論文 (2013).

[12] T. Saito, E. Ishiwata, H. Hasegawa, Development of quadruple precision arithmetic toolbox QuPAT on Scilab, ICCSA2010, Proceedings Part II, LNCS 6017, 60–70 (2010).

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

[14] J. R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete & Computational Geometry, 18(3):305–363 (1997).

Appendix A : Algorithms for DD and QD Arithmetics

We describe the details of the algorithms for DD and QD arithmetics. The procedures of algorithms are based on Knuth [5], Dekker [3], Priest [9], Shewchuk [14], Bailey [1] and Hida et al. [4].

A.1 Preliminaries for DD and QD arithmetics

In this section, we introduce some algorithms of floating-point arithmetic.

Assuming that $|a| \geq |b|$, Algorithm A.1, Fast-Two-Sum, produces a nonoverlapping expansion $s+e$ such that $a+b = s+e$, where s is an approximation to $a+b$ and e represents the round-off error in the calculation of s, in [14, p. 312].

Algorithm A.2, Two-Sum, is similar to Algorithm A.1, but Algorithm A.2 does not require the condition of $|a| \geq |b|$.

Algorithm A.1 Fast-Two-Sum(a, b) : Assume that $|a| \geq |b|$

$s \leftarrow a \oplus b$
$v \leftarrow s \ominus a$
$e \leftarrow b \ominus v$
return (s, e)

Algorithm A.2 Two-Sum(a, b)

$s \leftarrow a \oplus b$
$v \leftarrow s \ominus a$
$e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
return (s, e)

Algorithm A.3, Split, produces a 26 bit value a_h and a nonoverlapping 26 bit value a_l such that $|a_h| > |a_l|$ and $a = a_h + a_l$, in [14, p. 325].

Algorithm A.3 Split(a)

$t \leftarrow 134217729 \otimes a$
$v \leftarrow t \ominus a$
$a_h \leftarrow t \ominus v$
$a_l \leftarrow a \ominus a_h$
return (a_h, a_l)

Algorithm A.4, Two-Prod, produces a nonoverlapping expansion $p+e$ such that $a \times b = p+e$, where p is an approximation to $a \times b$ and e represents the round-off error in the calculation of p, in [14, p. 326].

Algorithm A.4 Two-Prod(a, b)

$p \leftarrow a \otimes b$
$[a_h, a_l] \leftarrow$Split(a)
$[b_h, b_l] \leftarrow$Split(b)
$e \leftarrow ((a_h \otimes b_h \ominus p) \oplus a_h \otimes b_l \oplus a_l \otimes b_h) \oplus a_l \otimes b_l$
return (p, e)

Algorithm A.5, Two-Sqr, produces a nonoverlapping expansion $p+e$ such that $a^2 = p+e$, where p is an approximation to a^2 and e represents the round-off error in the calculation of p.

Algorithm A.5 Two-Sqr(a)

$p \leftarrow a \otimes a$
$[a_h, a_l] \leftarrow$Split(a)
$e \leftarrow ((a_h \otimes a_h \ominus p) \oplus (a_h \otimes a_l) \otimes 2.0) \oplus a_l \otimes a_l$
return (p, e)

Algorithm A.6, Three-Sum, produces a nonoverlapping expansion $d+e+f$ such that $a+b+c = d+e+f$, in [4].

Algorithm A.6 Three-Sum(a, b, c)

$[t_0, t_1] \leftarrow$ Two-Sum(a, b)
$[d, t_2] \leftarrow$ Two-Sum(t_0, c)
$[e, f] \leftarrow$ Two-Sum(t_1, t_2)
return (d, e, f)

Algorithm A.7, Three-Sum2, produces two double-precision numbers $d = (a \oplus b) \oplus c$ and $e = (a+b+c) - s$, in [4].

Algorithm A.7 Three-Sum2(a, b, c)

$[t_0, t_1] \leftarrow$ Two-Sum(a, b)
$[d, t_2] \leftarrow$ Two-Sum(t_0, c)
$e = t_1 \oplus t_2$
return (d, e)

Supposing that a_0, a_1, a_2, a_3 and a_4 construct a five-term expansion with limited overlapping bits, with a_0 being the most significant component. Then Algorithm A.8, Renormalize, produces a four-term nonoverlapping expansion $b_{(qd)} = b_0 + b_1 + b_2 + b_3$.

Algorithm A.8 Renormalize$(a_0, a_1, a_2, a_3, a_4)$

```
[s, t3] ← Fast-Two-Sum(a3, a4)
[s, t2] ← Fast-Two-Sum(a2, s)
[s, t1] ← Fast-Two-Sum(a1, s)
[b0, t0] ← Fast-Two-Sum(a0, s)
[s, t2] ← Fast-Two-Sum(t2, t3)
[s, t1] ← Fast-Two-Sum(t1, s)
[b1, t0] ← Fast-Two-Sum(t0, s)
[s, t1] ← Fast-Two-Sum(t1, t2)
[b2, t0] ← Fast-Two-Sum(t0, s)
b3 = t0 ⊕ t1
return (b0, b1, b2, b3)
```

Algorithm A.9, Renormalize2, produces a four-term nonoverlapping expansion $b_{(qd)} = b_0 + b_1 + b_2 + b_3$. This algorithm is similar to Algorithm A.8 except for the number of arguments.

Algorithm A.9 Renormalize2(a_0, a_1, a_2, a_3)

```
[s, t2] ← Fast-Two-Sum(a2, a3)
[s, t1] ← Fast-Two-Sum(a1, s)
[b0, t0] ← Fast-Two-Sum(a0, s)
[s, t1] ← Fast-Two-Sum(t1, t2)
[b1, t0] ← Fast-Two-Sum(t0, s)
[b2, b3] ← Fast-Two-Sum(t0, t1)
return (b0, b1, b2, b3)
```

Table 11 shows the number of double-precision arithmetic operations for Algorithm A.1 $\sim$ A.9.

Table 11: Number of double-precision arithmetic operations for Algorithm A.1 $\sim$ A.9

Algorithm	$\oplus$, $\ominus$	$\otimes$	Total
Fast-Two-Sum (A.1)	3	0	3
Two-Sum (A.2)	6	0	6
Split (A.3)	3	1	4
Two-Prod (A.4)	10	7	17
Two-Sqr (A.5)	7	5	12
Three-Sum (A.6)	18	0	18
Three-Sum2 (A.7)	13	0	13
Renormalize (A.8)	28	0	28
Renormalize2 (A.9)	18	0	18

A.2 Algorithms for DD arithmetic

In this section, we show the algorithms of four arithmetic operations for double-double numbers.

A.2.1 addition

Algorithm A.10, dd_d_add, shows the procedure for adding a double-precision number b to a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$. If you want to add a double-double number $b_{(dd)}$ to a double-precision number a, dd_d_add (b_0, b_1, a) returns the result. d_dd_add is same as dd_d_add.

Algorithm A.10 dd_d_add (a_0, a_1, b)

```
[s, e] ← Two-Sum(a0, b)
e ← e ⊕ a1
[c0, c1] ← Fast-Two-Sum(s, e)
return (c0, c1)
```

Algorithm A.11, dd_dd_add, shows the procedure for adding a double-double number $b_{(dd)}$ to a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.11 dd_dd_add (a_0, a_1, b_0, b_1)

```
[s, e] ← Two-Sum(a0, b0)
e ← e ⊕ (a1 ⊕ b1)
[c0, c1] ← Fast-Two-Sum(s, e)
return (c0, c1)
```

A.2.2 subtraction

Algorithm A.12 (A.13), dd_d_sub (d_dd_sub), shows the procedure for subtracting a double- precision number b (a double-double number $b_{(dd)}$) from a double-double number $a_{(dd)}$ (a double-precision number a) and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.12 dd_d_sub (a_0, a_1, b)

```
s ← −b
[c0, c1] ← dd_d_add (a0, a1, s)
return (c0, c1)
```

Algorithm A.13 d_dd_sub (a, b_0, b_1)

```
s0 ← −b0
s1 ← −b1
[c0, c1] ← d_dd_add (a, s0, s1)
return (c0, c1)
```

Algorithm A.14, dd_dd_sub, shows the procedure for subtracting a double-double number $b_{(dd)}$ from a double-double number $a_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.14 dd_dd_sub (a_0, a_1, b_0, b_1)

1: $s_0 \leftarrow -b_0$
2: $s_1 \leftarrow -b_1$
3: $[c_0, c_1] \leftarrow$ dd_dd_add (a_0, a_1, s_0, s_1)
4: **return** (c_0, c_1)

A.2.3 multiplication

Algorithm A.15, dd_d_mul, shows the procedure for multiplying a double-double number $a_{(dd)}$ by a double-precision number b and returns the double-double number $c_{(dd)} = c_0 + c_1$. d_dd_mul is same as dd_d_mul.

Algorithm A.15 dd_d_mul (a_0, a_1, b)

1: $[p, e] \leftarrow$ Two-Prod(a_0, b)
2: $e \leftarrow e \oplus (a_1 \otimes b)$
3: $[c_0, c_1] \leftarrow$ Fast-Two-Sum(p, e)
4: **return** (c_0, c_1)

Algorithm A.16, dd_dd_mul, shows the procedure for multiplying a double-double number $a_{(dd)}$ by a double-double number $b_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.16 dd_dd_mul (a_0, a_1, b_0, b_1)

1: $[p, e] \leftarrow$ Two-Prod(a_0, b_0)
2: $e \leftarrow e \oplus (a_0 \otimes b_1)$
3: $e \leftarrow e \oplus (a_1 \otimes b_0)$
4: $[c_0, c_1] \leftarrow$ Fast-Two-Sum(p, e)
5: **return** (c_0, c_1)

A.2.4 division

Supposing that $b \neq 0$ and $b_0 \neq 0$. Algorithm A.17 (A.18), dd_d_div (d_dd_div), shows the procedure for dividing a double-double number $a_{(dd)}$ (a double-precision number a) by a double-precision number b (a double-double number $b_{(dd)}$) and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.17 dd_d_div (a_0, a_1, b)

1: $c_0 \leftarrow a_0 \oslash b$
2: $[p, e] \leftarrow$ Two-Prod(c_0, b)
3: $c_1 \leftarrow ((a_0 \ominus p) \ominus e \oplus a_1) \oslash b$
4: $[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
5: **return** (c_0, c_1)

Algorithm A.18 d_dd_div (a, b_0, b_1)

1: $c_0 \leftarrow a \oslash b_0$
2: $[p, e] \leftarrow$ Two-Prod(c_0, b_0)
3: $c_1 \leftarrow ((a \ominus p) \ominus e \ominus c \otimes b_1) \oslash b_0$
4: $[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
5: **return** (c_0, c_1)

Supposing that $b_0 \neq 0$. Algorithm A.19, dd_dd_div, shows the procedure for dividing a double-double number $a_{(dd)}$ by a double-double number $b_{(dd)}$ and returns the double-double number $c_{(dd)} = c_0 + c_1$.

Algorithm A.19 dd_dd_div (a_0, a_1, b_0, b_1)

1: $c_0 \leftarrow a_0 \oslash b_0$
2: $[p, e] \leftarrow$ Two-Prod(c_0, b_0)
3: $c_1 \leftarrow ((a_0 \ominus p) \ominus e \oplus a_1 \ominus c \otimes b_1) \oslash b_0$
4: $[c_0, c_1] \leftarrow$ Fast-Two-Sum(c_0, c_1)
5: **return** (c_0, c_1)

Table 12 shows the number of double precision arithmetic operations for double-double arithmetic.

A.3 Algorithms for QD arithmetic

A.3.1 addition

Algorithm A.20, qd_d_add, shows the procedure for adding a double-precision number b to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.20 qd_d_add (a_0, a_1, a_2, a_3, b)

1: $[c_0, e] \leftarrow$ Two-Sum(a_0, b)
2: $[c_1, e] \leftarrow$ Two-Sum(a_1, e)
3: $[c_2, e] \leftarrow$ Two-Sum(a_2, e)
4: $[c_3, e] \leftarrow$ Two-Sum(a_3, e)
5: $[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize(c_0, c_1, c_2, c_3, e)
6: **return** (c_0, c_1, c_2, c_3)

Algorithm A.21, qd_dd_add, shows the procedure for adding a double-double number $b_{(dd)}$ to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Table 12: Number of double precision arithmetic operations for double-double arithmetic

	Algorithm	$\oplus$ & $\ominus$	$\otimes$	$\oslash$	Total
Addition	dd_d_add	10	0	0	10
	dd_dd_add	11	0	0	11
Subtraction	dd_d_sub, d_dd_sub	10	0	0	10
	dd_dd_sub	11	0	0	11
Multiplication	dd_d_mul	14	8	0	22
	dd_dd_mul	15	9	0	24
Division	dd_d_div	16	7	2	25
	d_dd_div	16	8	2	26
	dd_dd_div	17	8	2	27

Algorithm A.21 qd_dd_add $(a_0, a_1, a_2, a_3, b_0, b_1)$

$[c_0, e_0] \leftarrow$ Two-Sum(a_0, b_0)
$[c_1, e_1] \leftarrow$ Two-Sum(a_1, b_1)
$[c_1, e_0] \leftarrow$ Two-Sum(c_1, e_0)
$[c_2, e_1] \leftarrow$ Two-Sum(a_2, e_1)
$[c_2, e_0] \leftarrow$ Two-Sum(c_2, e_0)
$[e_0, e_1] \leftarrow$ Two-Sum(e_0, e_1)
$[c_3, e_0] \leftarrow$ Two-Sum(a_3, e_0)
$e_0 = e_0 \oplus e_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(c_0, c_1, c_2, c_3, e_0)$
return (c_0, c_1, c_2, c_3)

Algorithm A.22, qd_qd_add, shows the procedure for adding a quad-double number $b_{(qd)}$ to a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.22 qd_qd_add $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$[c_0, e_0] \leftarrow$ Two-Sum(a_0, b_0)
$[c_1, e_1] \leftarrow$ Two-Sum(a_1, b_1)
$[c_2, e_2] \leftarrow$ Two-Sum(a_2, b_2)
$[c_3, e_3] \leftarrow$ Two-Sum(a_3, b_3)
$[c_1, e_0] \leftarrow$ Two-Sum(c_1, e_0)
$[c_2, e_0, e_1] \leftarrow$ Three-Sum(c_2, e_1, e_0)
$[c_3, e_0] \leftarrow$ Three-Sum2(c_3, e_2, e_0)
$e_0 = e_0 \oplus e_1 \oplus e_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(c_0, c_1, c_2, c_3, e_0)$
return (c_0, c_1, c_2, c_3)

A.3.2 subtraction

Algorithm A.23 (A.24), qd_d_sub (d_qd_sub), shows the procedure for subtracting a double-precision number b (a quad-double number $b_{(qd)}$) from a quad-double number $a_{(qd)}$ (a double-precision number a) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.23 qd_d_sub (a_0, a_1, a_2, a_3, b)

$s \leftarrow -b$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_d_add (a_0, a_1, a_2, a_3, s)
return (c_0, c_1, c_2, c_3)

Algorithm A.24 d_qd_sub (a, b_0, b_1, b_2, b_3)

$s_0 \leftarrow -b_0$
$s_1 \leftarrow -b_1$
$s_2 \leftarrow -b_2$
$s_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ d_qd_add (a, s_0, s_1, s_2, s_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.25 (A.26), qd_dd_sub (dd_qd_sub), shows the procedure for subtracting a double-double number $b_{(dd)}$ (a quad-double number $b_{(qd)}$) from a quad-double number $a_{(qd)}$ (a double-double number $a_{(dd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.25 qd_dd_sub $(a_0, a_1, a_2, a_3, b_0, b_1)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_dd_add $(a_0, a_1, a_2, a_3, b_0, b_1)$
return (c_0, c_1, c_2, c_3)

Algorithm A.26 dd_qd_sub $(a_0, a_1, b_0, b_1, b_2, b_3)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$b_2 \leftarrow -b_2$
$b_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ dd_qd_add $(a_0, a_1, b_0, b_1, b_2, b_3)$
return (c_0, c_1, c_2, c_3)

Algorithm A.27 shows the procedure for subtracting a quad-double number $b_{(qd)}$ from a quad-

double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.27 qd_qd_sub $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$b_0 \leftarrow -b_0$
$b_1 \leftarrow -b_1$
$b_2 \leftarrow -b_2$
$b_3 \leftarrow -b_3$
$[c_0, c_1, c_2, c_3] \leftarrow$ qd_qd_add $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$
return (c_0, c_1, c_2, c_3)

A.3.3 multiplication

Algorithm A.28, qd_d_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a double-precision number b and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.28 qd_d_mul (a_0, a_1, a_2, a_3, b)

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b)
$[p_1, q_1] \leftarrow$ Two-Prod(a_1, b)
$[p_2, q_2] \leftarrow$ Two-Prod(a_2, b)
$[p_1, q_0] \leftarrow$ Two-Sum(p_1, q_0)
$[p_2, q_0, q_1] \leftarrow$ Three-Sum(p_2, q_0, q_1)
$p_3 \leftarrow a_3 \otimes b$
$[p_3, q_0] \leftarrow$ Three-Sum2(p_3, q_0, q_2)
$q_0 \leftarrow q_0 \oplus q_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, q_0)$
return (c_0, c_1, c_2, c_3)

Algorithm A.29, qd_dd_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a double-double number $b_{(dd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.29 qd_dd_mul $(a_0, a_1, a_2, a_3, b_0, b_1)$

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, b_1)
$[p_2, q_2] \leftarrow$ Two-Prod(a_1, b_0)
$[p_3, q_3] \leftarrow$ Two-Prod(a_2, b_0)
$[p_4, q_4] \leftarrow$ Two-Prod(a_1, b_1)
$[p_1, p_2, q_0] \leftarrow$ Three-Sum(p_1, p_2, q_0)
$[p_2, p_3, p_4] \leftarrow$ Three-Sum(p_2, p_3, p_4)
$[q_1, q_2] \leftarrow$ Two-Sum(q_1, q_2)
$[p_2, q_1] \leftarrow$ Two-Sum(p_2, q_1)
$[p_3, q_2] \leftarrow$ Two-Sum(p_3, q_2)
$[p_3, q_1] \leftarrow$ Two-Sum(p_3, q_1)
$p_4 \leftarrow p_4 \otimes q_2 \otimes q_1$
$q_3 \leftarrow q_3 \oplus q_4 \oplus (a_3 \otimes b_0) \oplus (a_2 \otimes b_1)$
$[p_3, q_0] \leftarrow$ Three-Sum2(p_3, q_0, q_3)
$p_4 \leftarrow p_4 \oplus q_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

Algorithm A.30, qd_qd_mul, shows the procedure for multiplying a quad-double number $a_{(qd)}$ by a quad-double number $b_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.30 qd_qd_mul$(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$[p_0, q_0] \leftarrow$ Two-Prod(a_0, b_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, b_1)
$[p_2, q_2] \leftarrow$ Two-Prod(a_1, b_0)
$[p_1, p_2, q_0] \leftarrow$ Three-Sum(p_1, p_2, q_0)
$[p_3, q_3] \leftarrow$ Two-Prod(a_0, b_2)
$[p_4, q_4] \leftarrow$ Two-Prod(a_1, b_1)
$[p_5, q_5] \leftarrow$ Two-Prod(a_2, b_0)
$[p_2, q_1, q_2] \leftarrow$ Three-Sum(p_2, q_1, q_2)
$[p_3, p_4, p_5] \leftarrow$ Three-Sum(p_3, p_4, p_5)
$[p_2, p_3] \leftarrow$ Two-Sum(p_2, p_3)
$[p_4, q_1] \leftarrow$ Two-Sum(p_4, q_1)
$[p_3, p_4] \leftarrow$ Two-Sum(p_3, p_4)
$p_4 \leftarrow q_2 \oplus p_5 \oplus q_1 \oplus p_4$
$p_3 \leftarrow p_3 \oplus (a_0 \otimes b_3) \oplus (a_1 \otimes b_2) \oplus (a_2 \otimes b_1) \oplus (a_3 \otimes b_0) \oplus q_0 \oplus q_3 \oplus q_4 \oplus q_5$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

A.3.4 division

Supposing that $b \neq 0$ and $b_0 \neq 0$. Algorithm A.31 (A.32), qd_d_div (d_qd_div), shows the procedure for dividing a quad-double number $a_{(qd)}$ (a double-precision number a) by a double-precision number b (a quad-double number $b_{(qd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.31 qd_d_div (a_0, a_1, a_2, a_3, b)

$c_0 \leftarrow a_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_0, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(a_0, a_1, a_2, a_3, t_0, t_1)$
$c_1 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_1, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_2 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ Two-Prod(c_2, b)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_3 \leftarrow r_0 \oslash b$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.32 d_qd_div (a, b_0, b_1, b_2, b_3)

$c_0 \leftarrow a \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ d_qd_sub (a, t_0, t_1, t_2, t_3)
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Supposing that $b_0 \neq 0$. Algorithm A.33 (A.34), qd_dd_div (dd_qd_div), shows the procedure for dividing a quad-double number $a_{(qd)}$ (a double-double number $a_{(dd)}$) by a double-double number $b_{(dd)}$ (a quad-double number $b_{(qd)}$) and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.33 qd_dd_div $(a_0, a_1, a_2, a_3, b_0, b_1)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_0, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(a_0, a_1, a_2, a_3, t_0, t_1)$
$c_1 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_1, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_2 \leftarrow r_0 \oslash b$
$[t_0, t_1] \leftarrow$ d_dd_mul (c_2, b_0, b_1)
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_dd_sub $(r_0, r_1, r_2, r_3, t_0, t_1)$
$c_3 \leftarrow r_0 \oslash b$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Algorithm A.34 dd_qd_div $(a_0, a_1, b_0, b_1, b_2, b_3)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ dd_qd_sub $(a_0, a_1, t_0, t_1, t_2, t_3)$
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

Supposing that $b_0 \neq 0$. Algorithm A.35, qd_qd_div, shows the procedure for dividing a quad-double number $a_{(qd)}$ by a quad-double number $b_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Table 13 shows the number of double precision arithmetic operations for quad-double arithmetic. One quad-double arithmetic operation needs tens or hundreds of double-precision operations, then the computation time may require hundreds times greater than double-precision arithmetic.

Algorithm A.35 qd_qd_div $(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$

$c_0 \leftarrow a_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_0, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(a_0, a_1, a_2, a_3, t_0, t_1, t_2, t_3)$
$c_1 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_1, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_2 \leftarrow r_0 \oslash b_0$
$[t_0, t_1, t_2, t_3] \leftarrow$ d_qd_mul $(c_2, b_0, b_1, b_2, b_3)$
$[r_0, r_1, r_2, r_3] \leftarrow$ qd_qd_sub $(r_0, r_1, r_2, r_3, t_0, t_1, t_2, t_3)$
$c_3 \leftarrow r_0 \oslash b_0$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize2(c_0, c_1, c_2, c_3)
return (c_0, c_1, c_2, c_3)

A.4 The other algorithms for QD

Algorithm A.36, qd_sqr, shows the procedure for squaring a quad-double number $a_{(qd)}$ and returns the quad-double number $c_{(qd)} = c_0 + c_1 + c_2 + c_3$.

Algorithm A.36 qd_sqr (a_0, a_1, a_2, a_3)

$[p_0, q_0] \leftarrow$ Two-Sqr(a_0)
$[p_1, q_1] \leftarrow$ Two-Prod(a_0, a_1)
$p_1 \leftarrow p_1 \otimes 2.0$
$q_1 \leftarrow q_1 \otimes 2.0$
$[p_2, q_2] \leftarrow$ Two-Prod(a_0, a_2)
$p_2 \leftarrow p_2 \otimes 2.0$
$q_2 \leftarrow q_2 \otimes 2.0$
$[p_3, q_3] \leftarrow$ Two-Sqr(a_1)
$[p_1, q_0] \leftarrow$ Two-sum(p_1, q_0)
$[q_0, q_1] \leftarrow$ Two-sum(q_0, q_1)
$[p_2, p_3] \leftarrow$ Two-sum(p_2, p_3)
$[s_0, t_0] \leftarrow$ Two-Sum(q_0, p_2)
$[s_1, t_1] \leftarrow$ Two-Sum(q_1, p_3)
$[s_1, t_0] \leftarrow$ Two-Sum(s_1, t_0)
$t_0 \leftarrow t_0 \oplus t_1$
$[s_1, t_0] \leftarrow$ Fast-Two-Sum(s_1, t_0)
$[p_2, t_0] \leftarrow$ Fast-Two-Sum(s_0, s_1)
$[p_3, q_0] \leftarrow$ Fast-Two-Sum(t_1, t_0)
$p_4 \leftarrow 2.0 \otimes a_0 \otimes a_3$
$p_5 \leftarrow 2.0 \otimes a_1 \otimes a_2$
$[p_4, p_5] \leftarrow$ Two-Sum(p_4, p_5)
$[q_2, q_3] \leftarrow$ Two-Sum(q_2, q_3)
$[t_0, t_1] \leftarrow$ Two-Sum(p_4, q_2)
$t_1 \leftarrow t_1 \oplus p_5 \oplus q_3$
$[p_3, p_4] \leftarrow$ Two-Sum(p_3, t_0)
$p_4 \leftarrow p_4 \oplus q_0 \oplus t_1$
$[c_0, c_1, c_2, c_3] \leftarrow$ Renormalize$(p_0, p_1, p_2, p_3, p_4)$
return (c_0, c_1, c_2, c_3)

Table 13: Number of double precision arithmetic operations for quad-double arithmetic

	Algorithm	$\oplus, \ominus$	$\otimes$	$\oslash$	Total
Addition	qd_d_add	52	0	0	52
	qd_dd_add	71	0	0	71
	qd_qd_add	91	0	0	91
Subtraction	qd_d_sub, d_qd_sub	52	0	0	52
	qd_dd_sub, dd_qd_sub	71	0	0	71
	qd_qd_sub	91	0	0	91
Multiplication	qd_d_mul	96	22	0	118
	qd_dd_mul	154	35	0	189
	qd_qd_mul	171	46	0	217
Division	qd_d_div	261	21	4	286
	d_qd_div	540	66	4	610
	qd_dd_div	273	24	4	301
	dd_qd_div	569	66	4	639
	qd_qd_div	579	66	4	649

Supposing that n is a power of 2. Algorithm A.37, mul_pwr_dd, shows the procedure for multiplying each component of double-double number by n.

Algorithm A.37 mul_pwr_dd (a_0, a_1, n)

$b_0 \leftarrow a_0 \otimes n$
$b_1 \leftarrow a_1 \otimes n$
return (b_0, b_1)

Supposing that n is a power of 2. Algorithm A.38, mul_pwr_qd, shows the procedure for multiplying each component of quad-double number by n.

Algorithm A.38 mul_pwr_qd (a_0, a_1, a_2, a_3, n)

$b_0 \leftarrow a_0 \otimes n$
$b_1 \leftarrow a_1 \otimes n$
$b_2 \leftarrow a_2 \otimes n$
$b_3 \leftarrow a_3 \otimes n$
return (b_0, b_1, b_2, b_3)

Supposing that n is an integer. Algorithm A.39 shows the procedure for computing an n-th power of a quad-double number $a_{(qd)}$ and returns the quad-double number.

Algorithm A.39 qd_pow (a_0, a_1, a_2, a_3, n) : Assume that $n \geq 0$

if $n = 0$ **then**
  return 1.0
end if
if $n = 1$ **then**
  return (a_0, a_1, a_2, a_3)
end if
$r_0 \leftarrow a_0$
$r_1 \leftarrow a_1$
$r_2 \leftarrow a_2$
$r_3 \leftarrow a_3$
$s_0 \leftarrow 1.0$
$N \leftarrow n$
while $N > 0$ **do**
  if N is odd **then**
    $[s_0, s_1, s_2, s_3]$
    $\leftarrow$ qd_qd_mul$(s_0, s_1, s_2, s_3, r_0, r_1, r_2, r_3)$
  end if
  $N \leftarrow N \oslash 2$
  if $N > 0$ **then**
    $[r_0, r_1, r_2, r_3] \leftarrow$ qd_sqr (r_0, r_1, r_2, r_3)
  end if
end while
return (r_0, r_1, r_2, r_3)

Appendix B : Algorithms for Sparse Matrix Arithmetics

In this section, we describe the algorithms for sparse matrix arithmetics with CCS format based on Davis [2].

B.1 Addition

Let A be a $m \times n$ sparse matrix and B and C be $m \times n$ dense matrices, and C be the result of the addition of A and B. Let Ap, Ai and Ax denote column pointer, row index and value vectors with CCS format, respectively. Algorithm B.1, sparse_dense_add, shows the procedure for adding A to B.

Algorithm B.1 sparse_dense_add (A, B)

```
1: C ← B
2: for j = 1 to n do
3:     for p = Ap(j) to Ap(j + 1) − 1 do
4:         C(Ai(p), j) ← Ax(p) + B(Ai(p), j)
5:     end for
6: end for
```

Let A, B and C be $m \times n$ sparse matrices and C store the result of the addition of A and B. Let Ap, Bp and Cp denote column pointer vectors, Ai, Bi and Ci be row index vectors, and Ax, Bx and Cx denote value vectors of A, B and C with CCS format, respectively. Algorithm B.2, sparse_sparse_add, shows the procedure for adding A to B.

Algorithm B.2 sparse_sparse_add (A, B)

```
1: nz ← 0;   w ← (0, 0, . . . , 0);   y ← (0, 0, . . . , 0)
2: for j = 1 to n do
3:     Cp(j) ← nz + 1
4:     for p = Ap(j) to Ap(j + 1) − 1 do
5:         i ← Ai(p)
6:         if w(i) < j + 1 then
7:             w(i) ← j + 1
8:             Ci(nz + +) ← i + 1
9:             y(i) ← Ax(p)
10:        end if
11:    end for
12:    for p = Bp(j) to Bp(j + 1) − 1 do
13:        i ← Bi(p)
14:        if w(i) < j + 1 then
15:            w(i) ← j + 1
16:            Ci(nz + +) ← i + 1
17:            y(i) ← Bx(p)
18:        else
19:            y(i) ← y(i) + Bx(p)
20:        end if
21:    end for
22:    for p = Cp(j) to nz − 1 do
23:        Cx(p) ← y(Ci(p))
24:    end for
25: end for
26: Cp(n) ← nz + 1
```

B.2 Multiplication

Let A be a $m \times n$ sparse matrix and B be a $l \times n$ dense matrix and C be a $m \times n$ dense matrix which stores the result of the multiplication of A by B. Let Ap, Ai and Ax denote column pointer, row index and value vectors with CCS format, respectively. Algorithm B.3, sparse_dense_mul, shows the procedure for multiplying A by B.

Algorithm B.3 sparse_dense_mul (A, B)

```
1: for k = 1 to n do
2:     for j = 1 to l do
3:         for i = Ap(j) to Ap(j + 1) − 1 do
4:             C(Ai(i), k) ← C(Ai(i), k) + Ax(i) ∗ B(j, k)
5:         end for
6:     end for
7: end for
```

Let A be a $m \times l$ sparse matrix, B be a $l \times n$ sparse matrix and C be a $m \times n$ sparse matrix which stores the result of the multiplication of A by B. Let Ap, Bp and Cp denote column pointer vectors, Ai, Bi and Ci denote row index vectors, and Ax, Bx and Cx denote value vectors of A, B and C with CCS format, respectively. Algorithm B.4, sparse_sparse_mul, shows the procedure for multiplying A by B.

Algorithm B.4 sparse_sparse_mul (A, B)

```
 1: nz ← 0;   w ← (0, 0, ..., 0);   y ← (0, 0, ..., 0)
 2: for j = 1 to n do
 3:     Cp(j) ← nz + 1
 4:     for p = Bp(j) to Bp(j + 1) − 1 do
 5:         for q = Ap(Bi(p)) to Ap(Bi(p)) − 1 do
 6:             i ← Ai(p)
 7:             if w(i) < j + 1 then
 8:                 w(i) ← j + 1
 9:                 Ci(nz + +) ← i + 1
10:                 y(i) ← Bx(p) ∗ Ax(q)
11:             else
12:                 y(i) ← y(i) + Bx(p) ∗ Ax(q)
13:             end if
14:         end for
15:     end for
16:     for p = Cp(j) to nz do
17:         Cx(p) ← y(Ci(p))
18:     end for
19: end for
20: Cp(n) ← nz + 1
```

（平成 25 年 3 月 29 日受付）
（平成 25 年 8 月 21 日採録）