情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 1　73–84 (June 2008)

反復法ライブラリ向け 4倍精度演算の実装とSSE2を用いた高速化

小武守　恒[†1]　藤井昭宏[†2]
長谷川秀彦[†3]　西田晃[†4]

CG 法等のクリロフ部分空間法の収束性は丸め誤差に大きく影響される．収束の改善を図るには高精度演算が有効であるが計算時間が多くかかってしまう．我々は，反復解法ライブラリ Lis に double-double 精度を用いた 4 倍精度演算を実装し，SSE2 SIMD 命令を用いて高速化を行った．SSE2 に対して 2 段のループアンローリング等の高速化手法を適用し，計算時間は倍精度演算の約 3.5 倍，FORTRAN `REAL*16` の 0.2 倍程度となった．さらに，計算時間短縮のため反復解法中で必要なときだけ 4 倍精度演算を利用する DQ-SWITCH アルゴリズムを提案した．数値実験から DQ-SWITCH は適切なリスタート基準を決定できれば計算時間を短縮できることを示した．

Implementation of Fast Quad Precision Operation and Acceleration with SSE2 for Iterative Solver Library

Hisashi Kotakemori,[†1] Akihiro Fujii,[†2]
Hidehiko Hasegawa[†3] and Akira Nishida[†4]

The convergence of Krylov subspace methods, including CG method etc., are much influenced by the rounding errors. The high precision operation is effective for the improvement of convergence, however the arithmetic operations are costly. In this paper, we implemented the quadruple precision operations using the double-double precision for itaretive solver library Lis, and accelerated by using the SSE2 SIMD instruction. For speed-up of SSE2, we applied a loop unrolling etc.. The computation time of our implementation is 3.5 times as long as the Lis double precision, and is 0.2 times as long as FORTRAN `REAL*16`. Furthermore, we propose a DQ-SWITCH algorithm that efficiently uses the quadruple precision operations in order to reduce the computation time. The proposed method is shown to reduce computation time by numerical experiments.

1. はじめに

CG，BiCG 法等のクリロフ部分空間法は，理論的にはたかだか n 回（n は係数行列の次元数）の反復で収束する．しかしながら，倍精度演算では丸め誤差の影響のため収束までに多くの反復が必要となったり，停滞したりする．収束の改善には高精度演算，たとえば 4 倍精度演算が有効である[1)]．4 倍精度演算を利用するには FORTRAN の `REAL*16` を用いるのが簡単だが，倍精度演算と比較して 10 から 20 倍もの計算時間が必要となる．

反復法ライブラリの場合，一般には与えられる係数行列 A と右辺ベクトル b，初期ベクトル x_0 は倍精度であることに着目し，反復法内部の処理のみを 4 倍精度化し安定かつ高速な収束を達成することを考える．そのため，Bailey が提案した倍精度浮動小数点数を 2 個用いた “double-double” 精度のアルゴリズム[2)] を用い，SSE2 のような SIMD 命令を利用して高速化を図る．ここでの目的は反復法の内部処理の高精度化，高速化であり，外部とのデータのやりとりには倍精度を用いているため実装には自由度が生じる．このとき，入力として与えられる変数は倍精度，内部で定義される変数は 4 倍精度となるため実際は倍精度と double-double 精度の演算を高速化する必要がある．

関連研究として，double-double 精度浮動小数演算ライブラリには QD ライブラリ[3)] がある．QD は C++で記述されたライブラリである．さらに，倍精度浮動小数を 4 個用いた “quad-double” 精度のアルゴリズムを用いることで 8 倍精度演算も実現している．

4 倍精度演算をサポートした反復解法ライブラリとしては，GMM++（Generic Matrix Methods）[4)] がある．GMM++は C++で記述された密と疎行列に対するテンプレートライブラリで，4 倍精度演算については QD ライブラリを外部ライブラリとして呼び出して利用している．

Langou[5)] らは倍精度の直接解法で得られた解を 4 倍精度で反復改良することで，4 倍精

†1 株式会社 TCAD インターナショナル
TCAD International, Inc.
†2 工学院大学情報学部
Faculty of Informatics, Kogakuin University
†3 筑波大学大学院図書館情報メディア研究科
Graduate School of Library, Information and Media Studies, University of Tsukuba
†4 九州大学情報基盤研究開発センター
Research Institute for Information Technology, Kyushu University

度の直接解法より高速に解を求めている．4 倍精度演算には FORTRAN の `REAL*16` を用いている．

本論文では，文献 6)，7) をもとに double-double 演算の実装と SSE2 向け高速化法について詳述する．数値実験では，反復法に用いた場合の精度と高速性，混合精度反復法の可能性を示す．

2. 高速な 4 倍精度演算の実装

2.1 double-double 精度

Bailey の double-double 精度のアルゴリズムでは double-double 精度浮動小数 a を $a = a.hi + a.lo$，$\frac{1}{2}\mathrm{ulp}(a.hi) \geq |a.lo|$（上位 $a.hi$ と下位 $a.lo$ は倍精度）とし，4 倍精度演算は倍精度の四則演算の組合せで実現する．これは Dekker[8] と Knuth[9] のアルゴリズムに基づいている．$\mathrm{ulp}(x)$ は x の仮数部の “unit in the last place” を意味する[9]．double-double 精度は FORTRAN `REAL*16` よりも高速とされている[10]．しかし，FORTRAN `REAL*16` の表現形式[11] と比較して，指数部は倍精度と同じ 11 ビットで，仮数部は 104 ビットと 8 ビット少なくなっている．以下にアルゴリズムの概要を述べる．

2.2 4 倍精度加算

すべての演算は IEEE 倍精度演算で round-to-even 丸めと仮定する．x と y を倍精度とし，$x+y$ の倍精度加算の結果を $\mathrm{fl}(x+y)$ と表す．$\mathrm{err}(x+y)$ は $x+y = \mathrm{fl}(x+y) + \mathrm{err}(x+y)$ を満たすものとする．これより，$x+y$ の丸め誤差のない加算結果は 2 つの倍精度浮動小数 $\mathrm{fl}(x+y)$ と $\mathrm{err}(x+y)$ で表すことができる．Dekker と Knuth は**図 1** の方法で丸め誤差のない加算 $x+y$ ができることを示している．ここで，$s = \mathrm{fl}(x+y)$，$e = \mathrm{err}(x+y)$，$\frac{1}{2}\mathrm{ulp}(s) \geq |e|$ である．

図 1 の (I) と (II) を用いることで 4 倍精度加算 $a = b + c$ を計算できる．ただし，$a = (a.hi, a.lo)$，$b = (b.hi, b.lo)$，$c = (c.hi, c.lo)$ とする．4 倍精度加算は，まず b と c の上位 $b.hi$ と $c.hi$ に丸め誤差のない加算を行い：

$$b.hi + c.hi = \mathrm{fl}(b.hi + c.hi) + \mathrm{err}(b.hi + c.hi)$$

とする．ここで，$sh = \mathrm{fl}(b.hi + c.hi)$，$eh = \mathrm{err}(b.hi + c.hi)$ とする．次に，b と c の下位と eh の加算

$$eh = \mathrm{fl}(eh + b.lo + c.lo)$$

を行うと $sh + eh$ は 4 倍精度加算 $b + c$ の近似となる．今回の実装では高速な 4 倍精度演算を目的としているので下位の誤差 $\mathrm{err}(eh + b.lo + c.lo)$ を無視した．下位の足し合わせによって $\frac{1}{2}\mathrm{ulp}(sh) \geq |eh|$ にならない場合があるので，再度 sh と eh に丸め誤差のない加算をする．この結果 4 倍精度加算の演算数は 11 flops（floating operations）になる．**図 2** に 4 倍精度加算 $a = b + c$ の方法を示す．

(I) $|x| \geq |y|$ が仮定できる場合：

```
FAST_TWO_SUM(x,y,s,e) {
  s = x + y
  e = y - (s - x)
}
```

(II) $|x| \geq |y|$ が仮定できない場合：

```
TWO_SUM(x,y,s,e) {
  s = x + y
  v = s - x
  e = (x - (s - v)) + (y - v)
}
```

図 1　丸め誤差のない倍精度加算
Fig. 1　Rounding error free addition.

```
ADD(a,b,c) {
  TWO_SUM(b.hi,c.hi,sh,eh)
  eh = eh + b.lo + c.lo
  FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```

図 2　4 倍精度加算
Fig. 2　Quadruple precision addition.

2.3 4 倍精度乗算

加算の場合と同様に $x \times y = \mathrm{fl}(x \times y) + \mathrm{err}(x \times y)$ とする．Dekker は**図 3** の `TWO_PROD` を用いることで丸め誤差のない倍精度乗算 $x \times y$ が行えることを示している．ここで，$p = \mathrm{fl}(x \times y)$，$e = \mathrm{err}(x \times y)$，$\frac{1}{2}\mathrm{ulp}(p) \geq |e|$ である．倍精度の積和演算 $z := z + x \times y$ において（z は倍精度），Itanium2 や POWER5 のようにプロセッサが $x \times y$ を丸め誤差なしで計算した後，中間結果を 106 ビットで保持し，それに倍精度加算を行う積和命令をサポートしている場合は `TWO_PROD` の代わりに `TWO_PROD_FMA` を用いることで 17 flops から 3 flops と演算数を大幅に削減できる．`SPLIT` は倍精度浮動小数 x を $x = h + l$ に分割する．ただし，h は x の仮数部の上位の 26 ビット分を持ち l は残りの 26 ビット分を持つ．

図 1 と図 3 を用いることで 4 倍精度乗算 $a = b \times c$ を計算できる．4 倍精度乗算は，ま

特別な積和命令が利用できる場合：

```
TWO_PROD_FMA(x,y,p,e) {
  p = -x * y
  e = x * y + p
  p = -p
}
```

特別な積和命令が利用できない場合：

```
SPLIT(x,h,l) {
  t = 134217729.0 * x
  h = t - (t - x)
  l = x - h
}

TWO_PROD(x,y,p,e) {
  p = x * y
  SPLIT(x,xh,xl)
  SPLIT(y,yh,yl)
  e = ((xh*yh-p)+xh*yl+xl*yh)+xl*yl
}
```

図 3　丸め誤差のない倍精度乗算
Fig. 3　Rounding error free multiplication.

ず b と c の上位 $b.hi$ と $c.hi$ に丸め誤差のない乗算を行い：

$$b.hi \times c.hi = \mathrm{fl}(b.hi \times c.hi) + \mathrm{err}(b.hi \times c.hi)$$

とする．ここで，$p1 = \mathrm{fl}(b.hi \times c.hi)$，$p2 = \mathrm{err}(b.hi \times c.hi)$ とする．次に，b の上位と c の下位の乗算結果，b の下位と c の上位の乗算結果，$p2$ の加算

$$p2 = \mathrm{fl}(p2 + \mathrm{fl}(b.hi \times c.lo) + \mathrm{fl}(b.lo \times c.hi))$$

を行うと $p1 + p2$ は 4 倍精度乗算 $b \times c$ の近似となる．上記の加算によって $\frac{1}{2}\mathrm{ulp}(p1) \geq |p2|$ にならない場合があるので再度 $p1$ と $p2$ に丸め誤差のない加算をする．この結果 4 倍精度乗算の演算数は `TWO_PROD` を利用した場合 24 flops，`TWO_PROD_FMA` を利用した場合 10 flops になる．**図 4** に 4 倍精度乗算 $a = b \times c$ の方法を示す．

2.4　4 倍精度ベクトルの格納方法

4 倍精度ベクトルのデータ構造としては次の 2 通りが考えられる（`hi` は上位，`lo` は下位）．

- `hi` と `lo` を別の配列に格納する．
- `hi` と `lo` を交互に格納する．

```
MUL(a,b,c) {
  if 特別な積和命令が利用できない
    TWO_PROD(b.hi,c.hi,p1,p2)
  else
    TWO_PROD_FMA(b.hi,c.hi,p1,p2)
  endif
  p2 = p2 + (b.hi * c.lo)
  p2 = p2 + (b.lo * c.hi)
  FAST_TWO_SUM(p1,p2,a.hi,a.lo)
}
```

図 4　4 倍精度乗算
Fig. 4　Quadruple precision multiplication.

反復解法の高精度化の観点からは 4 倍精度のベクトルを倍精度のベクトルとしても利用できれば，アルゴリズムの可能性が広がる．前者では，`hi` を格納している配列のみ用いれば倍精度として扱うことができる．後者では，倍精度のベクトルとして利用する場合，`hi` の値が 1 つおきとなるため，4 倍精度版では行列ベクトル積（`matvec`），転置行列ベクトル積（`matvect`），ベクトルの内積（`dot`），ベクトルおよびその実数倍の加減（`axpy`）等のプログラムの修正が必要になる．そこで，今回は前者を採用した．

2.5　反復解法ライブラリ Lis への実装

疎行列に対するクリロフ部分空間法系統の反復解法の主要部は `matvec`，`matvect`，`dot`，`axpy` で実現されている．これらを 4 倍精度演算に置き換える．ただし，Lis[12] のユーザインタフェースはこれまでと同じインタフェースを保ちつつ，4 倍精度演算を実行するため

- 係数行列 A，右辺ベクトル b は倍精度
- 解ベクトル x の入出力は倍精度，反復解法中では 4 倍精度
- 反復解法中のベクトルとスカラーは 4 倍精度

とした．これにより，`matvec`，`matvect` では倍精度 $\odot$ 4 倍精度演算，`dot`，`axpy`，スカラー・スカラー演算では 4 倍精度 $\odot$ 4 倍精度演算が必要となる（$\odot$ は対応する演算子）．`matvec`，`matvect`，`dot`，`axpy` の主要な処理に対応させて，4 倍精度の積和演算関数 `FMA` と混合精度（4 倍精度と倍精度）の積和演算関数 `FMAD` を作成した．`FMA` は `dot` と `axpy` で，`FMAD` は `matvec` と `matvect` で利用される．

2 次元ポアソン方程式を有限差分で離散化した行列 $A1$（次数 1,000,000）に対して**表 1** の Xeon（2.8 GHz）上で実行した場合，倍精度演算の実行時間と比べて FORTRAN `REAL*16` では 22.32 倍，double-double 精度を用いた 4 倍精度演算を C でコーディングした場合で

表 1　計算環境
Table 1　Evaluation platforms.

CPU	Xeon 2.8 GHz	Opteron 2.2 GHz	Core2 Duo 2.4 GHz	POWER5 1.65 GHz
Cache	8 KB/512 KB/–	64 KB/1 MB/–	32 KB/4 MB/–	32 KB/1.9 MB/36 MB
Memory	1 GB	1 GB	2 GB	2 GB
Linux	2.4.20smp 32 bit	2.6.4smp 64 bit	2.6.9smp 64 bit	2.6.5smp 64 bit
Compiler	Intel C/C++ 9.0 Intel FORTRAN 9.0	Intel C/C++ 9.1 Intel FORTRAN 9.1		IBM XL C/C++ 7.0 IBM XL FORTRAN 9.1
Options	-O3	-O3		-O3

は 7.72 倍となる．そこで，高速化のために SSE2 SIMD 命令の利用を考える．

2.6　SSE2 による実装と高速化

SSE2 は Intel の Pentium4 に搭載された x87 命令に代わる高速化命令であり，128 bit のデータに対して SIMD 処理を行える．倍精度浮動小数なら同時に 2 つの演算が行えるので，2 倍の性能向上が期待できる．

4 倍精度演算関数 `ADD`，`MUL`，`FMA`，`FMAD` 等に対して SSE2 の組み込み関数を用いた `ADD_SSE2`，`MUL_SSE2`，`FMA_SSE2`，`FMAD_SSE2` 等を作成した．しかしながら，単に SSE2 を呼び出しただけでは計算の依存関係のため全体の 50%程度しか SSE2 の packed-double 命令で処理できず，高速化の効果は不十分である．

実際に `FMA` を使用する `dot`，`axpy`，`matvec`，`matvect` ではループ内で 1 回ずつ `FMA` が使われている．ここで 2 段のループアンローリングを行うとループ内で `FMA` が 2 回となり，SSE2 の packed-double 命令を用いて `FMA` を 2 個同時に処理できる．`a[0]`，`a[1]`，`b[0]`，`b[1]`，`q[0]`，`q[1]` を 4 倍精度型，`d[0]`，`d[1]` を倍精度型とし，以下のような積和演算を 2 個同時に実行する関数 `FMA2_SSE2`，`FMAD2_SSE2` を用意した：

```
FMA2_SSE2(a[0],a[1],b[0],b[1],q[0],q[1]) {
  a[0] = a[0] + b[0] * q[0]
  a[1] = a[1] + b[1] * q[1]
}
FMAD2_SSE2(a[0],a[1],b[0],b[1],d[0],d[1]) {
  a[0] = a[0] + b[0] * d[0]
  a[1] = a[1] + b[1] * d[1]
}
```

```
dot_fma2(x,y,dot) {
  t[0]=t[1]=0; //実際は XMM レジスタへロード
  for(i=0;i<n-1;i+=2)
    FMA2_SSE2(t[0],t[1],x[i],x[i+1],y[i],
      y[i+1]);
  ADD_SSE2(dot,t[0],t[1]);
  if(i!=n) FMA_SSE2(dot,x[i],y[i]);
}
```

図 5　`dot_fma2`（$dot = x^T y$）のコード
Fig. 5　Code segment of `dot_fma2` ($dot = x^T y$).

```
axpy_fma2(a,x,y) {
  aa[0]=aa[1]=a; //実際は XMM レジスタへロード
  for(i=0;i<n-1;i+=2)
    FMA2_SSE2(y[i],y[i+1],x[i],x[i+1],aa[0],
      aa[1]);
  if(i!=n) FMA_SSE2(y[i],x[i],a);
}
```

図 6　`axpy_fma2`（$y = y + \alpha x$）のコード
Fig. 6　Code segment of `axpy_fma2` ($y = y + \alpha x$).

実際の関数の中身は SSE2 の組み込み関数で記述している．また，それらを使う関数 `dot_fma2`，`axpy_fma2`，`matvec_fmad2`，`matvect_fmad2` を作成した（**図 5**，**図 6**，**図 7**，**図 8**）．`matvec_fmad2` と `matvect_fmad2` における行列の格納形式は CRS（Compressed Row Storage）[13] を仮定し，`A.ptr`，`A.index`，`A.value` はそれぞれ各行の開始位置を格納

```
matvec_fmad2(A,x,y) {
  for(i=0;i<n;i++) {
    t[0]=t[1]=0.0; //実際は XMM レジスタへロード
    for(j=A.ptr[i];j<A.ptr[i+1]-1;j+=2) {
      j0 = A.index[j];
      j1 = A.index[j+1];
      FMAD2_SSE2(t[0],t[1],x[j0],x[j1],
          A.value[j],A.value[j+1]);
    }
    ADD_SSE2(t[0],t[0],t[1]);
    for(;j<A.ptr[i+1];j++)
      FMAD_SSE2(t[0],x[A.index[j]],
          A.value[j]);
    y[i] = t[0];
  }
}
```

図 7　CRS に対する matvec_fmad2（$y = Ax$）のコード
Fig. 7　Code segment of matvec_fmad2 ($y = Ax$) for CRS format.

```
matvect_fmad2(A,x,y) {
  for(i=0;i<n;i++) y[i] = 0.0;
  for(i=0;i<n;i++) {
    t[0]=t[1]=x[i];//実際は XMM レジスタへロード
    for(j=A.ptr[i];j<A.ptr[i+1]-1;j+=2) {
      j0 = A.index[j];
      j1 = A.index[j+1];
      FMAD2_SSE2(y[j0],y[j1],t[0],t[1],
          A.value[j],A.value[j+1]);
    }
    for(;j<A.ptr[i+1];j++)
      FMAD_SSE2(y[A.index[j]],t[0],
          A.value[j]);
  }
}
```

図 8　CRS に対する matvect_fmad2（$y = A^T x$）のコード
Fig. 8　Code segment of matvect_fmad2 ($y = A^T x$) for CRS format.

する配列，非零要素の列番号を格納する配列，行列の非零要素の値を格納する配列である．

SSE2 では，メモリと 128 ビット XMM レジスタの間でデータのロードまたはストアが発生するとき，16 バイトにアライメントが合っていないと性能の低下が起こる．そこで，axpy_fma2，dot_fma2，matvec_fmad2，matvect_fmad2 のメモリアクセスに対して 16 バイトにアライメントが合っているかどうかを検討する．

- dot_fma2 ($dot = x^T y$)，axpy_fma2 ($y = y + \alpha x$)

2 つのベクトル x と y は先頭から 2 要素（16 バイト）ずつのアクセスとなり，ベクトルへのアクセスはすべて 16 バイトにアライメントが合っている．

- matvec_fmad2 ($y = Ax$)，
- matvect_fmad2 ($y = A^T x$)

A.value は前の行の非零要素数が奇数の場合，16 バイトにアライメントが合わなくなってしまう．また，matvec_fmad2 の場合，x が間接参照のため x[j0] と x[j1] が，matvect_fmad2 の場合，y が間接参照のため y[j0] と y[j1] が連続しているとは限らない．したがって，どちらも 16 バイトにアライメントを合わせることは困難である．

関数 dot_fma2，axpy_fma2，matvec_fmad2，matvect_fmad2 において，スカラー t または aa の値を for ループ中では XMM レジスタで保持するように各 FMA2_SSE2，FMAD2_SSE2 を修正することで，無駄なロード，ストア命令が削減できる．

2.7　性 能 評 価

表 1 の Core2 Duo（2.4 GHz）上で 2 次元ポアソン方程式を有限差分で離散化した行列 $A1$ の次数を 10,000 から 1,000,000 まで変化させたときの dot，axpy，matvec の 50 反復の実行時間を計測した．行列の格納形式は CRS を用いた．SSE2 を利用しない場合はコンパイル時に FMA が記述されている C ファイルに浮動小数の最適化を抑制するオプション -mp を追加している．

結果を**表 2** に示す．実行時間の単位は秒である．“non” は SSE2 を用いなかった場合，“naive” は SSE2 を素朴に適用した場合，“unrolling” は 2 段のアンローリングをして SSE2 を適用した場合，“all” は 2 段のアンローリングをして，配列を 16 バイトにアライメントを合わせ，スカラーの値を必要に応じて XMM レジスタで保持したうえで SSE2 を適用した場合である．この表から

- SSE2 を用いることで 1.53 倍から 1.83 倍の高速化
- SSE2 と 2 段のアンローリングで 1.80 倍から 3.14 倍の高速化
- すべての高速化技法によって 1.98 倍から 4.16 倍の高速化
- dot，axpy は 4 倍程度の良好な速度向上
- matvec は間接的なメモリアクセスのため 2 倍程度の速度向上

が分かる．このことより，前節で検討した高速化技法は有効であることが確認できた．

表 2　関数 50 反復の実行時間（Core2 Duo）．カッコ内は相対性能
Table 2　Execution time (relative performance) of 50 function iterations on Core 2 Duo.

SSE2		non		naive		unrolling		all
Function	Dimension	sec.		sec.		sec.		sec.
dot	10,000	0.01930	(1.00)	0.01063	(1.82)	0.00616	(3.14)	0.00469 (4.11)
	100,000	0.19425	(1.00)	0.10634	(1.83)	0.06227	(3.12)	0.04819 (4.03)
	1,000,000	1.96400	(1.00)	1.08260	(1.81)	0.63894	(3.07)	0.49737 (3.95)
axpy	10,000	0.01913	(1.00)	0.01068	(1.79)	0.00627	(3.05)	0.00460 (4.16)
	100,000	0.19287	(1.00)	0.10700	(1.80)	0.06320	(3.05)	0.04712 (4.09)
	1,000,000	1.94860	(1.00)	1.09049	(1.79)	0.67068	(2.91)	0.57694 (3.38)
matvec	10,000	0.10199	(1.00)	0.06582	(1.55)	0.05518	(1.85)	0.05071 (2.01)
	100,000	1.03388	(1.00)	0.67428	(1.53)	0.57554	(1.80)	0.52090 (1.98)
	1,000,000	10.37427	(1.00)	6.78081	(1.53)	5.77425	(1.80)	5.23550 (1.98)

3. 4 倍精度演算を用いた反復法

実際に Lis の疎行列反復解法において，今回実装した 4 倍精度演算の性能を評価するために，表 1 の環境で

- Lis の倍精度（DOUBLE）
- FORTRAN の `REAL*16`（行列，ベクトルすべて 4 倍精度）
- Lis の 4 倍精度 (表 2 の “all” を利用，Lis QUAD と表記)

の性能を前処理なしの BiCG 法を用いて比較した．行列の格納形式は CRS を用いた．

POWER5 プロセッサには SIMD 処理として Vector Multimedia Extension（VMX）命令があるが，倍精度演算はサポートされていないため SSE2 と同様な高速化は適用できない．しかしながら，丸め誤差なしで倍精度乗算した後，倍精度加算を行う積和命令をサポートしているので，`TWO_PROD_FMA` が利用可能である（Lis QUAD(`TWO_PROD_FMA`) と表記）．このとき，`FMA` が記述されている C ファイルに浮動小数の最適化を抑制するオプション`-qstrict` を追加した．

3.1 4 倍精度反復法の性能比較

行列 $A1$ の次数を 10,000 から 1,000,000 まで変化させ BiCG 法を 50 回反復させたときの実行時間と使用メモリ量を**表 3**，**表 4**，**表 5**，**表 6** に示す．

これらの表から Lis QUAD の相対性能は

- Core2 Duo では DOUBLE の 0.16 倍から 0.29 倍，FORTRAN `REAL*16` の 3.12 倍から 4.42 倍程度

表 3　BiCG 法 50 反復の実行時間とメモリ使用量（Core2 Duo）．カッコ内は相対性能
Table 3　Execution time (relative performance) and memory size of 50 BiCG iterations on Core2 Duo.

	DOUBLE		FORTRAN `REAL*16`		Lis QUAD	
Dimension	sec.	mem.	sec.	mem.	sec.	mem.
10,000	0.02017 (1.00)	1.0 MB	0.57517 (0.04)	2.0 MB	0.13012 (0.16)	1.5 MB
100,000	0.36900 (1.00)	9.9 MB	5.52724 (0.07)	19.8 MB	1.42318 (0.26)	15.3 MB
1,000,000	4.29637 (1.00)	99.2 MB	46.57581 (0.09)	198.4 MB	14.93291 (0.29)	152.6 MB

表 4　BiCG 法 50 反復の実行時間とメモリ使用量（Xeon）．カッコ内は相対性能
Table 4　Execution time (relative performance) and memory size of 50 BiCG iterations on Xeon.

	DOUBLE		FORTRAN `REAL*16`		Lis QUAD	
Dimension	sec.	mem.	sec.	mem.	sec.	mem.
10,000	0.05717 (1.00)	1.0 MB	2.01729 (0.03)	2.0 MB	0.25891 (0.22)	1.5 MB
100,000	0.82734 (1.00)	9.9 MB	20.09851 (0.04)	19.8 MB	2.49976 (0.33)	15.3 MB
1,000,000	8.44022 (1.00)	99.2 MB	199.23818 (0.04)	198.4 MB	25.26562 (0.33)	152.6 MB

表 5　BiCG 法 50 反復の実行時間とメモリ使用量（Opteron）．カッコ内は相対性能
Table 5　Execution time (relative performance) and memory size of 50 BiCG iterations on Opteron.

	DOUBLE		FORTRAN `REAL*16`		Lis QUAD	
Dimension	sec.	mem.	sec.	mem.	sec.	mem.
10,000	0.04402 (1.00)	1.0 MB	0.78882 (0.06)	2.0 MB	0.24998 (0.18)	1.5 MB
100,000	0.67599 (1.00)	9.9 MB	7.95484 (0.08)	19.8 MB	2.65616 (0.25)	15.3 MB
1,000,000	7.19525 (1.00)	99.2 MB	79.85377 (0.09)	198.4 MB	26.50909 (0.27)	152.6 MB

表 6　BiCG 法 50 回反復の実行時間とメモリ使用量（POWER5）．カッコ内は相対性能
Table 6　Execution time (relative performance) and memory size of 50 BiCG iterations on POWER5.

	DOUBLE		FORTRAN `REAL*16`		Lis QUAD(`TWO_PROD_FMA`)	
Dimension	sec.	mem.	sec.	mem.	sec.	mem.
10,000	0.04153 (1.00)	1.0 MB	0.46421 (0.09)	2.0 MB	0.30336 (0.14)	1.5 MB
100,000	0.49661 (1.00)	9.9 MB	4.71124 (0.11)	19.8 MB	3.07377 (0.16)	15.3 MB
1,000,000	7.09740 (1.00)	99.2 MB	47.85794 (0.15)	198.4 MB	33.23080 (0.21)	152.6 MB

- Xeon では DOUBLE の 0.22 倍から 0.33 倍，FORTRAN `REAL*16` の 7.79 倍から 8.04 倍程度
- Opteron では DOUBLE の 0.18 倍から 0.27 倍，FORTRAN `REAL*16` の 2.99 倍から 3.16 倍程度

```
for(k=0;k<100;k++)
for(i=0;i<N;i++){
  t = x[i] * y[i];
  for(j=0;j<LOOP;j++) t += t;
  z[i] += t;
}
```

図 9　演算とメモリアクセスの関係検証用コード
Fig. 9　A validation code.

表 7　図 9 のコードの実行時間（Core2 Duo）
Table 7　Execution time of Fig. 9 on Core2 Duo.

N	LOOP					
	0	2	4	6	8	10
10,000	0.00477	0.00485	0.00569	0.00730	0.00904	0.01087
1,000,000	0.70062	0.70736	0.72227	0.80890	0.95277	1.12531
$T_{1M} - T_{10K} \times 100$	0.22382	0.22206	0.15367	0.07850	0.04887	0.03791

- POWER5 では DOUBLE の 0.14 倍から 0.21 倍，FORTRAN `REAL*16` の 1.44 倍から 1.53 倍程度

である．上記のことより，Lis QUAD の実行時間は SSE2 が利用できる場合 DOUBLE の 2.99 倍から 6.45 倍，FORTRAN `REAL*16` の 0.12 倍から 0.33 倍，SSE2 が利用不可で，`TWO_PROD_FMA` が利用できる場合 DOUBLE の 4.68 倍から 7.30 倍，FORTRAN `REAL*16` の 0.65 倍から 0.69 倍の実行時間である．

倍精度演算の実行時間は，たとえば Xeon の場合，次数 10,000 から次数 1,000,000 に対して行列の次数に比例した実行時間より 47% 以上も増加しているが，4 倍精度演算の実行時間は行列の次数にほぼ比例している．積和演算処理を考えたとき，四則演算は加算と乗算の 2 回，メモリアクセスはロード 3 回，ストア 1 回である．倍精度演算の場合，四則演算とメモリアクセスの比は 1 : 2 となりメモリアクセスに性能が影響される．一方，4 倍精度演算の場合，四則演算は最低でも 10 flops 以上の倍精度演算で構成されているため，四則演算とメモリアクセスの比は 5 : 1 となる．このため，4 倍精度演算は倍精度演算と比較すると相対的にメモリアクセスの影響が軽減されると考えられる．実際に，**図 9** のコードを用いて Core2 Duo 上で $N = 10{,}000$，1,000,000 に対して `LOOP` を 0，2，4，6，8，10 と増加させたときの実行時間と $N = 10{,}000$ の実行時間 $\times 100$ と $N = 1{,}000{,}000$ の実行時間の差（$T_{1M} - T_{10K} \times 100$）を**表 7** に示す．ここで，`LOOP`$= 0$ は倍精度演算の積和演算を表し，`LOOP` の値を増加させることで 4 倍精度の積和演算の演算量に近づくことを意図している．この表より，`LOOP` の値が増加すると $T_{1M} - T_{10K} \times 100$ の値が小さくなっていることが分かる．このため，4 倍精度演算ではデータのロードがうまく隠蔽できるようになり，次数にほぼ比例した実行時間となったと思われる．

倍精度演算ではメモリアクセスに性能が影響されるため，各次数でのメモリ量とキャッシュの関係を調べる．次数 10,000，100,000，1,000,000 での必要メモリ量は 1.0 MB，9.9 MB，99.2 MB である．Xeon の場合，1 次，2 次キャッシュのサイズは 8 KB，512 KB であるので，次数 10,000 からすでに 2 次キャッシュに収まっていない．Opteron の場合，1 次，2 次キャッシュのサイズは 64 KB，1 MB であるので，次数 10,000 で 2 次キャッシュの容量が埋まってしまっている．Core2 Duo の場合，1 次，2 次キャッシュのサイズは 32 KB，4 MB であるので，次数 10,000 では 2 次キャッシュに収まっているが，次数 100,000 からは 2 次キャッシュに収まらない．POWER5 の場合，1 次から 3 次キャッシュのサイズはそれぞれ 32 KB，1.9 MB，36 MB であるので，次数 10,000 では 2 次キャッシュに，次数 100,000 では 3 次キャッシュに収まっている．すべての CPU に対して次数 1,000,000 ではキャッシュに収まっていない．これらのことより，実行時間の増加はデータがキャッシュに収まらずメモリアクセス時間が増加したためだと思われる．

また，Lis QUAD を 1 とした場合の FORTRAN `REAL*16` の相対実行時間が，たとえば次数 1,000,000 のとき Core2 Duo では 3.12 なのに対し，Xeon では 7.88 と 2 倍以上になっている．Intel Compiler の FORTRAN `REAL*16` は整数演算で実現されており Xeon は 32 Bit，Core2 Duo は 64 Bit の整数演算を行っているため，ほぼ倍の時間が必要と考えられる．実際，32 Bit モードでコンパイルした FORTRAN のコードを Core2 Duo 上で実行すると次数 1,000,000 の場合の実行時間は 93.79 秒となり 64 Bit モードの約 2 倍の実行時間でありビット数の違いが原因と分かる．

3.2　収束の改善

double-double 精度を用いた高速な 4 倍精度演算による収束の改善の効果を調べるため，係数行列 A として，Toeplitz 行列（次数：100,000）

表 8　行列 $A2$（$N = 10^5$）と $A3$ に対する BiCG 法の実行結果（Core2 Duo）
Table 8　Result of BiCG method for matrices $A2$（$N = 10^5$）and $A3$.

	DOUBLE			FORTRAN `REAL*16`			Lis QUAD		
Matrices	sec.	iter.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$	sec.	iter.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$	sec.	iter.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$
A2($\gamma = 1.0$)	0.35	58	5.81×10^{-13}	4.82	58	5.81×10^{-13}	1.49	58	5.81×10^{-13}
A2($\gamma = 1.1$)	0.42	70	7.05×10^{-13}	6.01	70	7.05×10^{-10}	1.79	70	7.05×10^{-13}
A2($\gamma = 1.2$)	0.51	86	8.80×10^{-13}	7.40	86	9.57×10^{-10}	2.21	86	9.57×10^{-13}
A2($\gamma = 1.3$)	–	–	–	9.70	113	7.82×10^{-13}	2.88	113	7.82×10^{-13}
A2($\gamma = 1.4$)	–	–	–	13.41	155	9.02×10^{-13}	3.94	155	9.02×10^{-13}
A3	–	–	–	103.22	1833	6.16×10^{-13}	16.42	1833	6.05×10^{-13}

$$A2 = \begin{pmatrix} 2 & 1 & & & & \\ 0 & 2 & 1 & & & \\ \gamma & 0 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix}, \tag{1}$$

ホール効果やイオンスリップといった効果を含めた一般化されたオームの法則から生じる電場ポテンシャル ϕ の $\nabla\cdot\tilde{\sigma}(-\nabla\phi + V \times B) = 0$ の関係から生じる対角性の悪いポアソン方程式[14] を離散化して生じる行列 $A3$（次数：23,994，非零要素数：214,060）を用いて数値実験を行った．ここで，$\tilde{\sigma}$ は導電率テンソル，V は速度ベクトル，B は磁場ベクトルである．

右辺ベクトル b は，$A2$ では $b = (1, \ldots, 1)^T$，$A3$ では離散化で生じたものを用いた．初期ベクトルは $x_0 = (0, \ldots, 0)^T$ とし，収束判定基準は $||r_{k+1}||_2/||r_0||_2 \leq 10^{-12}$ とした．Core2 Duo 上での行列 $A2$（γ を 1.0 から 1.4 まで変化）と $A3$ に対する前処理なし BiCG 法の結果を**表 8** に示す．“sec.” は実行時間（秒）を，“iter.” は反復回数を，$||b - Ax||_2/||b||_2$ は収束後に近似解 x_k を用いて改めて計算した相対残差の 2 ノルムを，“–” は 5,000 回の反復で収束しなかったことを意味する．この表から

- 行列 $A2$ に対して $\gamma = 1.2$ までは倍精度と 4 倍精度ともに収束している，
- 行列 $A2$（$\gamma = 1.3$ 以上）と $A3$ は倍精度では収束しないが，4 倍精度では収束している，
- FORTRAN `REAL*16` と Lis QUAD の収束はほぼ同等である，

ことが分かる．

これらのことより，我々の実装をクリロフ反復解法に適用した場合は FORTRAN `REAL*16` と数値的にほぼ同等の収束性であり，高精度化の目的は達成できている．FORTRAN `REAL*16` に対し，我々の SSE2 を用いた Lis QUAD は約 3 倍の高速化が図られている．

4. 混合精度算法 DQ-SWITCH

今回，入出力の係数行列 A と解 x と右辺 b は倍精度，反復解法内部での変数は 4 倍精度とし，SSE2 命令を用いた高速な 4 倍精度演算の実装を行ったが，それでも倍精度の 2.99 倍から 6.45 倍の計算時間が必要である．計算時間の短縮には，すべての反復を 4 倍精度演算で行うのではなく，必要なときだけ 4 倍精度演算を利用するようなアルゴリズムが必要となる．

その例として我々は，倍精度演算と高速な 4 倍精度演算を組み合わせた反復解法アルゴリズムとして DQ-SWITCH アルゴリズムを提案する．DQ-SWITCH は，倍精度演算で解いた解（あるいは途中の値）を初期値として高速な 4 倍精度演算で解く手法である．リスタート時に渡すのは生成された解 x_k のみで，その他の中間変数は渡さない．このアルゴリズムを**図 10** に示す．`nrm2` は相対残差ノルム，`restart_tol` はリスタートの判定基準，`tol` は収束判定基準である．2.4 節で述べたように，4 倍精度のベクトルは `hi` と `lo` が別々の配列に格納されているため，`hi` のみ用いれば倍精度のベクトルとして扱うことができる．したがって，倍精度と 4 倍精度のベクトルを個々に用意する必要も，リスタート時に解 x_k を倍精度ベクトルから 4 倍精度ベクトルへコピーする必要もない．

DQ-SWITCH で問題になるのは，`restart_tol` の決め方であるが，本論文ではいくつかの行列に対して `restart_tol` の値を変化させ，反復すべてを 4 倍精度演算で行うよりも高速化できることを示すにとどめ，その決定法については今後の課題とする．

4.1 数値実験

DQ-SWITCH の効果を BiCG 法を用いて比較した．**表 9** に DOUBLE では収束しなかった行列 $A2$（γ=1.3，1.4）に対する Core2 Duo 上での結果を示す．右辺ベクトルは $b=(1,\ldots,1)^T$，初期ベクトルは $x_0=(0,\ldots,0)^T$ とした．収束判定基準は $||r_{k+1}||_2/||r_0||_2 \leq 10^{-12}$ とした．表中の ϵ は DQ-SWITCH のリスタート基準 `restart_tol` を，“total” は倍精度演算と 4 倍精度演算の反復回数の合計を，“(double)” は倍精度演算の反復回数を意味する．この表から以下のことが分かる：

- DQ-SWITCH は Lis QUAD と比較して，最大 3.42 倍の高速化．
- 残差に大きな変化はない．
- リスタート基準 ϵ の値が小さいほど実行時間が短縮される傾向にある．
- $\gamma=1.4$ に対して 10^{-8} より小さくなると収束しないことから，ϵ は小さすぎてもいけない．

倍精度演算による BiCG 法の収束履歴を**図 11** に示す．$\gamma=1.3$ の場合は残差ノルムが 10^{-11} あたりまで減少した後，$\gamma=1.4$ の場合は残差ノルムが 10^{-9} あたりまで減少した後，収束が停滞している．これより，収束が停滞してから 4 倍精度に切り替えても効果がないことが分かる．DQ-SWITCH を活用するには停滞をうまく検出してリスタートを行うことが必要になる．

次に，応用問題で現れるいくつかの悪条件行列に対して検証する．行列 $A3$ と**表 10** に示す University of Florida Sparse Matrix Collection [15] の行列 $A4$ から $A6$ に対して前処理付き

```
for(k=0;k<最大反復回数;k++) {
  倍精度演算の反復解法
  if( nrm2<restart_tol ) break;
}
x 以外の作業変数をゼロクリア
for(k=k+1;k<最大反復回数;k++) {
  4 倍精度演算の反復解法
  if( nrm2<tol ) break;
}
```

図 10　DQ-SWITCH のアルゴリズム
Fig. 10　Algorithm of DQ-SWITCH.

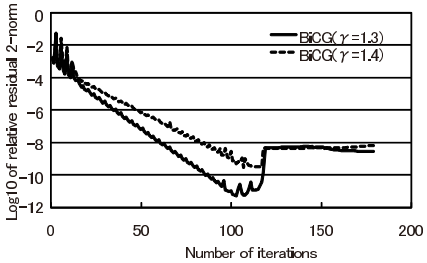

図 11　Toeplitz 行列に対する倍精度 BiCG 法の収束履歴
Fig. 11　Convergence history of double precision BiCG method for Toeplitz matrix.

表 9　行列 $A2$（$N=10^5$，$\gamma=1.3, 1.4$）に対する BiCG 法の収束性（Core2 Duo）
Table 9　Convergence of BiCG method for matrix $A2$ ($N=10^5$, $\gamma=1.3, 1.4$).

	$\gamma=1.3$				$\gamma=1.4$			
	iter.				iter.			
precision	total	(double)	sec.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$	total	(double)	sec.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$
Lis QUAD	113		2.88	7.82×10^{-13}	155		3.94	9.02×10^{-13}
DQ-SWITCH $\epsilon=10^{-3}$	114	(2)	2.87	6.78×10^{-13}	156	(2)	3.94	9.30×10^{-13}
$\epsilon=10^{-4}$	109	(11)	2.59	7.08×10^{-13}	152	(15)	3.62	8.20×10^{-13}
$\epsilon=10^{-5}$	105	(23)	2.26	6.80×10^{-13}	146	(31)	3.16	8.20×10^{-13}
$\epsilon=10^{-6}$	104	(35)	2.01	8.94×10^{-13}	138	(47)	2.67	7.65×10^{-13}
$\epsilon=10^{-7}$	95	(47)	1.58	6.28×10^{-13}	123	(65)	1.96	5.33×10^{-13}
$\epsilon=10^{-8}$	94	(61)	1.29	5.77×10^{-13}	119	(83)	**1.53**	5.92×10^{-13}
$\epsilon=10^{-9}$	95	(74)	1.08	7.99×10^{-13}	–			
$\epsilon=10^{-10}$	95	(86)	0.86	7.95×10^{-13}	–			
$\epsilon=10^{-11}$	103	(98)	**0.84**	2.95×10^{-13}	–			

表 10　テスト行列
Table 10　Test matrices.

Matrices	Dimension	Nonzeros	Discipline
$A4$：`ex8`	3,096	90,841	fluid dynamics
$A5$：`circuit_3`	12,127	48,137	circuit simulation
$A6$：`c-50`	22,401	180,245	non-linear optimization

表 11　悪条件問題に対する前処理付 BiCG 法の収束性（Core2 Duo）
Table 11　Convergence of preconditioned BiCG method for ill-conditioned problems.

Matrices	precision		precon.	iter. total	iter. (double)	sec.	$\|\|b-Ax\|\|_2/\|\|b\|\|_2$
$A3$	Lis QUAD		ILU(0)	488		11.18	3.46×10^{-13}
	DQ-SWITCH	$\epsilon=10^{-3}$	ILU(0)	610	(193)	10.32	5.14×10^{-13}
		$\epsilon=10^{-4}$	ILU(0)	617	(208)	**10.16**	5.37×10^{-13}
		$\epsilon=10^{-5}$	ILU(0)	–			
$A4$	Lis QUAD		SSOR	17098		165.55	6.65×10^{-8}
	DQ-SWITCH	$\epsilon=10^{-3}$	SSOR	23297	(12649)	127.68	8.01×10^{-8}
		$\epsilon=10^{-4}$	SSOR	22836	(13580)	**110.59**	7.85×10^{-8}
		$\epsilon=10^{-5}$	SSOR	24888	(15926)	111.35	7.87×10^{-8}
$A5$	Lis QUAD		ILUC	914		6.33	1.15×10^{-12}
	DQ-SWITCH	$\epsilon=10^{-3}$	ILUC	1730	(1227)	**5.52**	1.26×10^{-12}
		$\epsilon=10^{-4}$	ILUC	1888	(1455)	**5.52**	1.20×10^{-12}
		$\epsilon=10^{-5}$	ILUC	2042	(1590)	5.87	1.10×10^{-12}
$A6$	Lis QUAD		ILU(0)	2787		59.05	8.97×10^{-13}
	DQ-SWITCH	$\epsilon=10^{-3}$	ILU(0)	5380	(3368)	55.64	7.54×10^{-13}
		$\epsilon=10^{-4}$	ILU(0)	7674	(6252)	**54.35**	7.23×10^{-13}
		$\epsilon=10^{-5}$	ILU(0)	7705	(6279)	54.39	9.92×10^{-13}

BiCG 法を適用した．これらの行列は，倍精度では前処理を用いても収束しない問題である．前処理とは与えられた方程式 $Ax=b$ をより解きやすい等価な方程式 $K_1^{-1}AK_2^{-1}y=K_1^{-1}b$, $y=K_2x$ に直してから解くことであり，この場合，$K_1^{-1}AK_2^{-1}$，$K_1^{-1}b$ が反復解法にとっての入力データと見なせる．ここでは入力データの精度を倍精度として反復解法の内部処理のみを高精度化する方針をとっているから，前処理行列の生成は倍精度で行うことにした．実際，前処理行列は近似行列なので，精度の問題は起こりにくいと考えられる．次に，反復解法中で前処理を作用させる処理では，反復解法中のベクトルは 4 倍精度で与えられていて，前処理行列は倍精度であるので 4 倍精度と倍精度の混合演算を行うことになる．

右辺ベクトルは，$A4$ から $A6$ は $b=(1,\ldots,1)^T$，$A3$ では離散化で生じたものを用いた．初期ベクトルは $x_0=(0,\ldots,0)^T$ とし，収束判定基準は $||r_{k+1}||_2/||r_0||_2\le10^{-12}$ とした．結果を**表 11** に示す．表中の “precon.” は利用した前処理を示す．ILUC は Crout 版 ILU 前処理[16]，SSOR は SSOR 前処理[13] である．また，各行列で一番速く収束した結果を太字で示している．この表から以下のことが分かる：

- DQ-SWITCH は Lis QUAD と比較して，最大 1.5 倍の時間短縮．
- $A3$ では ϵ が 10^{-4} より小さくなると収束しない．

今回の実験では，$\epsilon=10^{-4}$ ですべての行列で安全かつ高速に収束することが確認できた．実用上は，問題に対するリスタート基準の決定法の検討が必要である．

5. ま と め

本論文では，反復解法ライブラリ Lis に対する double-double 精度を用いた 4 倍精度演算の実装と SSE2 を用いた高速化について述べた．4 倍精度演算は疎行列に対する反復解法を対象とし，入力の係数行列 A と右辺ベクトル b は倍精度のままで反復解法内部の変数を 4 倍精度としたことが特徴である．SSE2 に対する高速化として 2 段のループアンローリング，メモリアクセスに対して 16 バイトにアライメントを合わせる，`for` ループ中でスカラー値を XMM レジスタで保持する等を行った．これにより，行列ベクトル積，ベクトルの内積，ベクトルおよびその実数倍の加減で SSE2 を使わない double-double 精度の実装と比べて 1.98 倍から 4.16 倍の高速化が達成できた．

数値実験より SSE2 を用いた Lis の 4 倍精度 BiCG 法の計算時間はターゲットとしているキャッシュに収まらない大規模行列で比較すると，倍精度の 2.99 倍から 4.56 倍程度である．行列とベクトルすべてが 4 倍精度である FORTRAN `REAL*16` と比べ，行列 A と右辺 b は倍精度で反復解法内部の変数のみ 4 倍精度という我々の実装は，数値的な収束特性はほぼ同等かつメモリ増はわずかで，計算時間を $\frac{1}{2.99}$ から $\frac{1}{8.04}$ 程度に短縮できた．

収束の改善と計算時間の短縮のため，倍精度演算と 4 倍精度演算を組み合わせた反復解法 DQ-SWITCH アルゴリズムを提案し，その有効性をいくつかの例で示した．Lis の実装では 4 倍精度ベクトルのデータ構造を考慮することで，余計なオーバヘッドなしに DQ-SWITCH アルゴリズムが利用できる．DQ-SWITCH は適切なリスタート基準を決定できれば，4 倍精度演算の反復解法よりも計算時間を短縮できることを示した．このように，演算精度を上げれば，ILU 前処理等の処理の重い前処理を利用しなくても倍精度より少ない反復回数で収束する可能性があり，並列化による高速化の可能性も大きくなる．

今後の課題として，収束性の向上のため倍精度演算と 4 倍精度演算をうまく活用し，安定

かつロバストなライブラリにするため，DQ-SWITCH のようなアルゴリズムのリスタート基準の自動的な決定法の開発，より多くのテスト行列に対する検証等があげられる．

謝辞　本研究は，科学技術振興機構（JST）戦略的創造研究推進事業（CREST）「大規模シミュレーション向け基盤ソフトウェアの開発」プロジェクトの一部として実施した．ACS22 の査読者からは有益なコメントをいただきました．あわせて感謝いたします．

参 考 文 献

1) Hasegawa, H.: Utilizing the Quadruple-Precision Floating-Point Arithmetic Operation for the Krylov Subspace Methods, *The 8th SIAM Conference on Applied Linear Algebra* (2003).
2) Bailey, D.H.: A fortran-90 double-double library. http://www.nersc.gov/~dhbailey/mpdist/mpdist.html
3) Hida, Y., Li, X.S. and Bailey, D.H.: Algorithms for quad-double precision floating point arithmetic, *Proc. 15th Symposium on Computer Arithmetic*, pp.155–162 (2001).
4) Renard, Y. and Pommier, J.: GMM++ User Guide. http://www-gmm.insa-toulouse.fr/get fem/gmm_intro
5) Langou, J., et al.: Exploiting the Performance of 32 bit Floating Point Arithmetic in Obtaining 64 bit Accuracy (Revisiting Iterative Refinement for Linear Systems), SC06 Technical Paper (2006). http://sc06.supercomputing.org/schedule/pdf/pap254.pdf
6) 小武守恒，藤井昭宏，長谷川秀彦，西田　晃：SSE2 を用いた反復解法ライブラリ Lis 4 倍精度版の高速化，情報処理学会研究報告，2006-HPC-108, pp.7–12 (2006).
7) 小武守恒，藤井昭宏，長谷川秀彦，西田　晃：倍精度と 4 倍精度の混合型反復法の提案，HPCS2007, pp.9–16 (2007).
8) Dekker, T.: A floating-point technique for extending the available precision, *Numerische Mathematik*, Vol.18, pp.224–242 (1971).
9) Knuth, D.E.: *The Art of Computer Programming: Seminumerical Algorithms*, Vol.2, Addison-Wesley (1969).
10) Bailey, D.H.: High-Precision Floating-Point Arithmetic in Scientific Computation, *Computing in Science and Engineering*, pp.54–61 (2005).
11) Intel Fortran Compiler User's Guide Vol I.
12) http://www.ssisc.org/lis.
13) Barrett, R., et al.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM (1994).
14) Fujino, T., et al.: Influences of Electrical Conductivity of Wall on Magnethydrodynamic Control of Aerodynamic Heating, *Journal of Spacecraft and Rockets*, Vol.43, No.1, pp.63–70 (2006).
15) Davis, T.: UF Sparse Matrix Collection. http://www.cise.ufl.edu/research/sparse/matrices
16) Li, N., Saad, Y. and Chow, E.: Crout version of ILU for general sparse matrices, *SIAM J. Sci. Comput.*, Vol.25, pp.716–728 (2003).

(平成 19 年 10 月 9 日受付)
(平成 20 年 1 月 21 日採録)

小武守　恒（正会員）
1971 年生．1999 年岡山理科大学大学院理学研究科応用数学専攻博士後期課程修了．博士（理学）．科学技術振興機構研究員を経て，現在，株式会社 TCAD インターナショナル シニアエンジニア．数値計算法の研究に従事．日本応用数理学会会員．

藤井　昭宏（正会員）
1975 年生．1999 年東京大学理学部情報科学科卒業．2004 年東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程修了，博士（情報理工）．工学院大学 CPD センター講師を経て，現在，工学院大学情報学部講師．大規模線形問題に対するマルチレベルな解法や，ヘテロな計算環境への最適化技術に興味を持つ．

長谷川秀彦（正会員）
1958 年生．1983 年筑波大学大学院博士課程社会工学研究科中退．同年図書館情報大学助手．2002 年 10 月大学統合により筑波大学助教授．2007 年 11 月より筑波大学大学院図書館情報メディア研究科教授．博士（工学）．数値線形代数全般に興味を持つ．日本応用数理学会，SIAM，ACM 各会員．

西田　　晃（正会員）

1970 年生．1995 年東京大学理学部情報科学科卒業．1998 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．同年東京大学助手．2002 年より科学技術振興機構戦略的創造研究推進事業「大規模シミュレーション向け基盤ソフトウェアの開発」チーム研究代表者．中央大学 21 世紀 COE プログラム助教授，東京大学リサーチフェローを経て，現在九州大学情報基盤研究開発センター計算科学専門研究員．大規模数値解析，特に反復解法とその実装に興味を持つ．ACM，IEEE，SIAM，IACR，日本応用数理学会，日本ソフトウェア科学会，日本数学会各会員．