



KYUSHU UNIVERSITY 2011  
100th Anniversary

# 混合精度型反復解法の提案

九州大学情報基盤研究開発センター  
西田 晃





# はじめに

- きっかけ

- 前処理による反復解法計算の高速化(小柳義夫)
  - 4倍精度演算を利用した反復解法の精度向上(長谷川秀彦)
  - CREST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」(研究代表者:西田晃)で開発中の反復解法ライブラリ Lis に SSE を利用した4倍精度演算を実装(2006年春, 小武守恒)
  - グラフィック処理を目的としたプロセッサ (GPU, STI Cell B.E. Engine 等)では単精度演算性能 >> 倍精度演算性能
- 
- 「ある程度まで低精度で解いた後に高精度演算を使えば, 最初から高精度で反復解法を解くよりも早く解けるのではないか?」(2006年夏, 西田)



## これまでの外部発表

- 小武守恒, 藤井昭宏, 長谷川秀彦, 西田晃, 「高速な4倍精度演算を用いたクリロフ部分空間法の安定化」, 第12回日本計算工学会講演論文集, pp.631-634, 2007. (第12回日本計算工学会講演会ベストペーパーアワード受賞)
- 小武守恒, 「反復解法ライブラリー Lis の紹介」, 線形計算フォーラム, 九州大学情報基盤研究開発センター, 平成19年10月22日. (招待講演)
- 小武守恒, 藤井昭宏, 長谷川秀彦, 西田晃, 「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」, 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, pp.73-84, 情報処理学会, 2008年6月.  
等

# 反復解法ライブラリ Lis の開発(2002～)

## – 連立一次方程式解法

- 構造, 流体解析など多くのアプリケーション
- 反復解法とその前処理手法に関する研究成果を実問題に適用したい
- 計算機環境の高並列化  
→スケラブルなアルゴリズムが必要
- さまざまな解法, 前処理, 及び疎行列格納形式に対応した並列反復解法ライブラリ Lis (A Library of Iterative Solvers for Linear Systems, 仏語でユリの意.) を開発. BSDライセンスに基づくオープンソースソフトウェアとしてコードを公開中  
(<http://www.ssisc.org/lis/>)
- デスクトップ PC から大規模並列計算機まで, 様々な環境で利用可能

# 反復解法ライブラリ Lis

Lis 1.1.2 に対応している解法, 前処理手法, 及び行列格納形式の一覧

1.0.x	CG	1.1.2 で追加	CR
	BiCG		BiCR
	CGS		CRS
	BiCGSTAB		BiCRSTAB
	BiCGSTAB(l)		GPBiCR
	GPBiCG		BiCRSafe
	Orthomin(m)		FGMRES(m)
	GMRES(m)		IDR(s)
	TFQMR		
	Jacobi		
	Gauss-Seidel		
	SOR		

1.0.x	Jacobi	1.1.0 で追加	Crout版ILU
	ILU(k)		ILUT
	SSOR		Additive schwarz
	Hybrid		
	I+S		ユーザ定義前処理
	SA-AMG		
	SAINV		

Point	Compressed Row Storage
	Compressed Column Storage
	Modified Compressed Sparse Row
	Diagonal
	Ellpack-Itpack generalized diagonal
	Jagged Diagonal Storage
	Dense
Coordinate	
Block	Block Sparse Row
	Block Sparse Column
	Variable Block Row



# Lis を用いたプログラミング例

```
1: LIS_MATRIX      A;
2: LIS_VECTOR      b, x;
3: LIS_SOLVER      solver;
4: int             iter;
5: double           times, itimes, ptimes;
6:
7: lis_initialize(argc, argv);
8: lis_matrix_create(LIS_COMM_WORLD, &A);
9: lis_vector_create(LIS_COMM_WORLD, &b);
10: lis_vector_create(LIS_COMM_WORLD, &x);
11: lis_solver_create(&solver);
12: lis_input(A, b, x, argv[1]);
13: lis_vector_set_all(1.0, b);
14: lis_solver_set_optionC(solver);
15: lis_solve(A, b, x, solver);
16: lis_solver_get_iters(solver, &iter);
17: lis_solver_get_times(solver, &times,
18: printf("iter = %d time = %e (p=%e
19: lis_finalize());
```



# Lis ビルド手順

ファイルのダウンロード

<http://www.ssiscc.org/lis/>

ファイルの展開

```
>gunzip -c lis-1.1.2.tar.gz | tar xvf -
```

configureスクリプトの実行

```
>./configure
```

makeの実行

```
>make
```

インストール

```
>make install
```

## configureオプション

OpenMPを利用	--enable-omp
MPIを利用	--enable-mpi
FORTTRAN APIを利用	--enable-fortran
SA-AMG前処理を利用	--enable-saamg
4倍精度演算を利用	--enable-quad



## Krylov 部分空間法の丸め誤差

- CG法, BiCG法等: 理論的には高々行列の次数回の反復で収束
- 実際には丸め誤差の影響で多くの反復回数が必要(または収束が停滞)
- 低コストな高精度演算の必要性
- Fortran の REAL\*16 では倍精度の10~20倍の計算時間



## 4倍精度演算の実装方針

- Bailey の提案による “double-double” 精度アルゴリズムを利用
- SIMD 命令 (Intel SSE2 など) を利用して高速化
- IEEE754 4倍精度規格とは非互換
- 反復法内部の処理のみを4倍精度化
- 外部とのデータのやりとりには倍精度を使用



## 関連研究

- “double-double”精度ライブラリ
  - QD Library (Hida et al.): C++ で記述. 倍精度浮動小数  $\times 4$  で8倍精度もソフトウェア的に実現.
  - GMM++ (Renard et al.): C++ で記述. 4倍精度密・疎行列向け反復解法を QD ライブラリを用いて実現.
- Fortran REAL\*16の利用
  - 倍精度直接解法の反復改良に4倍精度演算を適用 (Langou et al.)

## “double-double” 精度演算 (加算)

- 倍精度演算は round-to-even を仮定
- 加算の場合
  - 丸め誤差のない倍精度加算 (Dekker, Knuth)

- $|x| \geq |y|$  が仮定できる場合

```
FAST_TWO_SUM(x, y, s, e) {
```

```
    s = x + y
```

```
    e = y - (s - x)
```

```
}
```

- $|x| \geq |y|$  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)
```

```
}
```



## “double-double” 精度演算 (加算の続き)

- 以上を  $a(a.\text{hi}, a.\text{lo}) = b(b.\text{hi}, b.\text{lo}) + c(c.\text{hi}, c.\text{lo})$  の計算に適用
  - 倍精度加算  $sh = fl(b.\text{hi} + c.\text{hi})$
  - 誤差  $eh = err(b.\text{hi} + c.\text{hi})$から
  - $eh = fl(eh + b.\text{lo} + c.\text{lo})$
  - を計算, 丸め誤差のない (=極めて微小な) 加算  $sh + eh$  を  $b + c$  の近似とする ( $err(eh + b.\text{lo} + c.\text{lo})$  は微小なので無視)
- 加算アルゴリズム

```
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)
}
```



## “double-double” 精度演算 (乗算)

- 乗算の場合

- 丸め誤差のない倍精度乗算 (Dekker)

- Itanium, POWER

- 丸め誤差なしに乗算後, 104ビットで中間結果を保持, 倍精度加算を行う積和命令を持つ

- 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  
}
```

## “double-double” 精度演算 (乗算続き)

- 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  
}
```



## “double-double” 精度演算 (乗算続き)

- 以上を  $a(a.\text{hi}, a.\text{lo}) = b(b.\text{hi}, b.\text{lo}) * c(c.\text{hi}, c.\text{lo})$  の計算に適用
  - 倍精度乗算  $p1 = \text{fl}(b.\text{hi} * c.\text{hi})$
  - 誤差  $p2 = \text{err}(b.\text{hi} * c.\text{hi})$

から

$$p2 = \text{fl}(p2 + \text{fl}(b.\text{hi} * c.\text{lo}) + \text{fl}(b.\text{lo} * c.\text{hi}))$$

を計算, 丸め誤差のない加算  $p1 + p2$  を  $b * c$  の近似とする

- 4倍精度乗算

```
MUL(a, b, c) {  
    if 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)  
}
```



## 反復解法ライブラリ Lis への実装

- 4倍精度ベクトルの格納
  - hi と lo を別の配列に格納
  - hi を格納している配列のみを用いれば倍精度ベクトルとして扱える利点
- 行列-ベクトル間は倍精度 × 4倍精度演算関数 (FMAD), ベクトル間, スカラー間は4倍精度演算関数 (FMA) を定義して実装
- 2次元 Poisson 方程式 (次数1,000,000) の場合, Pentium 4 Xeon での計算時間は倍精度に比べて Fortran REAL\*16 (Intel Compiler 9.0) で22.32倍, double-double で7.72倍

# SIMD 命令を利用した高速化

- Intel SSE2
  - Pentium 4 以降の Intel プロセッサに搭載された SIMD 命令
  - 128ビットのデータに対して SIMD 命令が実行可能
  - 倍精度浮動小数なら同時に2演算を実行可能
- 2段のループアンローリングを実装

```
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]  
}
```
- XMMレジスタを利用して無駄なロード、ストア命令を削減



## SIMD 命令を利用した高速化 (dot, axpy)

```
dot_fma2(x, y, dot) {  
    t[0] = t[1] = 0;          // XMM レジスタにロード  
    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]);  
}
```

```
axpy_fma2(a, x, y) {  
    aa[0] = aa[1] = a;      // XMM レジスタにロード  
    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);  
}
```

# SIMD 命令を利用した高速化 (matvec)

```
matvec_fmadv2(A, x, y) {  
    for (i=0; i<n; i++) {  
        t[0] = t[1] = 0.0;           // XMM レジスタにロード  
        for (j=A.ptr[i]; j<A.ptr[i+1]-1; j+=2) {  
            j0 = A.index[j];  
            j1 = A.index[j+1];  
            FMADV2_SSE2(t[0], t[1], x[j0], x[j1], A.value[j],  
                        A.value[j+1]);  
            y[i] = t[0];  
        }  
    }  
}
```



# 性能評価

- 評価環境

CPU	Xeon 2.8GHz	Opteron 2.2 GHz	Core2 Duo 2.4GHz	POWER5 1.65 GHz
Cache	8KB/512KB/-	64KB/1 MB/-	32KB/4 MB/-	32KB/1.9MB/36MB
Memory	1GB	1GB	2GB	2GB
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
Options	9.1 -O3		-O3	-O3

## 性能評価（要素演算）

Core 2 Duo を利用して2次元 Poisson 方程式を離散化した行列 A1を10,000から1,000,000まで次数を変化させて dot, axpy, matvec の50反復の実行時間を計測

- SSE2 を用いることにより, 1.53倍から1.83倍の高速化
- SSE2, 2段のアンローリングの併用により, 1.80倍から3.14倍の高速化
- SSE2, 2段のアンローリング, 128ビット XMM レジスタのための16バイト・アラインメント, XMM レジスタを利用したロード, ストア命令の削減を併用することにより, 1.98倍から4.16倍の高速化(dot, axpy は4倍程度, matvec は間接参照のため2倍程度)

## 性能評価(4倍精度反復法)

Lis 4倍精度版の性能を Lis 倍精度版, Fortran REAL\*16 と比較(行列A1をサイズ10,000から1,000,000まで変化させ, 前処理なしの BiCG 法の実行時間を計測)

- Core 2 Duo では倍精度版の0.16倍から0.29倍, Fortran REAL\*16 の3.12倍から4.42倍程度
- Xeon では倍精度版の0.22倍から0.33倍, Fortran REAL\*16 の7.79倍から8.04倍程度
  - Intel Compiler では Fortran REAL\*16 は整数演算で実装. Core 2 Duo の64ビット整数演算に対し, Xeon では32ビット演算.
- Opteron では倍精度版の0.18倍から0.27倍, Fortran REAL\*16 の2.99倍から3.16倍程度
- POWER5 では倍精度版の0.14倍から0.21倍, Fortran REAL\*16 の1.44倍から1.53倍程度(TWO\_PROD\_FMA を使用)
- Fortran REAL\*16 と Lis 4倍精度版の収束特性はほぼ同等





# 混合精度解法の提案

- ここまでの工夫により, 倍精度の2.99倍から6.46倍の時間で4倍精度演算が可能
- 倍精度で解いた解(または途中の反復ベクトル)を初期値として4倍精度演算で解いてはどうか

## DQ-SWITCH アルゴリズム

```
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 以外の作業領域をゼロクリア  
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;  
}
```

# 数値実験 (Toeplitz 行列)

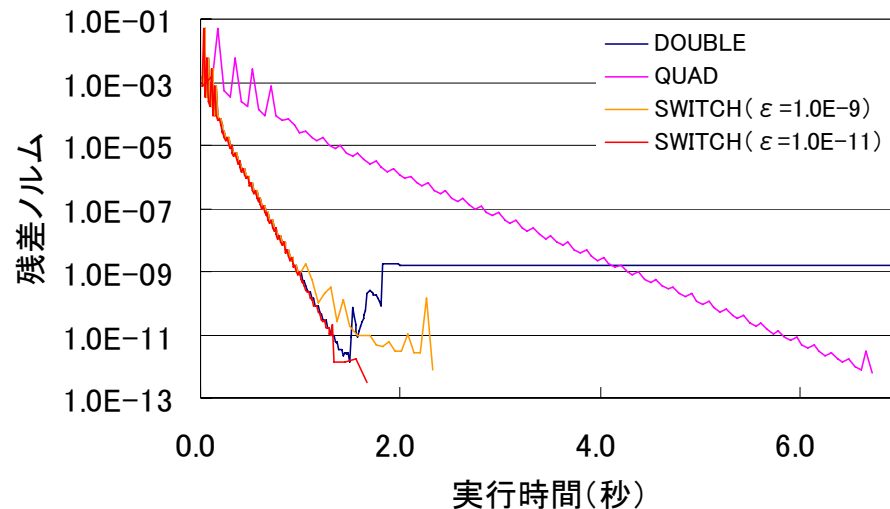
- A2 ( $\gamma=1.3, 1.4$ ) に対して Core 2 Duo 上で評価
- 右辺ベクトル  $b=(1, \dots, 1)^T$ , 初期ベクトル  $x_0=(0, \dots, 0)^T$
- 収束判定基準  $\|r_{k+1}\|_2 / \|r_0\|_2 \leq 10^{-12}$

行列 A2 に対する BiCG 法の収束性

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

# 数値実験 (Toeplitz 行列 続き)

- DQ-SWITCH は4倍精度版と比較して, 最大3.42倍の高速化
- リスタート基準は小さいほど計算時間は短縮
- 停滞, 発散を検知する必要



# 数値実験 (Univ. Florida Sparse Matrix Collection)

テスト行列 ( Univ. Florida Sparse Matrix Collection から倍精度では前処理を用いても収束しないものを選択)

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

悪条件問題に対する前処理付BiCG 法の収束性 (Core2 Duo 上で測定)

Matrices	precision	precon.	Total (double)	sec.	$\ b - Ax\ _2 / \ b\ _2$
A4	4倍精度版	SSOR	17098	165.55	$6.65 \times 10^{-8}$
	DQ-SWITCH				
	$\varepsilon = 10^{-3}$	SSOR	23297 (12649)	127.68	$8.01 \times 10^{-8}$
	$\varepsilon = 10^{-4}$	SSOR	22836 (13580)	<b>110.59</b>	$7.85 \times 10^{-8}$
	$\varepsilon = 10^{-5}$	SSOR	24888 (15926)	111.35	$7.87 \times 10^{-8}$
A5	4倍精度版	ILUC	914	6.33	$1.15 \times 10^{-12}$
	DQ-SWITCH				
	$\varepsilon = 10^{-3}$	ILUC	1730 (1227)	<b>5.52</b>	$1.26 \times 10^{-12}$
	$\varepsilon = 10^{-4}$	ILUC	1888 (1455)	<b>5.52</b>	$1.20 \times 10^{-12}$
	$\varepsilon = 10^{-5}$	ILUC	2042 (1590)	5.87	$1.10 \times 10^{-12}$
A6	4倍精度版	ILU(0)	2787	59.05	$8.97 \times 10^{-13}$
	DQ-SWITCH				
	$\varepsilon = 10^{-3}$	ILU(0)	5380 (3368)	55.64	$7.54 \times 10^{-13}$
	$\varepsilon = 10^{-4}$	ILU(0)	7674 (6252)	<b>54.35</b>	$7.23 \times 10^{-13}$
	$\varepsilon = 10^{-5}$	ILU(0)	7705 (6279)	54.39	$9.92 \times 10^{-13}$



## 数値実験 (Univ. Florida Sparse Matrix Collection 続き)

- 右辺ベクトル  $b=(1,\dots,1)^T$ , 初期ベクトル  $x_0=(0,\dots,0)^T$
- 収束判定基準  $\|r_{k+1}\|_2/\|r_0\|_2 \leq 10^{-12}$
- 前処理行列は倍精度で生成
- 前処理行列-反復ベクトル間は倍精度-4倍精度の混合演算
- DQ-SWITCH は4倍精度版と比較して, 最大で33%の時間短縮
- $\varepsilon = 10^{-4}$  ですべての行列が収束



## まとめ

- “double-double” 精度演算を SIMD 命令により高速化
  - 内部演算を4倍精度で処理
  - 要素演算で1.98倍から4.16倍の高速化
- 反復解法ライブラリ Lis に実装, 評価
  - 4倍精度版 Lis では, BiCG 法で倍精度に比べて2.99倍から4.56倍の高速化
  - Fortran REAL\*16 に比べて, 2.99倍から8.04倍の高速化
- DQ-SWITCH アルゴリズムの提案
  - 4倍精度ベクトルのデータ構造を工夫
  - オーバーヘッドなしに DQ-SWITCH アルゴリズムが利用可
  - リスタート基準の決定が課題



## 最後に

- 反復解法ライブラリ Lis
  - <http://www.ssisc.org/lis/> でソフトウェアを公開中！
  - 今秋に新版を公開予定
  - 九大センター計算機システムに対応