倍精度と4倍精度の混合型反復法の提案

小武守 恒（JST・東京大学）
藤井 昭宏（工学院大学）
長谷川 秀彦（筑波大学）
西田 晃（中央大学・JST）

発表の流れ

- はじめに
- double-double精度演算
  - 反復解法ライブラリLisへの実装
  - SSE2による高速化
- 倍精度と4倍精度の混合型反復法
- 数値実験
  - 収束性
  - 解の精度
- まとめ

はじめに

- クリロフ部分空間法は，理論的には高々行列の次数回の反復で収束
- 丸め誤差の影響で収束までに多くの反復が必要または停滞
- 収束の改善には高精度演算，例えば4倍精度演算が有効であるが計算コスト大
  - メモリ2倍　演算時間20倍

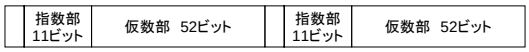
はじめに（続き）

- 高速な4倍精度演算
  - 厳密な4倍精度ではなく高速な4倍精度
  - 倍精度浮動小数点数を2個用いるdouble-double精度演算を採用
  - SSE2を活用して高速化
  - 演算時間4.5倍
- 混合型反復法の提案
  - 倍精度演算と高速な4倍精度演算を組み合わせてさらなる高速化

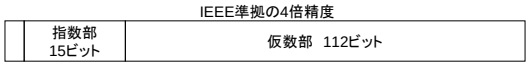
double-double 精度演算

- 倍精度浮動小数を2個用いて4倍精度を実現
  - double-double精度a = a.hi + a.lo, |a.hi|>|a.lo|
  - 仮数部がIEEE準拠より8ビット少ない
  - 有効桁数
    - double-double精度　約32桁
    - IEEE準拠4倍精度　約33桁

double-double精度



IEEE準拠の4倍精度



丸め誤差のない倍精度加算

- 丸め誤差のない倍精度加算は2つの倍精度で表せる

$$a + b = fl(a + b) + err(a + b)$$

  - a,b：倍精度浮動小数
  - fl(a + b) ：a + bの倍精度浮動小数加算
  - err(a+b)：(a+b) – fl(a+b)

丸め誤差のない倍精度加算の計算方法

- Dekkerは以下の2つの方法で丸め誤差のない倍精度加算a+bが可能なことを示している

|a|>=|b|の場合

```
FAST_TWO_SUM(a,b,s,e)
    s = a + b
    e = b - (s - a)
```

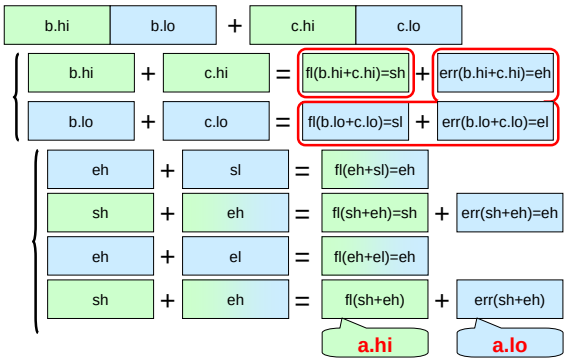
|a|>=|b|でない場合

```
TWO_SUM(a,b,s,e)
    s = a + b
    v = s - a
    e = (a - (s - v)) + (b - v)
```

– s = fl(a+b), e = err(a+b)

– |s|>|e|

4倍精度加算a=b+c



4倍精度加算a=b+c

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

a=(a.hi,a.lo), b=(b.hi,b.lo), c=(c.hi,c.lo)

Lisでの実装方針

- 係数行列A，右辺ベクトルbの入力は高々倍精度
- 倍精度と同一のインタフェース
  - 入力（係数行列A，右辺ベクトル b，初期ベクトル x_0 ）は倍精度
  - 反復解法中の解ベクトル x，補助ベクトル,スカラーは4倍精度
  - 出力は倍精度，4倍精度でも可能

実装方法

- 反復解法を4倍精度演算に置き換える
  - 行列ベクトル積（matvec）
  - ベクトルの内積（dot）
  - ベクトルおよびその実数倍の加減（axpy）
- matvec,dot,axpyの主な処理は積和演算
  - MULとADDの関数をまとめることでメモリストアを削減
- 2つの関数の作成
  - FMA（Floating Multiply-Add）：4倍精度の積和演算
    - dotとaxpyで利用
  - FMAD：混合精度（4倍精度と倍精度）の積和演算
    - matvecで利用

SSE2による高速化

- ベクトル単位（dot, axpy, matvec）のSSE2化
  - 2個同時に積和演算を
- ループ内に1回ずつFMAが使われているので2段のループアンローリングを行うとFMAが2回となる
  - すべてSSE2のpd命令で処理できる
- 2個同時に積和演算をする関数FMA2_SSE2, FMAD2_SSE2を用意

スピードテスト

- 計算環境

CPU	Xeon 2.8GHz
OS	Linux 2.4.20smp
Compiler	Intel C++ 7.0 Intel Fortran 9.0

  - 最適化オプションは-O3
  - FMAが記述されているCファイルは浮動小数の最適化を抑制(-mp)
- 2次元ポアソン方程式を有限差分で離散化
  - 行列A1（次数1,000,000）
  - 行列格納形式はCRS

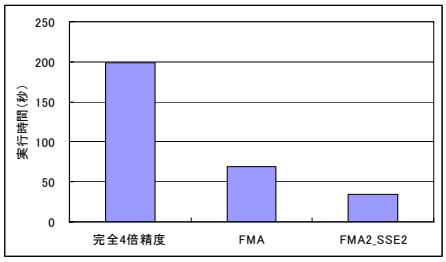
必要メモリ量

	倍精度	Lisの4倍精度	完全4倍精度
行列A(CRS)	4(n+nnz) +8nnz	4(n+nnz) +8nnz	4(n+nnz) +16nnz
ベクトルb	8n	8n	16n
ベクトルx	8n	16n	16n
補助ベクトル(BiCG)	6*8n	6*16n	6*16n
行列A1の場合の合計	121.9MB	175.8MB	221.6MB

n:次数　nnz:非零要素数

- 完全4倍精度:Fortranコンパイラによる4倍精度

実行時間（BiCG法50回反復）



- 完全4倍精度との比較で5．71倍の高速化
- FMAとの比較で1.97倍の高速化

倍精度 と 4倍精度 の混合型反復法

- 4倍精度演算の削減
  - 計算時間の短縮
- 2つの混合型反復法
  - SWITCHアルゴリズム：　“リスタート”
  - PERIODICアルゴリズム:“一定間隔で修正”

SWITCHアルゴリズム

- 倍精度の解（あるいは途中の値）を初期値として4倍精度で反復
  - リスタート時に渡すのは倍精度演算で生成された解x_kのみ

```
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
}
```

PERIODICアルゴリズム

- 一定間隔ごと4倍精度で反復を行う
  - 精度切り替え時にすべての変数を渡す
  - 4倍精度から倍精度への切り替えはコストなし
  - 倍精度から4倍精度への切り替えは反復計算に必要なベクトルの下位をゼロクリア

```
for(k=0;k<最大反復回数;k++) {
  if( k%interval<num ) {
    4倍精度演算の反復解法
  } else {
    必要なベクトルの下位をゼロクリア
    倍精度演算の反復解法
  }
```

数値実験

- 前処理なしBiCG法で比較
  - FMA2_SSE2(QUAD)
  - SWITCH
  - PERIODIC

収束性の比較

- Toeplitz行列
  - 次数：100,000
  - γを大きくすると解きにくくなる

$$\begin{pmatrix} 2 & 1 & & & & \\ 0 & 2 & 1 & & & \\ \gamma & 0 & 2 & 1 & & \\ & \ddots & \ddots & \ddots & \ddots & \\ & & \gamma & 0 & 2 & 1 \\ & & & \gamma & 0 & 2 \end{pmatrix}$$

- 右辺ベクトルbは$(1,\ldots,1)^T$
- 初期ベクトルx_0は$(0,\ldots,0)^T$
- 収束判定基準は相対残差ノルム10^{-12}

γ=1.3の場合

		iter. total	iter. double	sec.	‖b−Ax‖
FMA2_SSE2		113	0	6.60	2.47E−10
SWITCH	ε =1.0E−09	95	74	2.33	2.53E−10
	ε =1.0E−10	95	86	1.82	2.51E−10
	ε =1.0E−11	103	100	1.67	9.34E−11
PERIODIC	num=1	−	−		
	num=2	−	−		
	num=3	−	−		
	num=4	−	−		
	num=5	107	52	3.98	3.16E−10
	num=6	118	46	4.89	2.02E−10

- ||b-Ax||は収束後に近似解xで改めて計算したノルム
- ε はSWITCHのリスタート基準
- numは10反復の内4倍精度反復がnum回となる

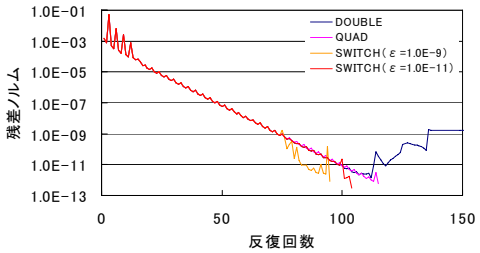
γ=1.4の場合

		iter. total	iter. double	sec.	‖b−Ax‖
FMA2_SSE2		155	0	8.77	2.85E−10
SWITCH	ε =1.0E−07	119	65	4.04	2.51E−10
	ε =1.0E−08	115	83	3.06	3.04E−10
	ε =1.0E−09	−			
PERIODIC	num=1	−	−		
	num=2	−	−		
	num=3	−	−		
	num=4	−	−		
	num=5	−	−		
	num=6	−	−		

混合型反復法の評価

- SWITCHはgood
  - FMA2_SSE2と比較して，最大3.95倍の高速化
  - ε の値を小さくとると実行時間が短縮される傾向にある
- PERIODICはno good
  - 収束しないことが多い
  - numの値は大きくとる必要がある

γ=1.3の場合の残差履歴



- 停滞してから4倍精度に切り替えても効果がない
- 停滞をうまく検出してリスタートを行うことが好ましい

精度の比較

- University of Florida Sparse Matrix Collection

行列	次数	非零要素数	メモリ	
			倍精度	Lis Quad
airfoil_2d	14,214	259,688	3.9MB	4.7MB
wang3	26,064	177,168	3.7MB	5.1MB
language	399,130	1,216,334	39.8MB	61.1MB

- 右辺ベクトルbは$(1,\ldots,1)^T$
- 初期ベクトルx_0は$(0,\ldots,0)^T$
- 収束判定基準は相対残差ノルム10^{-12}

airfoil_2d		iter. total	iter. double	sec.	‖b−Ax‖
DOUBLE		4567	4567	18.64	3.25E−08
QUAD		3838		69.39	5.36E−10
SWITCH	ε =1.0E−10	4402	4091	24.25	3.15E−10
	ε =1.0E−11	4331	4176	21.66	3.13E−10
	ε =1.0E−12	4709	4567	22.87	3.56E−10
wang3		total	double	sec.	‖b−Ax‖
DOUBLE		476	476	2.03	3.52E−10
QUAD		372		7.31	1.49E−10
SWITCH	ε =1.0E−10	460	361	3.67	1.59E−10
	ε =1.0E−11	459	444	2.42	9.22E−11
	ε =1.0E−12	479	476	2.32	1.46E−10
language		total	double	sec.	‖b−Ax‖
DOUBLE		39	39	3.42	2.96E−09
QUAD		36		10.53	4.25E−11
SWITCH	ε =1.0E−10	38	34	4.57	1.71E−10
	ε =1.0E−11	37	35	4.07	4.20E−10
	ε =1.0E−12	40	39	4.18	4.47E−10

– εはSWITCHのリスタート基準

- QUADとSWITCHは解の精度が最大2桁向上
- SWITCHは倍精度の2割増の時間でロバスト

まとめ

- SSE2を用いた高速な4倍精度演算を実装
  - 仮数部が8ビット少ない（double-double精度）
  - メモリ容量を節約
  - 5.71倍高速化
- 混合型反復解法を提案
  - 倍精度と高速な4倍精度の組み合わせ
  - SWITCHとPERIODIC
  - 高精度・ロバスト
  - 高速

まとめ：混合型反復法

- SWITCH：○
  - 倍精度で解けない問題が解ける
  - 倍精度の実行時間の2割増程度で最大2桁精度のよい解が得られる
- PERIODIC：△
  - 収束しないことが多い
  - 解の精度は倍精度と同じ

今後の課題

- 4倍精度をうまく活用して収束性を向上，安定して解けるロバストなライブラリの開発
  - SWITCHの最適なリスタート基準の決定法

Lisの取得

http://ssi.is.s.u-tokyo.ac.jp/lis/

lis ssi 検索

バージョン1．1．0β1を公開中
4倍精度演算（SSE2），混合型反復法利用可能