A Mixed Precision Iterative Method with Fast Quadruple Precision Arithmetic Operation tuned by SSE2

Hisashi Kotakemori (TCAD International, Inc.)
Hidehiko Hasegawa (U. of Tsukuba)

Motivation

- Krylov methods converge at most n iterations (n: dimension of Matrix) in exact computation.
- Diverge, stagnation, and many iterations occur because of round-off errors.
- High Accuracy computation is one choice, however it is very costly.
  - Real *16 of Fortran:
    Double of Memory
    20 times of Computation time.

Story

1) Fast Quadruple Arithmetic Operations
   - Not standards
   - double-double : use 2 double floating numbers
   - Accelerate computation with SSE2
   - Computation time: 3.2 times of Double

2) Mixed Precision Iterative method
   - Reduce Computation time with less Quadruple Arithmetic Operations
   - Restart with different precision
   - Computation time: 1.2 times of Double

Story (continued)

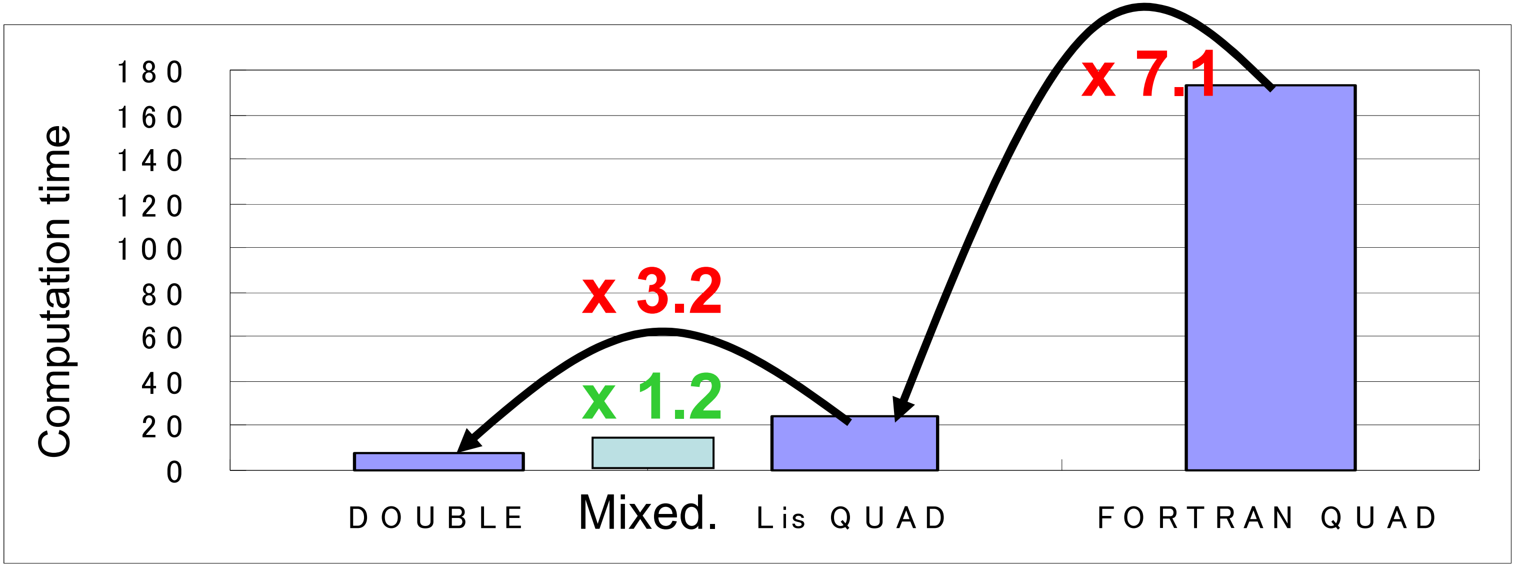
3) Auto restart strategy
   - Automatically changes precision

4) Parallelization
   - Almost same Data transfer
   - Much more computations (20 times)
   - Less round-off errors
   - → light preconditioner (easy to parallelize)

5) Provide Library: Lis and GUI: Lis-test

Computation time

Poisson（n=10^6, CRS）, Xeon 2.8GHz



Parallel	4	8	4	8
	***3.74**	***6.84**	***3.88**	***7.61**

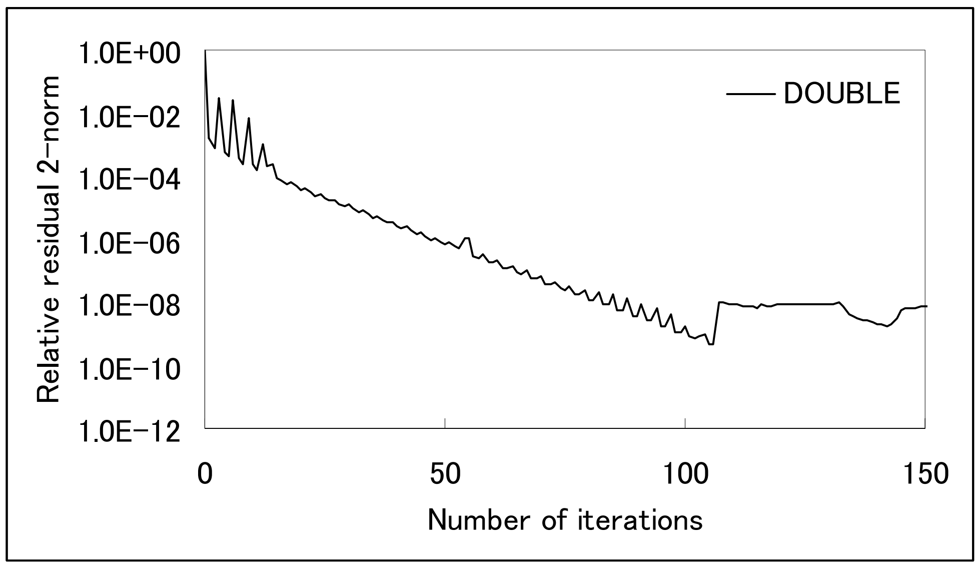
Implementation of Fast Quadruple Arithmetic Operations

Problem

- Theoretically Krylov methods converge at most n iterations
- Divergence and Stagnation happens because of round-off errors

$$
\begin{pmatrix}
2 & 1 & & & & \\
0 & 2 & 1 & & & \\
1.4 & 0 & 2 & 1 & & \\
 & \ddots & \ddots & \ddots & \ddots & \\
 & & 1.4 & 0 & 2 & 1 \\
 & & & 1.4 & 0 & 2
\end{pmatrix}
$$

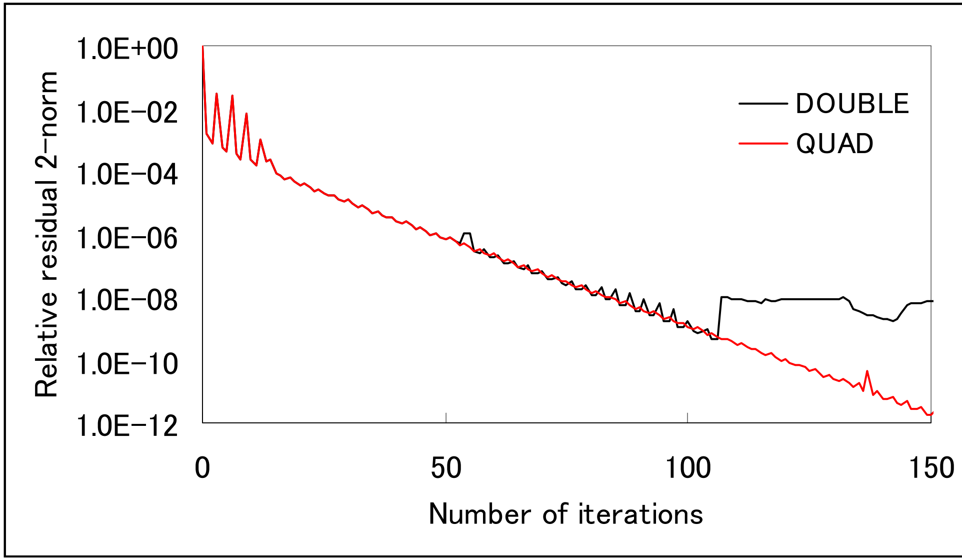
n：10^5



Answer

- To improve convergence, High-precision arithmetic operation is effective.
- However it is costly. real *16 of Fortran:
  - memory: Double
  - comp. 20 times

Fast Quadruple is necessary



Round-off error free Double Arithmetic Addition

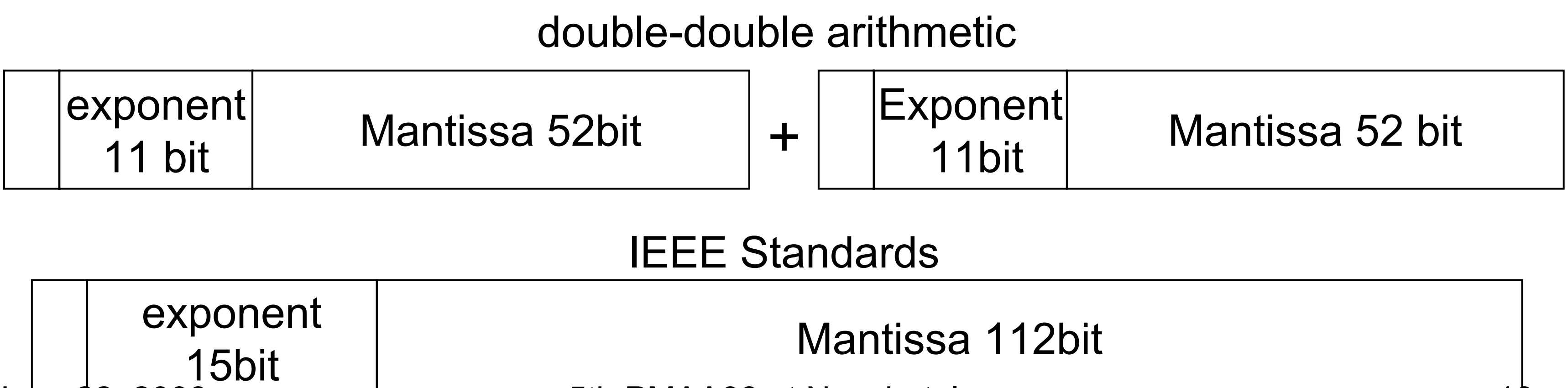
- Round-off error free addition can be done with two double precision variables:

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

  - a,b : double floating point variables
  - $fl(a + b)$: addition of a and b in double
  - $err(a+b)$: $(a+b) - fl(a+b)$: error

double-double arithmetic

- Quadruple value is stored in two double floating point numbers
  - double-double arithmetic: a = a.hi + a.lo, |a.hi|>|a.lo|
  - 8 bit less than IEEE standards
  - Effective digits is approx. 31 vs. 33 digits.

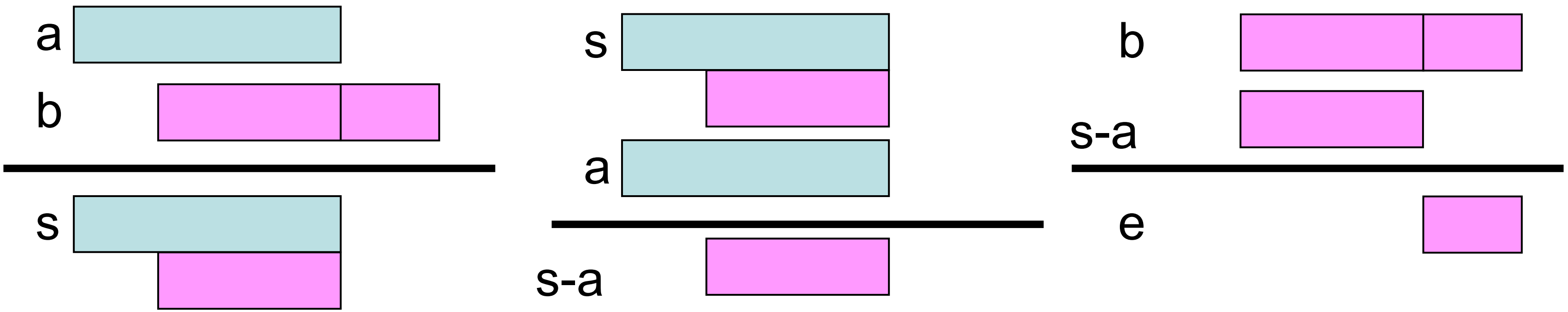


Basic Algorithm

- Dekker showed round-off error free addition in double

|a|>=|b| 3flops. Others 6flops.

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



Quadruple Addition of a=b+c

b.hi b.lo + c.hi c.lo

A. TWO_SUM for upper parts

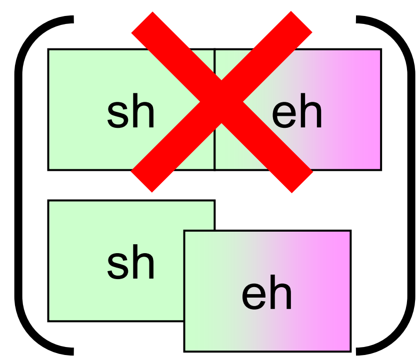
b.hi + c.hi = sh eh

B. Addition of lower parts

b.lo + c.lo = b.lo+c.lo

C. Addition of result and error of upper part

b.lo+c.lo + eh = eh



D. FAST_TWO_SUM of results A and C

sh + eh = a.hi a.lo

Quadruple Addition of a=b+c

20 flops

```
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)

Design of Fast Quad. operations for Lis (**a Library of Iterative Solvers for linear systems**)

- Same API with Double
- Double: Input（A, b, x_0 ）
- Double: Output
- Double: Creation of Preconditioner
- Fast Quad.: iterative solution x,
  All working variables
- Fast Quad.: Applying Preconditioner Mu=v

Implementation

- Replace Double to Fast Quadruple Arith. Op.
  - Matrix-vector product(`matvec`)
  - Inner product(`dot`)
  - Vector operations(`axpy`)
- Use multiply-and-Add for matvec,dot,axpy
  - Reduce memory access, especially store
- Make two Multiply-and-Add functions
  - FMA：Operation with Fast Quadruple variables
    - Used for Dot and axpy
  - FMAD： Operation with Double and Fast Quadruple
    - Used for matvec

Implementation FMA･FMAD

- FMA $a = a + b \times c$

33 flops

```
FMA(a,b,c) {
  TWO_PROD(b.hi,c.hi,p1,p2)
  p2 = p2 + (b.hi * c.lo)
  p2 = p2 + (b.lo * c.hi)
  FAST_TWO_SUM(p1,p2,p1,p2)
  TWO_SUM(a.hi,p1,sh,eh)
  TWO_SUM(a.lo,p2,sl,el)
  eh = eh + sl
  FAST_TWO_SUM(sh,eh,sh,eh)
  eh = eh + el
  FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```

- FMAD $a = a + b \times c$

29 flops

```
FMAD(a,b,c) {
  TWO_PROD(b,c.hi,p1,p2)
  p2 = p2 + (b * c.lo)

  FAST_TWO_SUM(p1,p2,p1,p2)
  TWO_SUM(a.hi,p1,sh,eh)
  TWO_SUM(a.lo,p2,sl,el)
  eh = eh + sl
  FAST_TWO_SUM(sh,eh,sh,eh)
  eh = eh + el
  FAST_TWO_SUM(sh,eh,a.hi,a.lo)
}
```

Acceleration by SSE2

- SSE2 Used for vectors（dot, axpy, matvec）
  - 2 Multiply-and-add in same time
- Two FMA in a loop with loop unrolling
  - pd instruction of SSE2 can be used to all
- Code tuning
  - Alignment
  - Some Hand optimization

Speed test

- Machine

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

- Compiler options

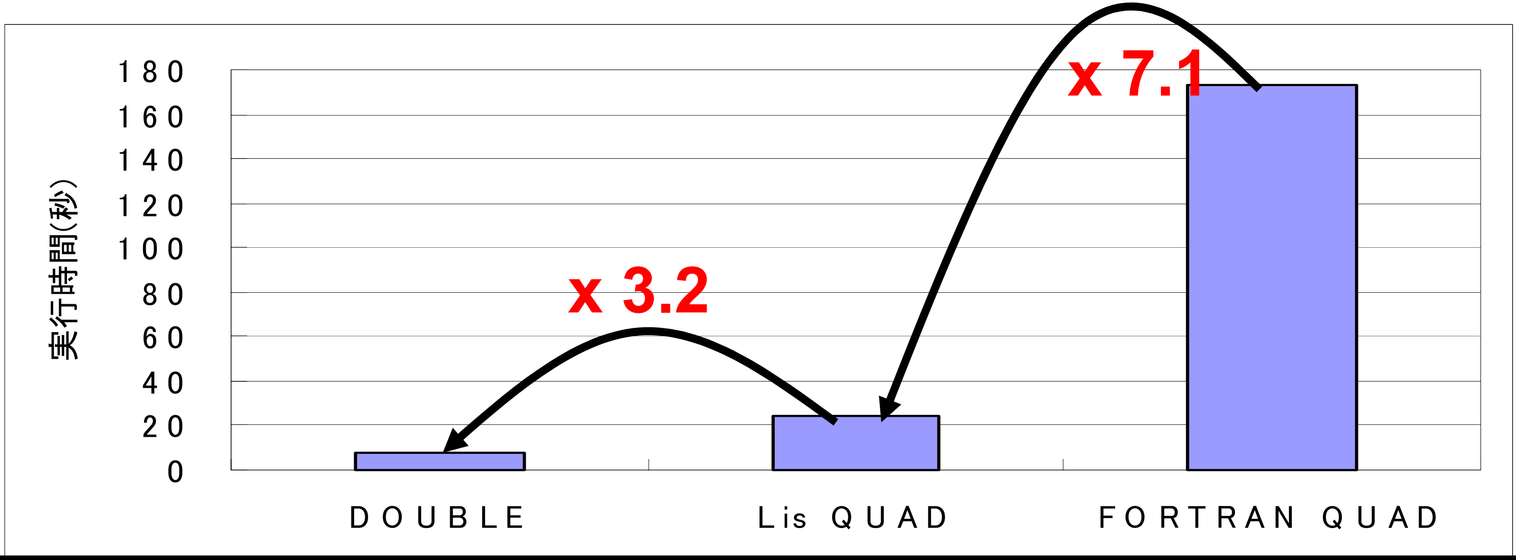
 - Optimization is -O3

 - -mp option used for source file of FMA

 - -xW used for auto-vectorization

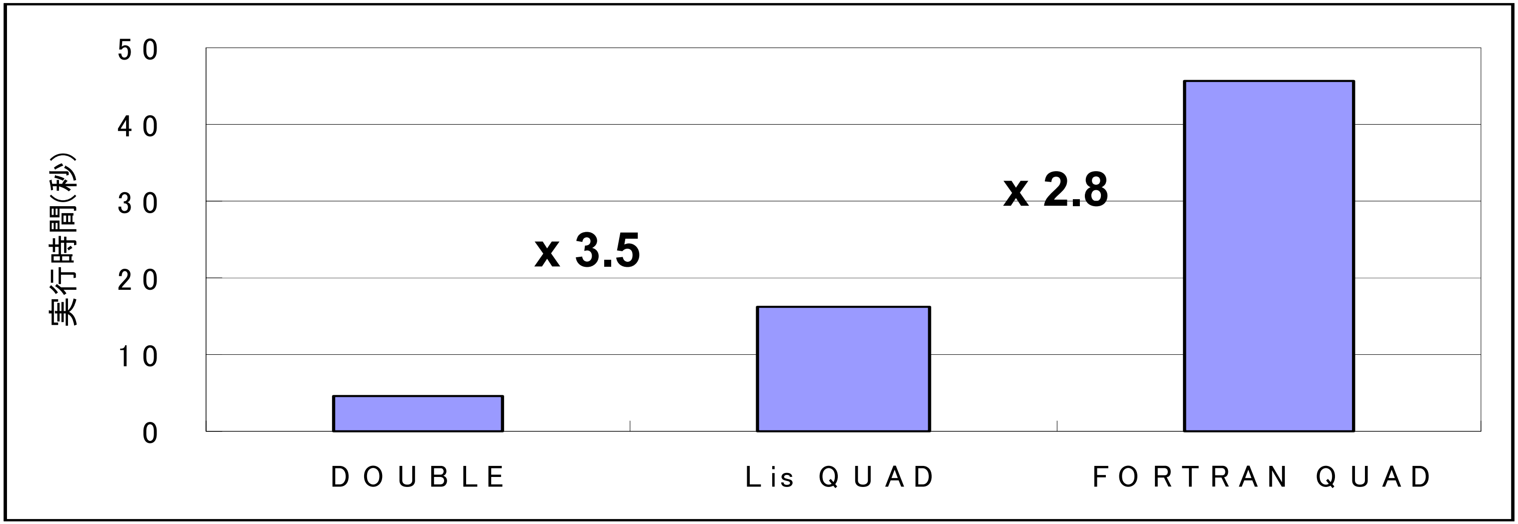
Time for 50 BiCG iterations

Poisson（n=10^6, CRS）, Xeon 2.8GHz



	DOUBLE	Lis QUAD	FORTRAN QUAD
Matrix A(CRS)	4(n+nnz)+8nnz	4(n+nnz)+8nnz	4(n+nnz)+16nnz
Vector b	8n	8n	16n
Vector x	8n	16n	16n
Workings	6*8n	6*16n	6*16n
sum	121.9MB	175.8MB	221.6MB

Time for 50 BiCG iterations
Poisson（n=10^6, CRS）, Core2 Duo 2.4GHz



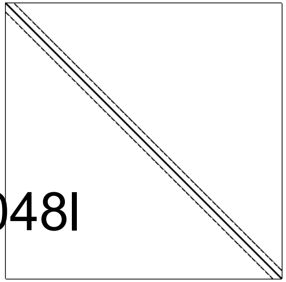
- Double : Fortran Quad is 1 : 9.9.
  - Intel C/C++ 9.1, Intel Fortran 9.1, -O3 -xW

Confirming Convergence

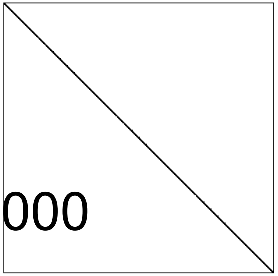
- BiCG
- Right hand size b
  - Solution $x = (1,\ldots,1)^T$
- Initial solution $x_0 = (0,\ldots,0)^T$
- Convergence criteria $\|r_{k+1}\|_2 / \|r_0\|_2 \leq 10^{-12}$
- Max iterations = 10^4

Test Problems

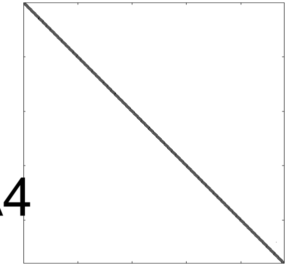
- Poisson
  - 2 dimension, FDM
  - N=10^6, nnz=5x10^6
- rdb2048l (Chemical engineering)
  - MatrixMarket, n=2048, nnz=12032, cond = 1.8x10^3
- olm1000 (Hydrodynamics)
  - MatrixMarket, n=1000, nnz=3996, cond = 3x10^6
- A4 (Electronic potential)
  - n=23,994, nnz=214,060
- **Cryg10000 (CRYSTAL GROWTH EIGENMODES)**
  - UF Sparse Matrix Collection, n=10000, nnz=49699
- circuit_3 (Circuit Simulation)
  - n=12,127, nnz=48,137



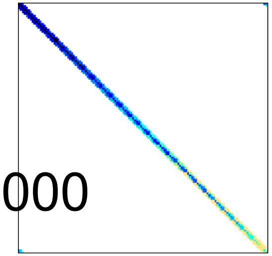
rdb2048l



olm1000



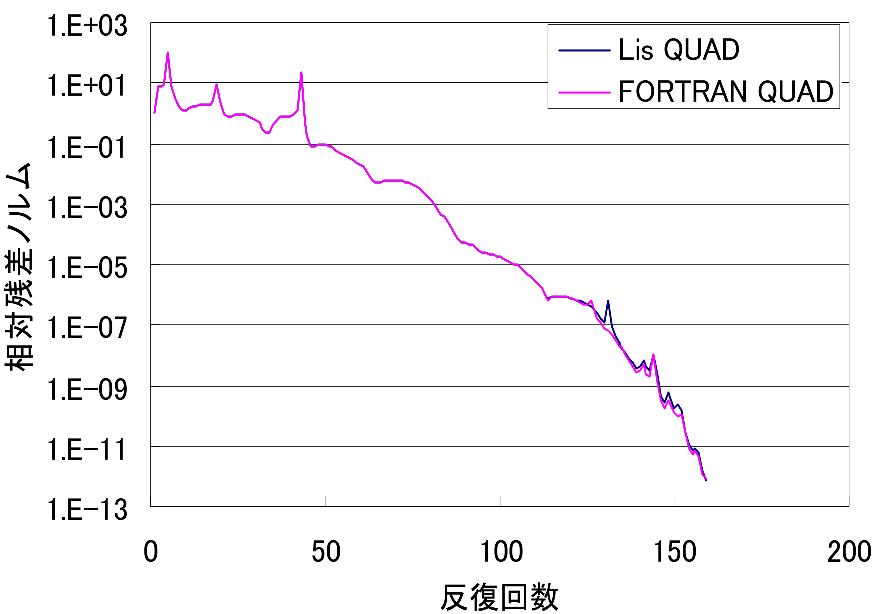
A4



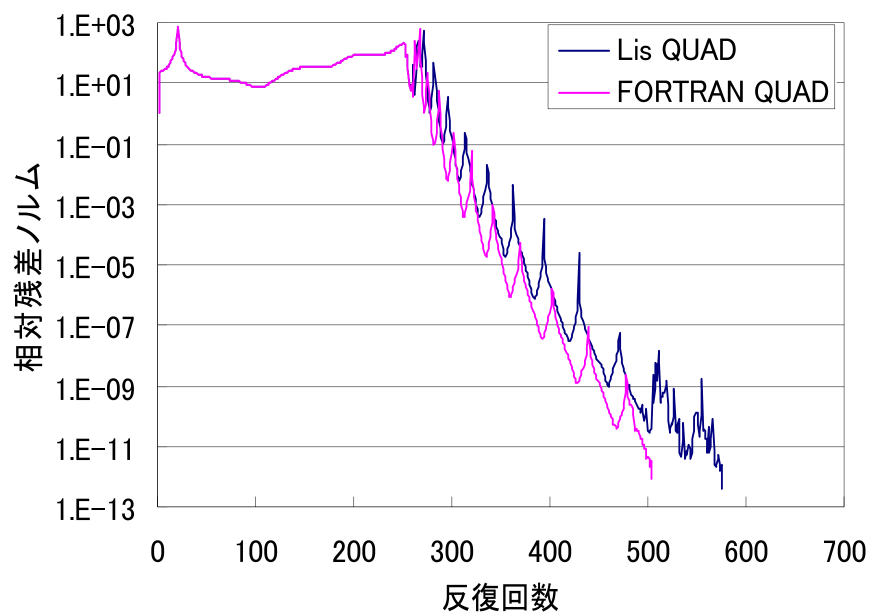
cryg10000

Comparison of Real *16 vs. Fast Quadruple with BiCG

rdb2048l (n=2048,cond=1.8E+3)

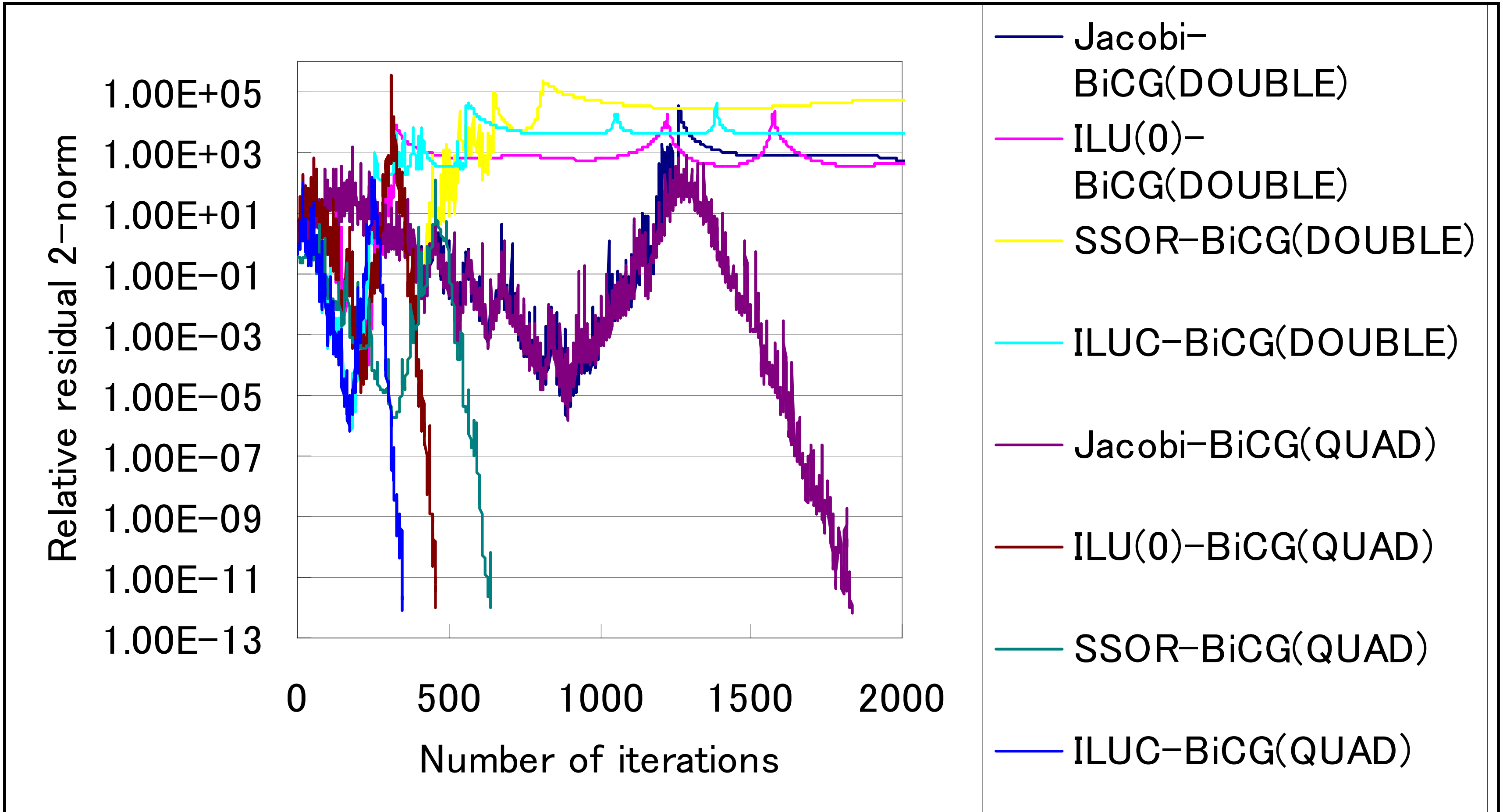


olm1000 （n=1000,cond=3.0E+6）



- same accuracy = same number of iterations (Difference is at most 10%)!

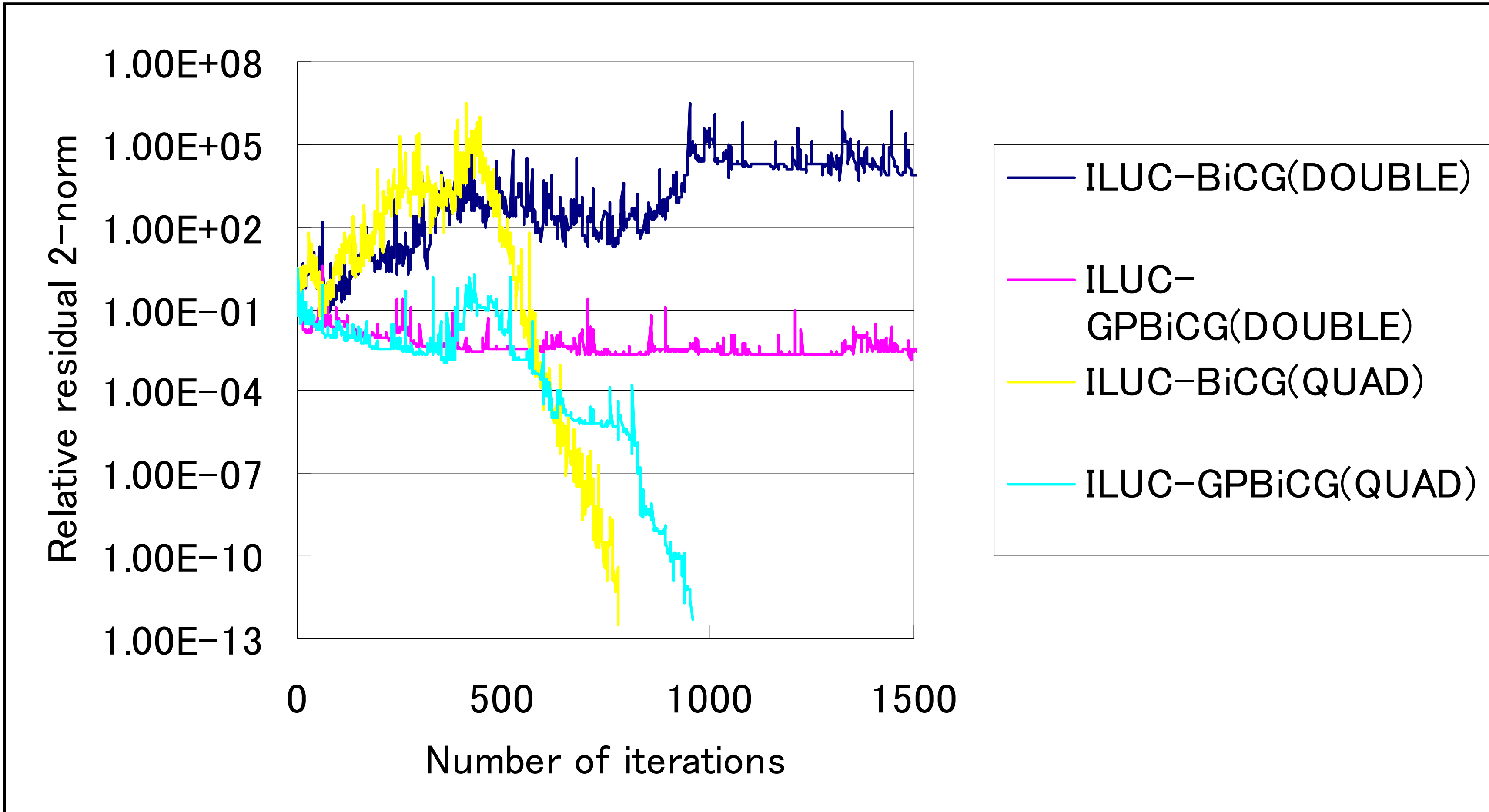
Convergence History of A4 with Preconditioned BiCG



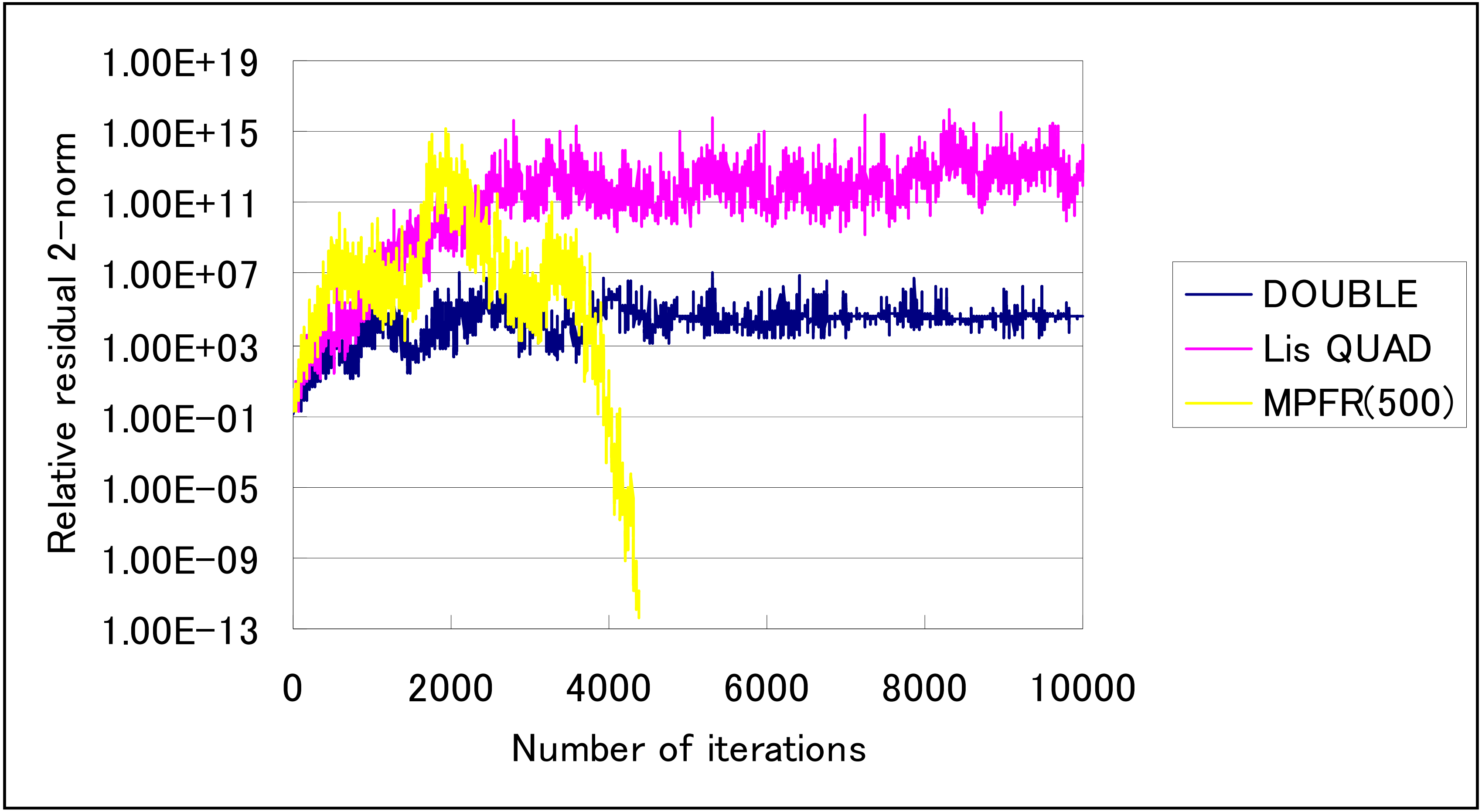
Result of Problem A4

Pre.	Double			Fast Quadruple		
	sec.	iter.	TRR	sec.	iter.	TRR
BiCG						
Jacobi				26.58	1833	7.68E-15
ILU(0)				20.41	460	1.25E-14
SSOR				29.78	642	1.27E-14
ILUC				17.78	350	1.13E-14
GPBiCG						
Jacobi				34.50	1403	6.89E-15
ILU(0)	2.99	407	1.91E-14	18.43	225	1.17E-14
SSOR				42.53	500	1.02E-14
ILUC	11.71	364	1.67E-14	25.95	274	3.05E-15

cryg10000 BiCG with ILU(0)



cryg10000 BiCG with Jacobi



Mixed Precision Iterative Methods

Combination of Double and Fast Quadruple

Mixed Precision Iterative methods

Lis QUAD: Fast Quadruple Arithmetic

- Improve convergence! Make robust?!
- Over quality
- Still Costly
  x 3.2 on Xeon, x 3.1 Core2 Duo

Reduce computation time

→ Reduce Quadruple Operations

Basic idea of restart

- Until Now:

$$(1)\text{Solve } Ax^{*}=b \ \text{ with some initial value } x_0$$

$$(2)\text{Solve } Ax=b \ \text{ with an initial value } x^{*}$$

  - In general, (1) and (2) have same spaces, same methods, and same precisions
  - (1) and (2) have same spaces, same methods but **different precision (combination of Double and Fast Quadruple)**.

Mixed Precision Iterative method with combination of double and Fast Quadruple

- Reduction of Fast Quadruple operations
  - Reduction of Computation time

- Two Methods
  - SWITCH Algorithm based on “restart”

    D → Q

    Q → D
  - PERIODIC Algorithm based on “correction”

SWITCH algorithm (D → Q, Q → D)

- Restart with different precision arithmetic
  - Current solution x_k is passed at the restart
  - Upper and Lower part of Double-Double var. are stored in different arrays
  - Only Upper part is used for Double Precision
  - Two Stages are performed by Different Precision

```
for(k=0;k<matitr;k++){
    The first step
    if( nrm2<restart_tol ) break;
}
Clear all values except x
for(k=k+1;k<maxtr;k++) {
    The second step
    if( nrm2<tol ) break;
}
```

PERIODIC algorithm

- A Fast Quadruple is used each k iterations
  - All values are passed at the change
  - No cost at the change of Q → D
  - Lower part is cleared at the change of D → Q

```
for(k=0;k<maxitr;k++) {
   if( k%interval<num ) {
     Fast Quadruple is used
   } else {
     Lower part is cleared
     Double is used
   }
```

Teoplitz γ=1.3, n=10^5

		iter. total	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

– Epsilon is restart criterion of DQ-SWITCH

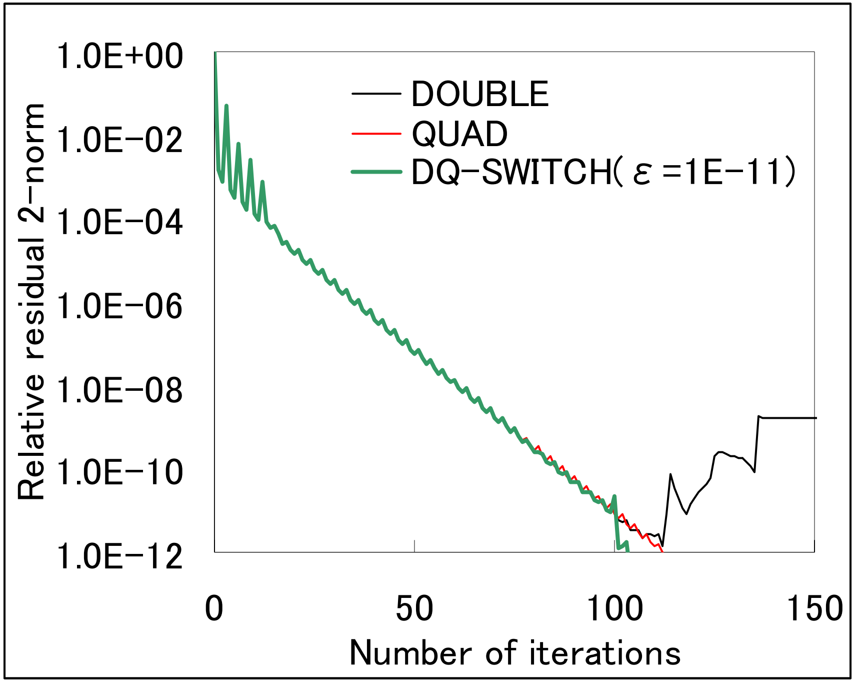
– Num: Quad. Ops. Used num times per 10 iterations

Teoplitz γ=1.4, n=10^5

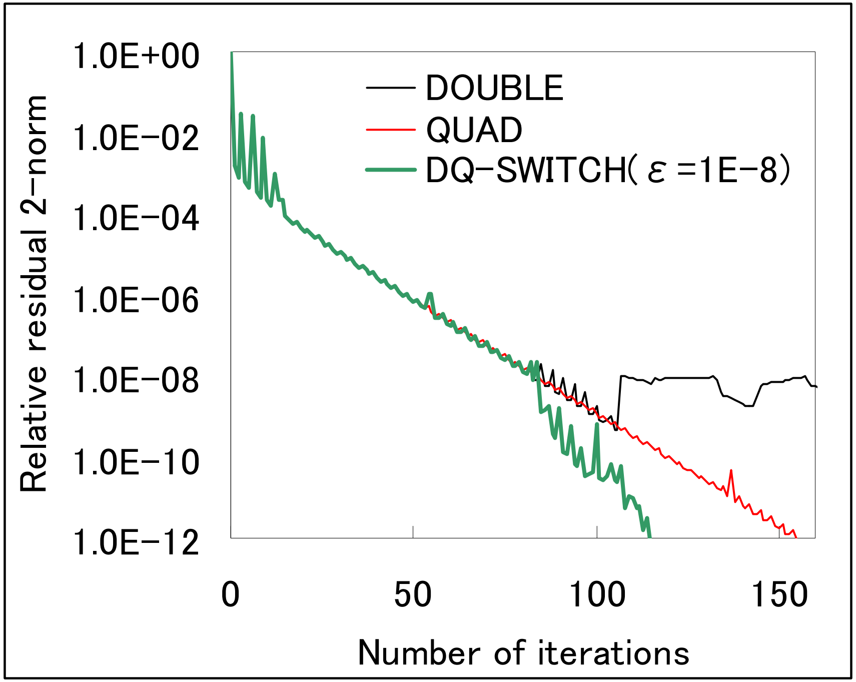
		iter. total	double	sec.	||b−Ax||
FMA2_SSE2		155	0	8.77	2.85E−10
SWITCH	ε =1.0E−07	119	65	4.04	2.51E−10
	ε =1.0E−08	115	83	3.06	3.04E−10
	ε =1.0E−09	−			
PERIODIC	num=1	−	−		
	num=2	−	−		
	num=3	−	−		
	num=4	−	−		
	num=5	−	−		
	num=6	−	−		

Convergence History

$\gamma = 1.3$



$\gamma = 1.4$



- DQ-SWITCH is good convergence

Epsilon dependency

	ε	γ=1.3 iter. total	double	quad	sec	γ=1.4 iter. total	double	quad	sec
QUAD		113			2.88	155			3.94
DQ-SWITCH	1.00E-03	114	2	112	2.87	156	2	154	3.94
	1.00E-04	109	11	98	2.59	152	15	137	3.62
	1.00E-05	105	23	82	2.26	146	31	115	3.16
	1.00E-06	104	35	69	2.01	138	47	91	2.67
	1.00E-07	95	47	48	1.58	123	65	58	1.96
	1.00E-08	94	61	33	1.29	119	83	36	1.53
	1.00E-09	95	74	21	1.08	--			
	1.00E-10	95	86	9	0.86	--			
	1.00E-11	103	98	5	0.84	--			

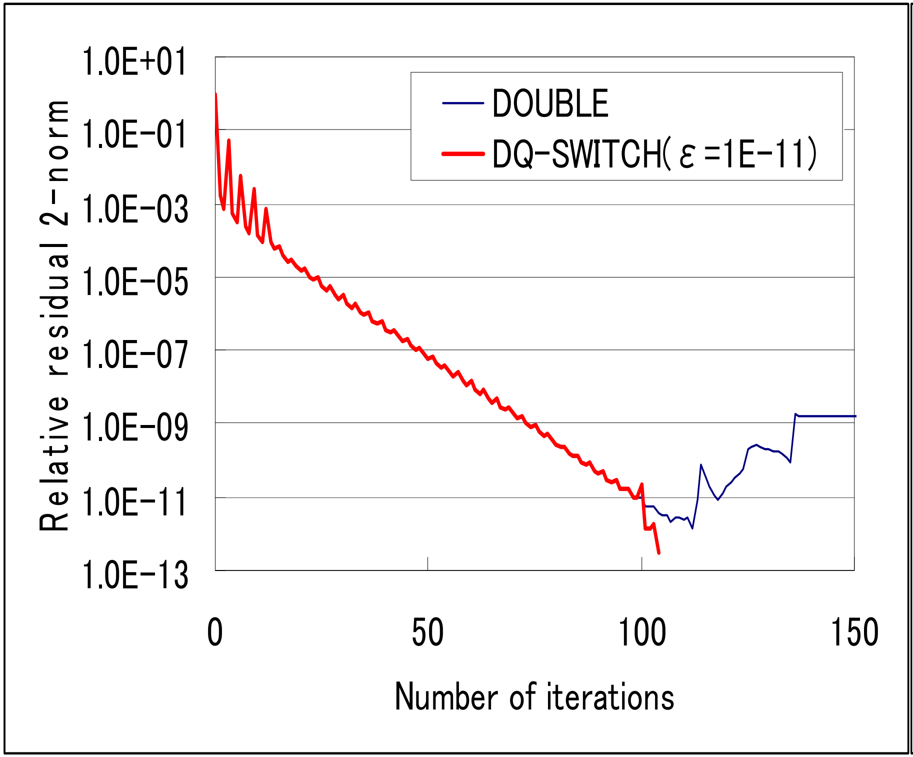
- Choice of appropriate epsilon is important
- Small epsilon reduces much computation time
- Smaller epsilon makes divergence

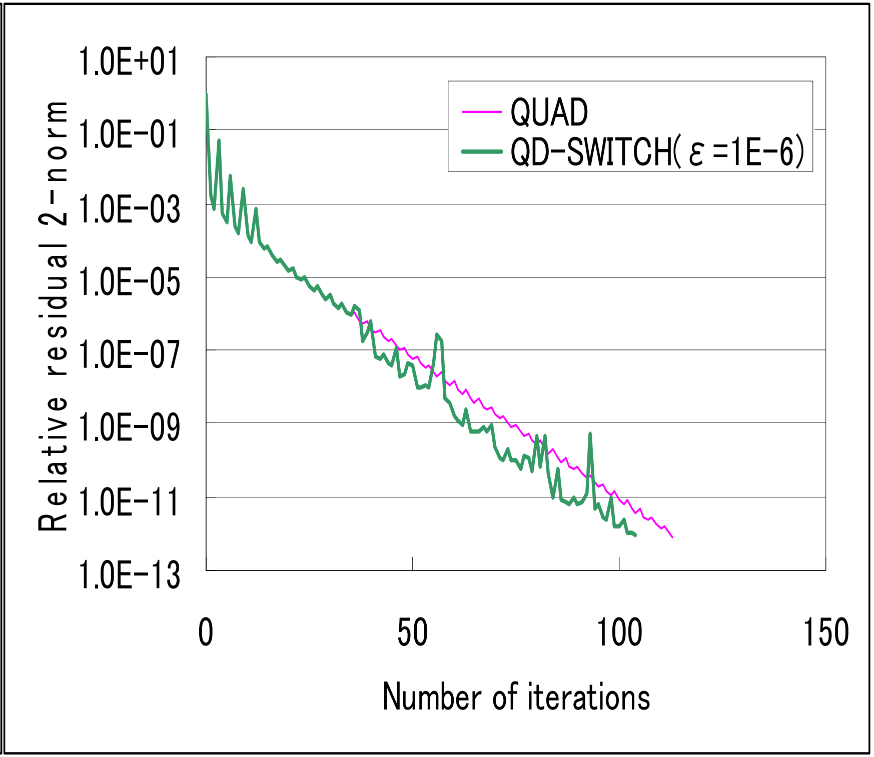
D → Q vs. Q → D Switch, BiCG

	ε	γ=1.3 iter. total	double	quad	sec.
QUAD		113			2.88
DQ-SWITCH	1.0E-05	105	23	82	2.26
	1.0E-06	104	35	69	2.01
	1.0E-07	95	47	48	1.58
	1.0E-08	94	61	33	1.29
	1.0E-09	95	74	21	1.08
	1.0E-10	95	86	9	0.86
	1.0E-11	103	98	5	0.84
QD-SWITCH	1.0E-05			23	
	1.0E-06	104	69	35	1.40
	1.0E-07	95	48	47	1.56
	1.0E-08	100	39	61	1.85
	1.0E-09	95	21	74	2.05
	1.0E-10	95	9	86	2.27
	1.0E-11	103	3	100	2.58

- Choice of Appropriate epsilon is necessary
- DQ-SWITCH has wide convergent area

Convergence History





- Decision of restart in QD-SWITCH is difficult.

Comparison of Double and DQ-SWITCH

- University of Florida Sparse Matrix Collection

Matrix	dimension	nnz	Size of memory	
			Double	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

airfoil_2d		iter. total	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

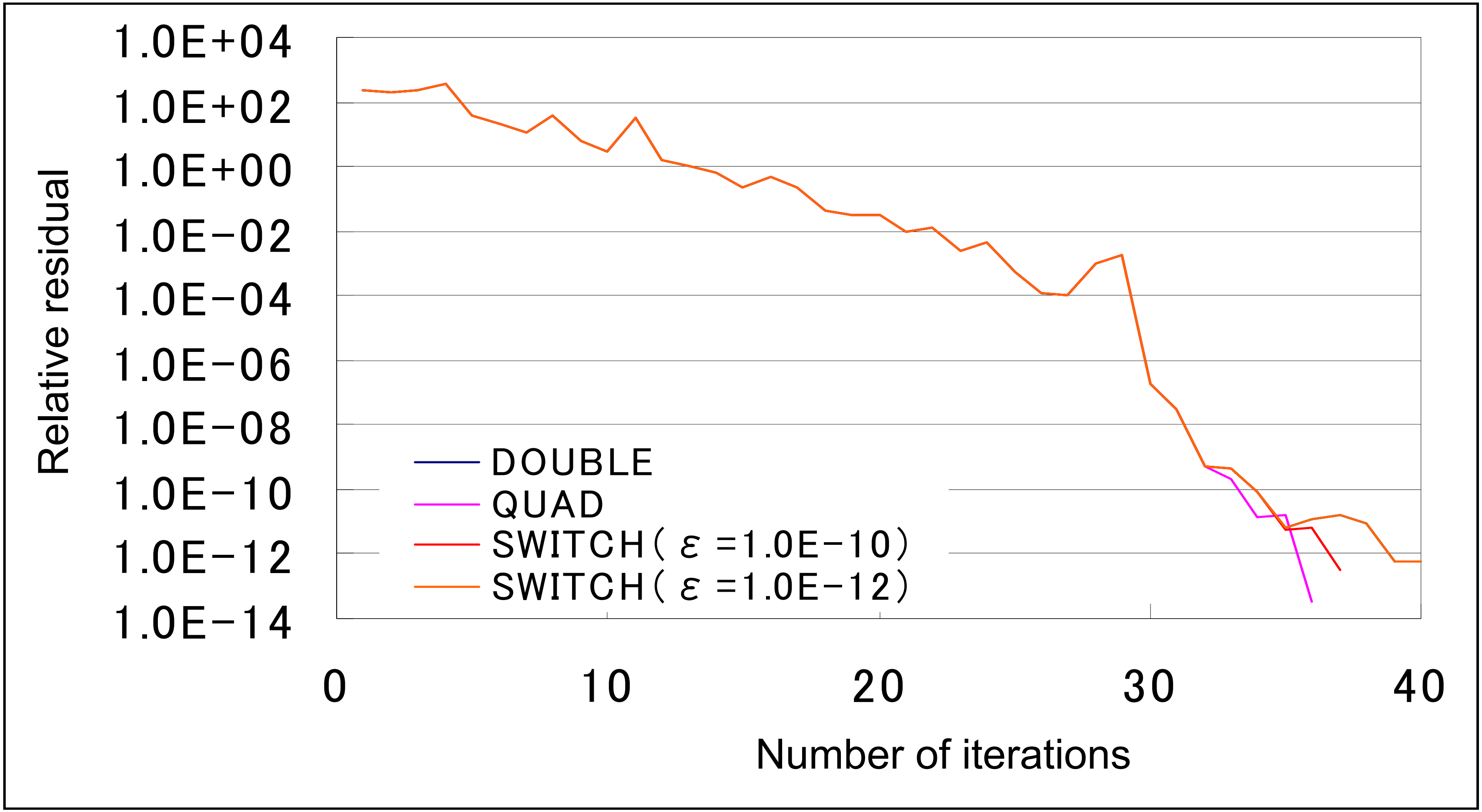
wang3		total	double	sec.	||b−Ax||
DOUBLE		476	476	2.03	3.52E−10
QUAD		372		7.31	1.49E−10
SWITCH	ε =1.0E−10	460	361	3.67	1.59E−10
	ε =1.0E−11	459	444	2.42	9.22E−11
	ε =1.0E−12	479	476	2.32	1.46E−10

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 is restaring criterion of SWITCH

- QUAD and SWITCH improve 2 digits for solution’ quality
- SWITCH is 20% overhead on the double, however robust

DQ-SWITCH for language, BiCG



For BiCG with ILU(0)

Toeplitz $\gamma=1.4$

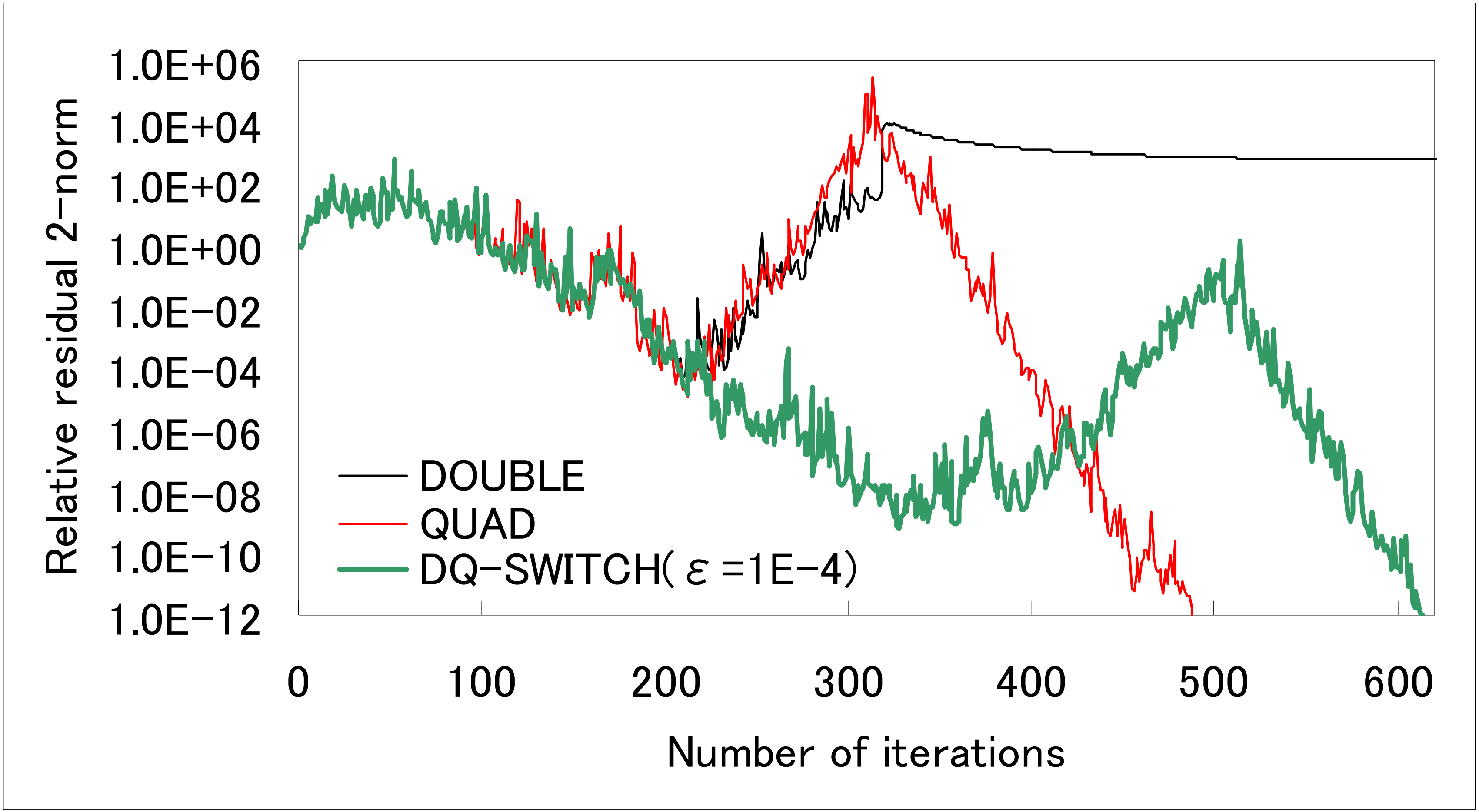
	switch_tol	precon.	iter. total	iter. double	sec.	||b−Ax||
DOUBLE		ILU(0)				
QUAD		ILU(0)				
SWITCH	1.00E−06	ILU(0)	51	23	1.45767	1.39E−10
	1.00E−07	ILU(0)	49	29	1.17805	2.41E−10
	1.00E−08	ILU(0)	51	39	0.94101	2.16E−10

language

	switch_tol	precon.	iter. total	iter. double	sec.	||b−Ax||
DOUBLE		ILU(0)	9	9	0.67325	6.53E−10
QUAD		ILU(0)	9		1.82325	4.27E−11
SWITCH	1.00E−12	ILU(0)	10	9	0.92794	3.14E−11

A4: electronics effect

Circuit_3, BiCG with ILU(0)



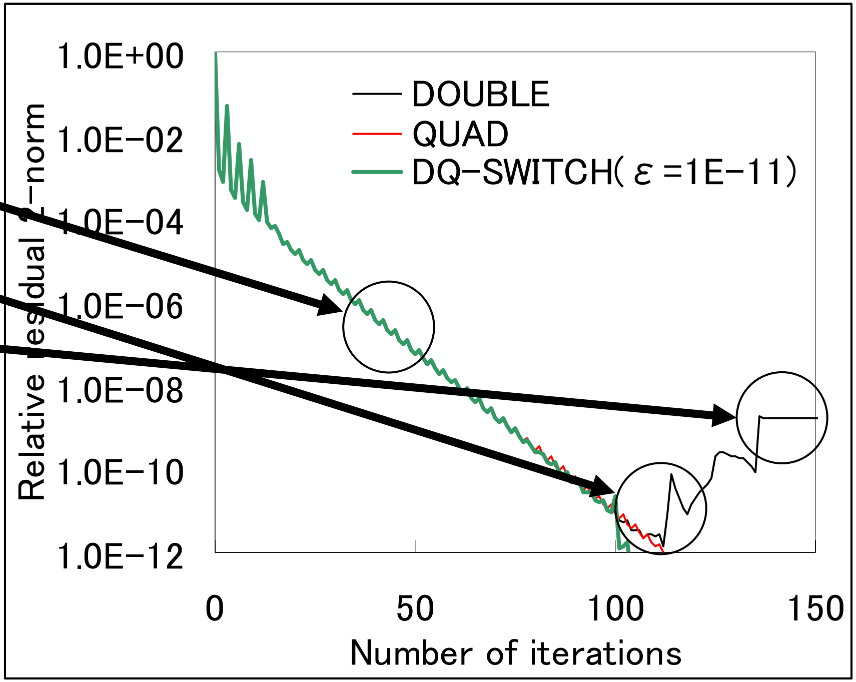
Mixed Precision Iterative Methods

- Difficult problems are solved with Mixed or QUAD.
- Overhead of the mixed precision iterative methods is 20%
- PERIODIC is NO Good
  - Not effective
- SWITCH is Good
  - Maximum speed-up is 3.9 times compared Lis QUAD.
  - Small epsilon makes more speed up, but dangerous
- DQ-SWITH is the best in these

Auto restart

For Auto restart with Different precisions

- Convergent history shows three patterns:
  - (C)Converge
  - (D)Diverge
  - (S)Stagnate



- To detect (D) and (S)
  - Then, restart at the point

Auto restart of DQ-SWITCH

- Compute deviation of residual norm

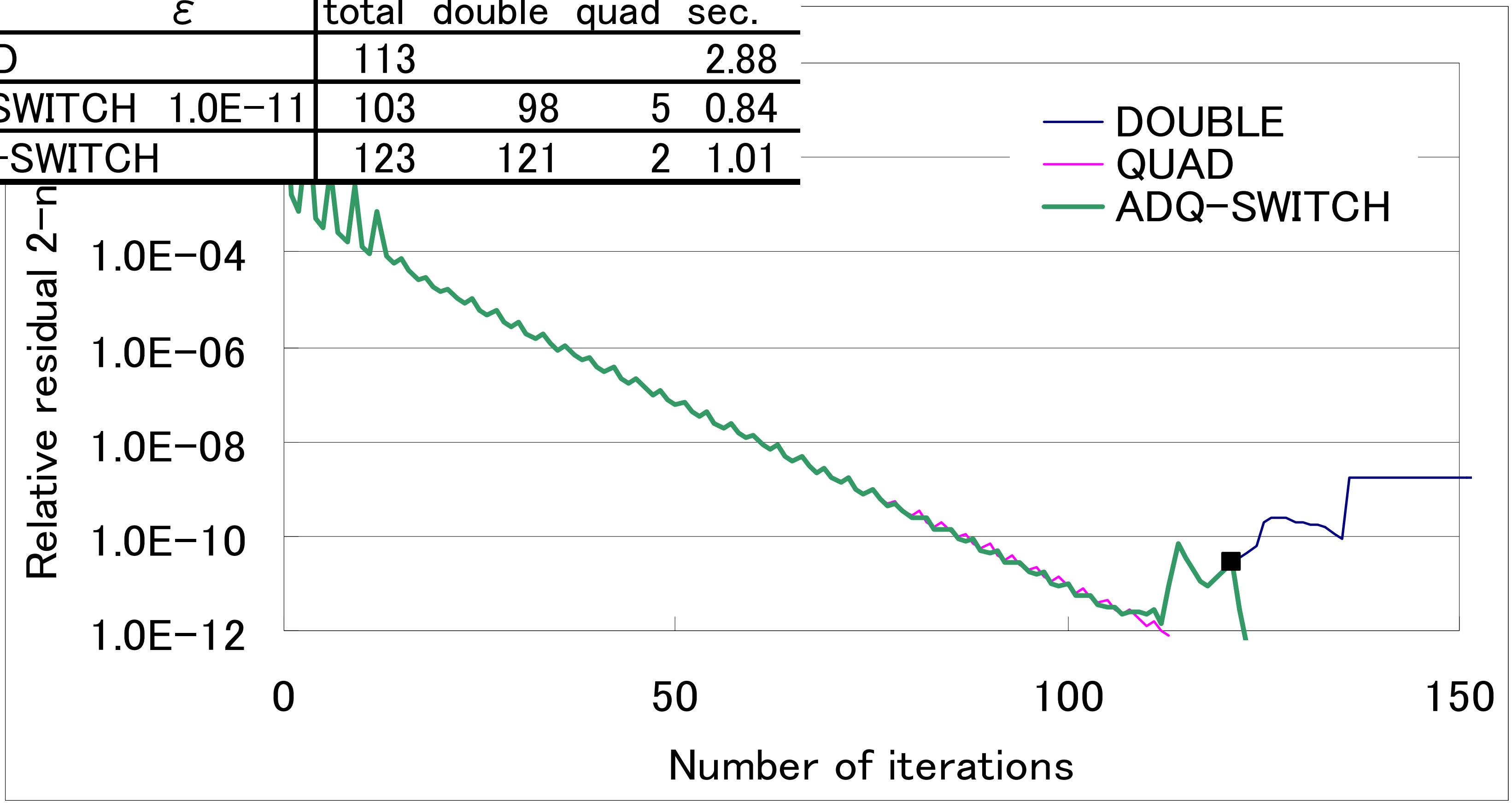
$$v = \frac{1}{p}\sum_{i=1}^{p}\left(\frac{nrm(i) - nrm(1)}{nrm(1)}\right)^2$$

- (D) $v \geq 10^2$
- (S) $v \leq 10^{-1}$

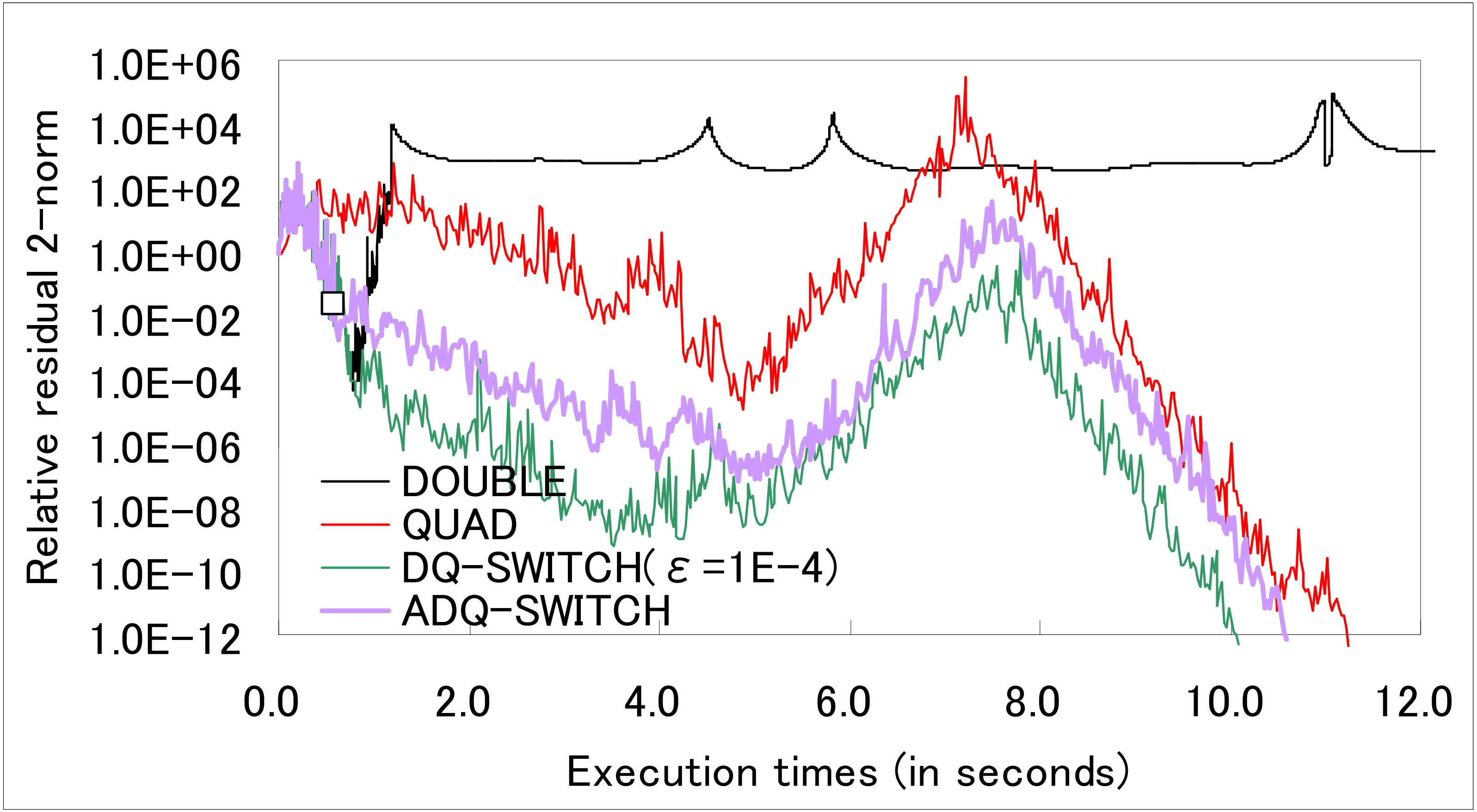
```
if( nrm2 < nrm2_min )
     nrm2_min = nrm2;   x_bak = x;
nrm_bak[k%10] = nrm2;
if( k>=10 ) {
   v = 0.0;    c = 0;
   for(i=0;i<10;i++) {
     t = nrm_bak[i] - nrm_bak[(k-9)%10];
     t = t / nrm_bak[(k-9)%10];
     v = v + t*t;
     if( nrm_bak[(k-9)%10] <= nrm_bak[i] )
       c = c+1;
   }
   v = v / 10;
   if( v<=0.1 || (c==10 && v>=100) ) break;
   if( nrm2<tol ) break;
}
```

Toeplitz(1.3)

		γ=1.3			
		iter.			
	ε	total	double	quad	sec.
QUAD		113			2.88
DQ-SWITCH	1.0E-11	103	98	5	0.84
ADS-SWITCH		123	121	2	1.01



Electoronics BiCG with ILU(0)



- Divergence and Stagnation are detected.
- Computation time is reduced.

Summary：Mixed Precision

- PERIODIC： NG
- SWITCH： Good at least 2 digits with 20% more
  - Q → D: timing of restart is difficult
  - D → Q: easy, robust, however depends on timing of restart
- Auto restart of DQ-SWITH
  - Deviation is used for detecting “Diverge” and “Stagnate”
  - Almost fine for these test problems

Parallel Issues

Parallel Issues for Fast Quad.

- Depends on the implementation of Ax, A^Tx, $M^{-1}x$, $M^{-T}x$, and Matrix Storage Format
- Data transfer is almost same
- Heavy Computation

→ Suitable for Distributed Parallel

- Less round-off errors

→ light preconditioner (easy to parallelize)

50 BiCG iterations on Distributed Parallel

# of PEs	Double	Fast Quad
1	7.56sec	24.21sec
2	3.90sec(1.93)	12.22sec(1.98)
4	2.02sec(3.74)	6.23sec(3.88)
8	1.11sec(6.87)	3.18sec(7.61)

Parallel Computing

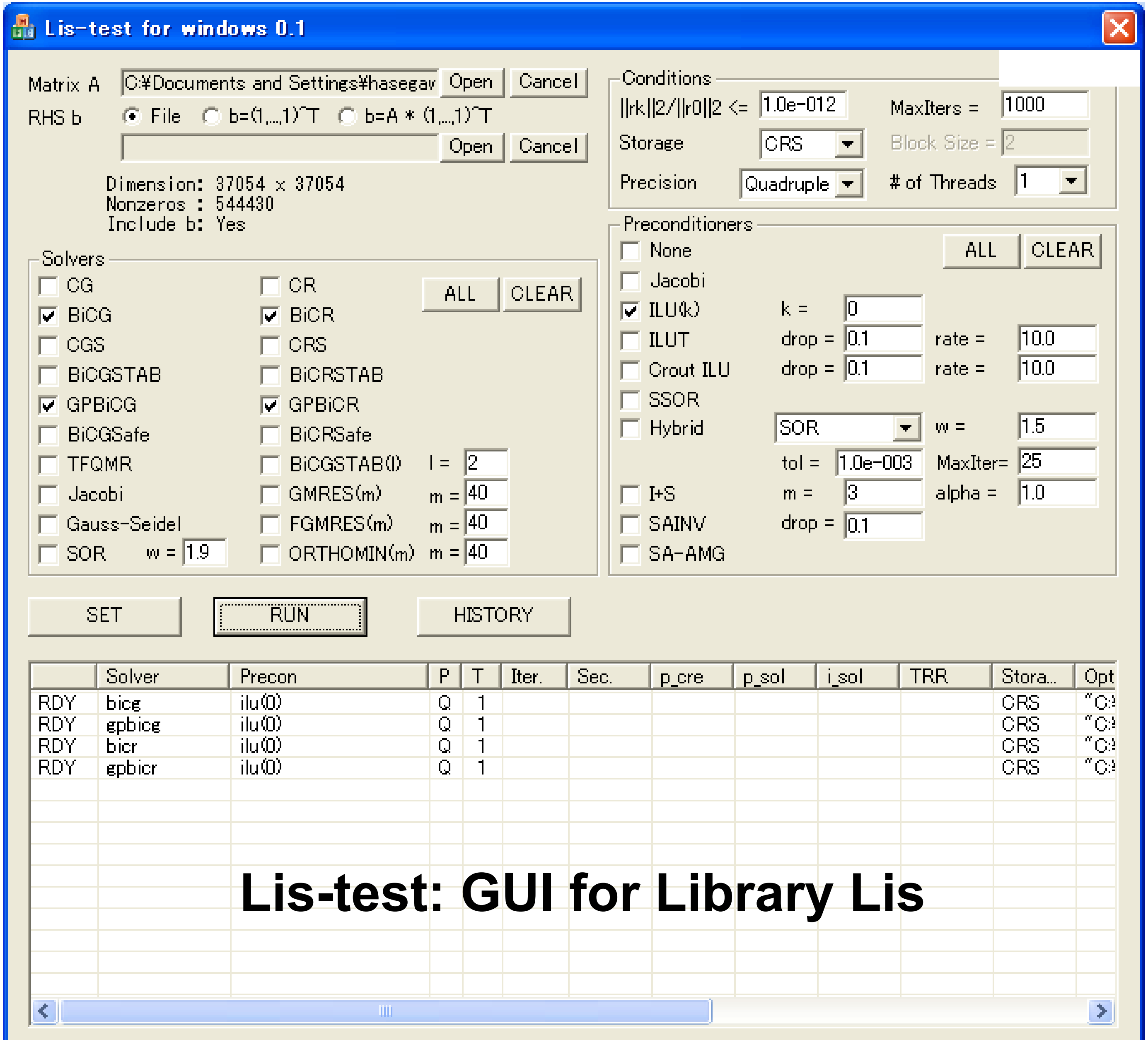
- Performance depends on problems, implementations, and computing environments

- Sorry!
  Currently No results for “real” problems

Free Library Lis and GUI Lis-test

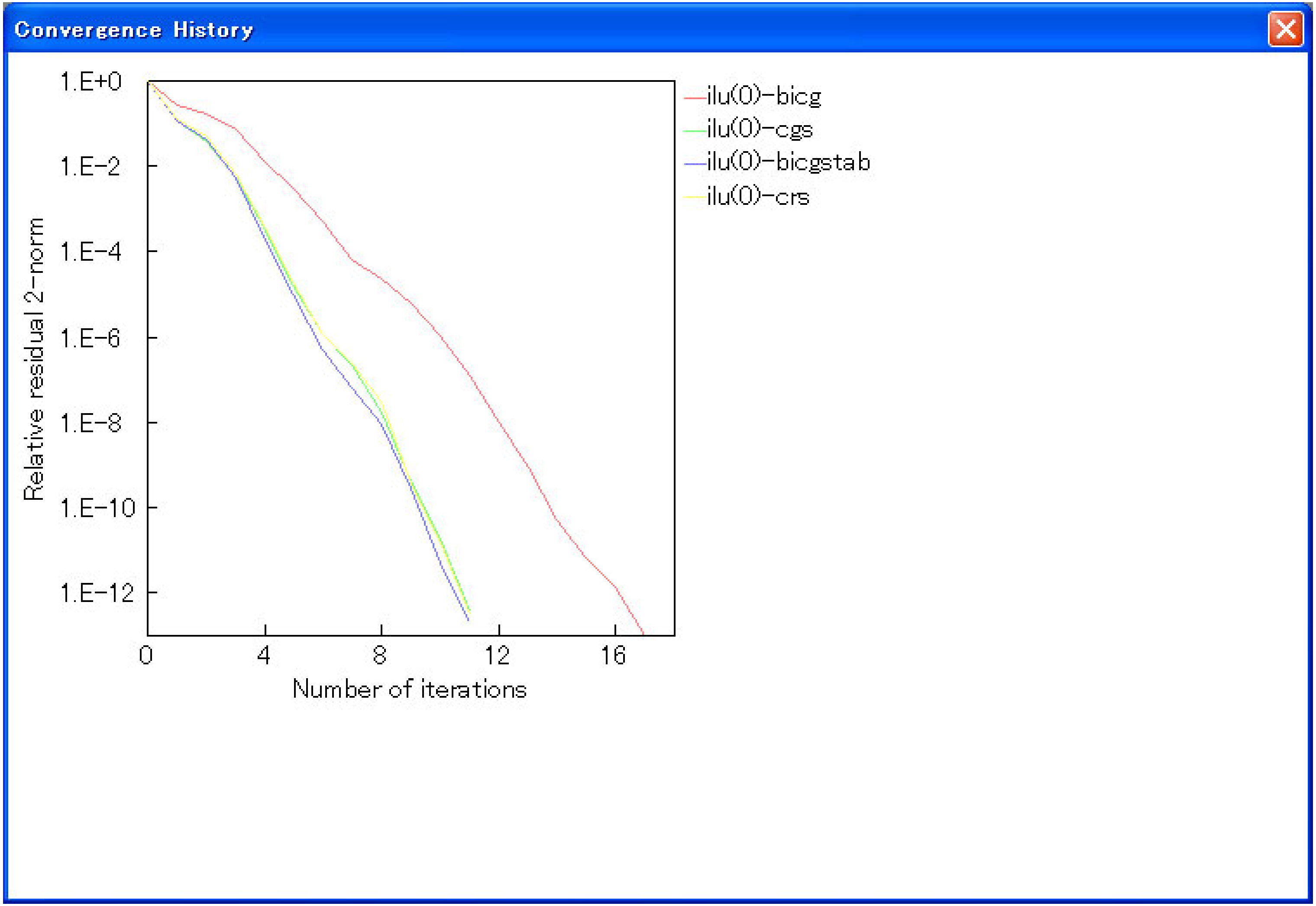
Lis-test for evaluation

- Consists of two Windows files
- Not necessary to install. Run from USB
- Prepare Matrix data as text file with Matrix Market' exchange format
- Run in parallel if the PC is multi-core
- To click, solutions, history, etc are computed



Lis-test: GUI for Library Lis

Comparison is done easily!



To get codes Lis-test/Lis

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

lis ssi 検索

version 1.1.0 β 2

Lis has more than 10*13*11 combinations

Precond.
Jacobi
SSOR
ILU(k)
Hybrid
I+S
SAINV
SA-AMG
Crout ILU
additive schwarz
User defined

Solvers
CG
BiCG
CGS
BiCGSTAB
BiCGSTAB(l)
GPBiCG
BiCGSafe
Orthomin(m)
GMRES(m)
TFQMR
Jacobi
Gauss-Seidel
SOR

Storage Format
CRS: Compressed Row
CCS: Compressed Column
MSR: Modified Compressed Sparse Row
DIA: Diagonal
ELL: Ellpack-Itpack gen. diag.
JDS: Jagged Diagonal
COO: Coordinate
DNS: Dense
BSR: Block Sparse Row
BSC: Block Sparse Column
VBR: Variable Block Row

Other features

- 4 computing environments
  - Serial
  - OpenMP for Shared memory
  - MPI for Distributed memory
  - Hybrid of OpenMP and MPI

- Fast quadruple arithmetic operations

- Same interface with Double/Quadruple

Requirement

- C Compiler (necessary)
  - Intel C/C++ 7.0,8.0,9.0,9.1 IBM XL C 7.0
  - SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  - GCC 3.3 or later
- Fortran Compiler (optional)
  - For FORTRAN API: F77 or later
  - For SAAMG Precond.: F90 or later
  - Intel Fortran 8.1,9.0,9.1 IBM XL FORTRAN 9.1
  - SUN WorkShop 6, ONE Studio 7, ONE Studio 11
  - g77 3.3 or later gfortran 4.1(NG for SAAMG) g95 0.91

Steps

1. Initialize
2. Make matrix
3. Make vector
4. Define Solver
5. Set Values
6. Set conditions
7. Execute
8. Finalize

```
LIS_MATRIX    A;
LIS_VECTOR    b,x;
LIS_SOLVER    solver;
int           iter;
double        times,itimes,ptimes;

lis_initialize(argc, argv);
lis_matrix_create(LIS_COMM_WORLD,&A);
lis_vector_create(LIS_COMM_WORLD,&b);
lis_vector_create(LIS_COMM_WORLD,&x);
lis_solver_create(&solver);
lis_input(A,b,x,argv[1]);
lis_vector_set_all(1.0,b);
lis_solver_set_optionC(solver);
lis_solve(A,b,x,solver);
lis_solver_get_iters(solver,&iter);
lis_solver_get_times(solver,&times,&itimes,&ptimes);
printf("iter = %d time = %e (p=%e i=%e)¥n",iter,times, ptimes, itimes);
lis_finalize();
```

Summary

- Fast Quadruple Arithmetic Operations with SSE2
  - Reducing round-off errors
  - Same accuracy with FORTRAN real *16
  - Computation time is about 3.2 times of Double
- A Mixed Precision Iterative method
  - Combination of Double and Fast Quadruple
  - Restart with different precision
  - Select appropriate epsilon is necessary
  - Automatic D-Q SWITCH has proposed.
- Parallel Issues
  - Same data transfer and heavy computation cost
- Library Lis and its GUI Lis-test for free to public
  - More than 1K combinations of precond., solutions, and format.
  - Serial, shared and distributed parallel and Hybrid parallel
  - Fast Quadruple Arithmetic Operations

Conclusion

- DQ-SWITCH is absolutely faster the difficult problems. (Double does not converge!)
- DQ-SWITCH is robust but slower for the ordinary easy problems
- However A Fast Quadruple Arithmetic Operations fits for Parallel computing environments
- Maybe lighter preconditioner are effective for parallel environments and A Fast Quadruple Arithmetic Operations.
- We should test for more problems

Thank you!

ありがとうございました。