# SLAB: A MATLAB-like Interpreter with Result Verification

Shin'ichi OISHI
Waseda University

January 21, 2003, Schloss Dagstuhl

# Abstract

Recently, we have investigated fast verification method of numerical solutions for finite dimensional linear simultaneous equations [1]. Here, we have proposed a rounding mode controlled verification method. Based on this method, we have also proposed fast enclosing method for eigenvalues of matrices [2]. Moreover, we have developed a fast method for monotone sparse linear systems [3] and a verified version of iterative residual method [4], [5].

To test our fast enclosure method, we have developed a new interpreter SLAB. Slab is a MATLAB-like interpreter and has many new features which are suited for verified numerical computation. For example, SLAB has a validation mode in which the solution of Ax=b can be obtained by $x = A \setminus b$ with guaranteed accuracy. SLAB is a GPL-licensed software and can be down loaded from the web cite

`http://www.oishi.info.waseda.ac.jp/~oishi/slab/slab-e.htm`

**1** Shin'ichi Oishi and Siegfried M. Rump: "Fast verification of solutions of matrix equations", Numerische Mathematik **90** (2002) pp.755-773

**2** Shin'ichi OISHI: "Fast Enclosure of Matrix Eigenvalues and Singular Values via Rounding Mode Controlled Computation", Linear Algebra and its Applications, **324** (2001) pp.133-146.

**3** Takeshi Ogita, Shin'ichi Oishi and Yasunori Ushiro: "Fast Verification of Solutions for Sparse Monotone Matrix Equations", Computing [Supple] **15** (2001) pp.175-187.

**4** T. Ogita, S. Oishi and Y. Ushiro:" Fast Inclusion and Residual Iteration for Solutions of Matrix Equations", Computing [Supple] **16** (2002).

**5** T. Ogita, S. Oishi and Y. Ushiro:" Computation of Sharp Rigorous Component wise Error Bounds for Approximate Solutions of System of Linear Equations" to appear in Reliable Computing (2003)

# §0 Fast Inclusion of Solutions for Linear Systems

Let us consider

$$Ax = b, \tag{1}$$

where A is an $n \times n$ matrix and $b$ is an $n$-vector. Assume all elements of $A$ and $b$ are doubles of IEEE754. LU-decompostion method gives an approximate solution with

$$\frac{2}{3}n^3 + O(n^2)$$

floating points operations (FLOPS).

Oishi and Rump have shown that a verification can be done with the same computational cost!

Shin'ichi Oishi and Siegfried M. Rump: "Fast verification of solutions of matrix equations", Numerische Mathematik **90** (2002) pp.755-773

# §1 Rounding Mode Controlled Computation

The key idea is to introduce "Rounding Mode Controlled Computation".

Let us use IEEE754 double. Let $A$ and $B$ be matrices whose elements are all double. Then inclusion of a matrix product $AB$ can be calculated by

```
down();
C=A*B;
up();
D=A*B;  /* AB is included in [C,D] */
```

We can use the optimized BLAS's dgemm in this calculation!

## §2 Fast Verification

Assume that an approximate LU decomposition of $A$, say $L$ and $U$, are calculated for the purpse of calculating approximate solution $\tilde{x}$ of $Ax = b$. Our fast method is

1. Calculates $X_L$ and $X_U$ of approximate inverses of $L$ and $U$, respectively.

2. Evaluate an error bound by

$$\|\tilde{x} - x_{true}\| \leq \frac{\|X_U * X_L * (A\tilde{x} - b)\|_\infty}{\|X_U * X_L * A - I\|_\infty}.$$

Here, we propose to use the following formula which is based on Higham's backward formula:

$$\begin{aligned} & \|X_U * X_L * A - I\|_\infty \\ \leq\ & 2\gamma_n \| |X_U||X_L||L||U| \|_\infty + \gamma_n \| |X_U||U| \|_\infty. \end{aligned}$$

Here,

$$\gamma_n = \frac{nu}{1-nu}, \quad u = 2^{-53}.$$

We can further use

$$|||X_U||X_L||L||U|||_\infty = (|X_U|(|X_L|(|L|(|U|e)))),$$

which shows that $|||X_U||X_L||L||U|||_\infty$ can be calculated with $O(n^2)$ FLOPS. Here, $e = (1, 1, \cdots, 1)^T$.

Example For $1000 \times 1000$ $A$, time for calculating an approximate solution and that for verification are about 1 second, respectively provided that we use Pentium III 1.12Ghz and the optimized BLASS. The method can be applicable up to 5000 dimensional $A$ with random elements.

We have solved 15000 dimensional full matrix problem with 8 CPU PC craster with SCALAPACK and MPI. In the next month, we will use 128 CPU PC craster system.

# §3 What are problems for large problem?

## Accumlation of Rounding Error

We have proposed a verification version of residual iteration.

T.Ogita, S. Oishi and Y. Ushiro:" Fast Inclusion and Residual Iteration for Solutions of Matrix Equations", Computing Suppl. 16 (2002).

Let $\tilde{x}$ be an approximate solution. Let $\tilde{z}$ be an approximate solution of $Az = r,\ r = A\tilde{x} - b$. Then,

$$\|\tilde{z} - x_{true}\|_\infty \leq \|\tilde{z}\|_\infty + \|A^{-1}\|_\infty \|A\tilde{z} - r\|_\infty$$

We must calculate $r$ precisely. For the purpose, Ogita, Rump and me have develop a new efficient arithmetic system which utilize combination of double.

## Memory (Space Complexity)

1. We have shown in principle that verifican can be done with twice space complexity.

2. However, by experiments, we know that to keep the speed it may needed more memory.

3. Thus, the limit is coming from a limitation of memory rather than execution time.

## §4 Eigenvalue Problems

If we use our rounding mode controlled computation, inclusion of all eigenvalues can be done faster than calculation of all eigenvalues and eigenvectors.

Shin'ichi OISHI: "Fast Enclosure of Matrix Eigenvalues and Singular Values via Rounding Mode Controlled Computation", Linear Algebra and its Applications, vol. 324 (2001) pp.133-146.

## §5 Verification using Iterative Solver

For monotone matrix equation, we have developed a verification method which can be implemented on iterative solvers such as CG based method.

Takeshi Ogita, Shin'ichi Oishi and Yasunori Ushiro: "Fast Verification of Solutions for Sparse Monotone Matrix Equations", Computing [Supple] bf 15, (2001) pp.175-187 .

## §6 Software

We have tested our fast methods on various systems:

- C with CLAPACK and optimized BLAS – good bud complicated–
- SCALAPACK and MPI on PC clusters – good bud complicated–
- MATLAB from V to 6.5 –good bud no sourse code and expensive–

- Scilab –good bud LINPACK–

- Octave –good bud LINPACK and no operator overloading–

- Rlab –good bud complicated if we add instructions–

§7 SLAB Thus, I have written my own interpreter.

- Byson and Flex
- Instruction for changing rounding mode.
- real and complex numbers
- GPL-license
- grammer similar to MATLAB but more like C
- linear programming

- gnuplot for graphics

- verification mode

# §8 Builtin Functions up(), down(), near()

Syntax

up(),

down(),

near()

These functions are functions for changing the rounding mode of IEEE754 double precision floating point numbers.

The function up() set the rounding mode to the infinity,

the function down() set the rounding mode to the -infinity,

the function near() to the nearest, respectively.

Slab has many new features which are suited for verified numerical computaion. Such functions are:

**Rounding instructions** up(), down(), and near() are defined for rounding to the infinity, -infinity and to the nearest, respectively.

**validation mode** To enter the validation mode, type !.

In the validation mode, the solution of Ax=b can be obtained by $x = A\backslash b$ with guaranteed accuracy.

Example:

```
A> a=1000000;
A> sin(a)
ans =
      -0.3499935021712917
A> !sin(a)
ans =
      -0.3499935021712929
V> # The correct value of sin(a) is
V> #  -0.34999350217129512...
V> # Thus up to around 10^(-16)
V> # the value of sin(a) is
V> # validated in the validation mode.
V> # On the contrary,
V> # in the usual mode the value of sin(a)
V> # is correct up
V> # to arround 10^(-14).
```

Syntax

int(a,b)

The int instruction is used for making an interval. The object a can be a double and matrix.

Example:

```
A> a=int(3,5)
ans =
       [ 3 , 5 ]
A> A = rand(2);
A> Z = [A,A+0.1]
ans =
       | [0.3 , 0.4 ]  [ -0.1 ,   0 ] |
       | [0.2 , 0.3 ]  [  0.1 , 0.2 ] |
```

The addtion, subtraction, mulitiplication and (division) are overloading.

# §9 Make a User Defined Function

Syntax

```
function func_name(a,b,c,...,m) {
   x = sin(a);
   y = cos(b);
   ...
   z = tan(m);
   result = x+y+z;
}
```

Here, `func_name` is a name of a function which will be defined. Each sentence should be separaed by semicolon, ";".

The value which will be returned is a value of the final sentence.

```
function f(A,b,n) {# validation of Ax=b
    R=inv(A);x=R*b;
    down();
    U=R*A-eye(n);
    s=A*x-b;
    up();
    V=R*A-eye(n);
    t=A*x-b;
    up();
    r=int(s,t);
    T=int(U,V);
    d=abs(T);
    Ar=R*r;
    ar=abs(Ar);
    dd=norm(d);
    arr=norm(ar);
    e=arr/(1-dd);
 }
```

# §10 Builtin Function eig

Syntax

eig(A)

For $n \times n$ matrix A, eig(A) returns its all eigenvalues and eigenvectors

```
A> A=rand(3);
A> sol=eig(A)
ans.val =
          | * * * |
          | * * * |
          | * * * |
ans.vec =
          | * * * |
          | * * * |
          | * * * |
```

In this example, sol.val gives a diagnal matrix whose diagoanl elements consist all eigenvalues of A. On the other hand, n-th column of sol.vec is a eigenvector of A correspoindig to the n-th diagnal element of sol.val.

This function uses CLAPACK functions with optimized BLAS functions:

For a real symmetir A, `dsyev_` is used.

For a real general A, `dgeev_` is used.

For an Hermite A, `zheev_` is used.

For a general complex A, `zgeev_` is used.

## §10' s-file veig

Syntax

veig(A)

The function 'veig(A)' is a verified eigenvalue calculator. Here, A is an n x n matrix, whose eigenvalues are to be determined. If the function "veig(A)" returns a value e, then

$$\min_i |r - r_i| \leq e$$

holds. Here, $r_i$ are calculated eigenvalues of A through eig(A) and r is an eigenvalue of A.

Example:

```
A> # Since 'veig' function is defined in s-file,
A> # one should first
A> # read s-file 'veig.s' by
A> read veig.s
A> # Assume that a matrix A is defined by for exam
A> A=[1,2;3,4];
A> # Then, the eigenvalues of A can be
A> # calculated with guranteed
A> # accuracy by
A> veig(A)
ans =
      0.00000000000000495
```

# §11 Builtin Function dif

Syntax

dif(val,dimension,i)

The dif instruction is used for initializing automatic differentiation. Let $f : R^n \to R$. Here, R is a set of real numbers. The double should be set equal to n. The double i is to designate the partial differentiation with respect to $x_i$.

For example, let $f(a,b)$ is a user-defined function defined by

```
A> function f(a,b) {
A>     a_=a*a-b*b-3*a+2;
A>     b_=2*a*b-3*b;
A>     z_=[a_,b_];
A> }
```

If we set

```
A> a=dif(3,2,0)
ans =
      3  < 1  0 >
A> b=dif(-5,2,1)
ans =
      -5 < 0  1 >
```

Then, if we put a and b into f, we can calculate the value $f(3,-5)$ and the partial derivatives $f_a(3,-5)$ and $f_b(3,-5)$:

```
A> f(a,b)
ans = -23 <   3  10 >
      -15 < -10   3 >
```

# §12 s-file fsolve

Syntax

fsolve(func,x)

The function 'fsolve(func,x)' is a simultanuous nonlinear equation solver based on the Newton method. Here, the func is a name of function, which should be defined by func = name(f) provided that a user-defined function f(x) is defined separately. Then, use like fsolve(func,x) to solve the nonlinear equation

$$f(x) = 0.$$

Here, $x$ is an initial guess of a solution.

Example:

```
A> # Since 'fsolve' function is defined
A> # in s-file, one should first
A> # read s-file 'fsolve.s' by
A> read fsolve.s
A> # Then, define a nonlinear function.
A> function f(x) {
A>     a_=x[0]*x[0]-x[1]*x[1]-3*x[0]+2;
A>     b_=2*x[0]*x[1]-3*x[1];
A>     z_=[a_,b_];
A> }
A> # Then, solve f(x)=0.
A> x=[1;1];
A> a=name(f);
A> y=fsolve(a,x)
ans =
      | 1.000 |
      | 0.000 |
```

# §13 Builtin Function fft

Syntax

fft(a)

or

fft(a,n)

Here, a is 1 x m matrix, and n is to take n-point fft. If n is not specified, n is automatically set m. It should be noted that n must be a power of 2.

ifft(b) calculates the inverse real fft.

Example:

```
A> a=[1,1,1,1,0,0,0,0];
A> b=fft(a)
ans =
     | 4   0   1   2.41424   0   0   1   0.41421
A> c=ifft(b)
ans =
     | 1   1   1   1   0   0   0   0 |
```

# §14 Builtin Function linpro

Syntax

linpro(c,C,b)

The instruction linpro(c,C,b) solves the following linear programming problem:

```
max: c'x;
subject to
     Cx <= b;
     x >= 0;
```

Here, c is an n-dimensional objective vector, C m x n matrix, and b a right hand side vector.

Example:

```
A> c = [-1,2];
A> C = [2,1;-4,4];
A> b = [5,5];
A> linpro(c,C,b)
Value of objective function: 3.75
x0                    1.25
x1                    2.5
```

The function linpro is an interface to the GNU 'lp_solve'.

Remark: Input file of 'lp_solve' is outputted in the file "./linpro/_temp". Out put file of 'lp_solve' is written in the file "./linpro/_out". The command 'lread()' read './linpro/_out' file. For detail, type 'lread'.