A method of obtaining verified solutions for linear systems suited for Java

K. Ozaki [a,*] T. Ogita [b,c] S. Miyajima [c] S. Oishi [c] S.M. Rump [d]

[a] *Graduate School of Science and Engineering, Waseda University, Tokyo 169-8555, Japan*

[b] *CREST, Japan Science and Technology Agency (JST)*

[c] *Faculty of Science and Engineering, Waseda University, Tokyo 169-0072, Japan*

[d] *Institut für Informatik III, Technische Universität Hamburg-Harburg, Hamburg 21071, Germany*

Abstract

Recent development of Java's optimization techniques makes Java one of the most useful programming languages for numerical computations. This paper proposes a numerical method of obtaining verified approximate solutions of linear systems. Usual methods for verified computations use switches of rounding modes defined in IEEE standard 754. However, such switches of rounding modes have not been supported in Java. This method avoids using directed rounding, so that it is implementable on a wide range of programming languages including Java. Numerical experiments using Java illustrate that the method can give a very accurate error bound for an approximate solution of a linear system with almost same computational cost as that for calculating an approximate inverse by the Gaussian elimination.

Key words: Verified computation, Java, linear system

1 Introduction

In this paper, we are concerned with a problem of verifying accuracy of an approximate solution of a linear system

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

* Corresponding author.
Email address: `k_ozaki@suou.waseda.jp` (K. Ozaki).

where A is a real $n \times n$ matrix and b a real n-vector. For this problem, various methods have been proposed (e.g. [12,13]) using the switch of rounding modes defined in IEEE standard 754 double precision floating-point arithmetic. The aim of this paper is to propose a numerical method of obtaining verified error bounds for approximate solutions of linear systems implementable on a wide range of programming language including Java. Here, it should be explained why special attention must be paid for Java. On the one hand, Java has remarkable splendid features. For instance, it is a portable programming language, *i.e.*, it is designed to be independent of operating systems and compilers. Thus, once one develops a Java's program, one can obtain the same result on every platform. The performance of Java has recently been surprisingly increased via developments of its optimization techniques such as Just-In-Time (JIT) compiler and Hot Spot VM. As a result, recently, Java has been used in high performance computing [6]. On the other hand, to keep the portability, the switch of rounding modes in IEEE 754 standard has not been supported in Java (See, [7]). To overcome this and develop a numerical method of obtaining verified error bounds for approximate solutions of linear systems is a main purpose of this paper.

Recently, Oishi and Rump [12] have developed a fast verification method for an approximate solution of (1). Let R denote an approximate inverse of A and I the $n \times n$ identity matrix. Oishi-Rump's method is based on the following fact: If R satisfies

$$\|RA - I\|_\infty < 1 \ , \tag{2}$$

then it is proven that A^{-1} exists and the following inequality holds for an approximate solution $\widetilde{x}$:

$$\|\widetilde{x} - A^{-1}b\|_\infty \leq \frac{\|R(A\widetilde{x} - b)\|_\infty}{1 - \|RA - I\|_\infty} \tag{3}$$

To include quantities appeared in (2) and (3) by numerical computations, the modes of rounding-upward and rounding-downward defined by IEEE standard 754 [1] are controlled dynamically in Oishi-Rump's method. Since the switch of rounding modes has not been supported portably in Java, this method cannot be implemented on Java if one wants to keep the portability of a verification program. Namely, only a way of realizing such a switch in Java is to use JNI (Java Native Interface). Since the use of JNI lessens Java's portability, in this paper it is avoided the use of JNI. Very recently, Ogita, Rump and Oishi [11] have proposed a verification method for linear systems. Ogita-Rump-Oishi's method does not use the directed rounding of IEEE754. Their method uses only the rounding-to-nearest mode to give verified error bounds. For the purpose, they have presented a priori error estimates for floating-point arithmetic. Since Ogita-Rump-Oishi's method uses such a priori error estimates for floating-point arithmetic, it gives usually considerably overestimated error bounds. The aim of this paper is to show that if we combine Ogita-Rump-Oishi's method with the accurate and portable dot product algorithm proposed in [10], then this overestimation can be removed with additional computational cost being almost negligible. Namely, by numerical experiments it will be shown that the proposed method gives a very accurate error bound for an approximate solution of a linear system with almost same computational cost as that for calculating an approximate inverse of a coefficient matrix by the Gaussian elimination.

2 Floating Point Arithmetic in Java

In Java, the formats of IEEE 754 single and double precisions are adopted with respect to the floating-point numbers [4]. The rounding-to-nearest mode is set up in default. However, the switch of rounding modes is not supported. The extended precisions for single and double precisions are admitted by IEEE 754, respectively. In Java, the extended precisions as "widefp mode" are set up in default. However, such extended precisions depend on CPUs in use so that it lessens portability of computational result. To keep the portability, one should use "strictfp mode". In this mode computations are executed strictly in IEEE 754 single or double precision. Therefore, computational results are always the same in every computer environment provided that one uses strictfp mode.

Let $\mathbb{F}$ be a set of floating-point numbers. Let $\mathit{fl}(\cdots)$ be the result of a floating-point computations, where all operations inside parentheses are executed by ordinary floating-point arithmetic only in rounding-to-nearest mode. We assume that over/underflow do not occur (Even if considering the presence of over/underflow, discussions in this paper do not change essentially).

We cite here the notations used in this paper. Let $\mathbf{u}$ be the unit roundoff (especially, $\mathbf{u} = 2^{-53}$ in IEEE 754 double precision). For $n \in \mathbb{N}$, we define $\widetilde{\gamma}_n$ by $\widetilde{\gamma}_n := \mathit{fl}(n\mathbf{u}/(1 - n\mathbf{u}))$. For $x = (x_1, \ldots, x_n)^T \in \mathbb{F}^n$ and $A = (a_{ij}) \in \mathbb{F}^{m \times n}$, the maximum norms are defined as

$$\|x\|_\infty := \max_{1 \le i \le n} |x_i| \quad \text{and} \quad \|A\|_\infty := \max_{1 \le i \le m} \sum_{j=1}^{n} |a_{ij}| \ .$$

For $a, b \in \mathbb{F}$ and $x, y \in \mathbb{F}^n$, we will use the following relations [11]:

$$|a + b| \le (1 + \mathbf{u})\mathit{fl}(|a + b|) \tag{4}$$

$$(1+\mathbf{u})^n|a| \le \mathit{fl}\left(\frac{|a|}{1-(n+1)\mathbf{u}}\right) \tag{5}$$

$$\|x\|_\infty = \mathit{fl}(\|x\|_\infty) \tag{6}$$

$$|x^T||y| \le \mathit{fl}\left(\frac{|x^T||y|}{1-(n+1)\mathbf{u}}\right) \tag{7}$$

$$|x^Ty| \le |\mathit{fl}(x^Ty)| + \mathit{fl}(\tilde{\gamma}_{2n+1}|x^T||y|) \tag{8}$$

Note that $x \le y$ means $x_i \le y_i$ for all i. Moreover, we denote by $|x|$ the non-negative vector with $|x| = (|x_1|, \ldots, |x_n|)^T$. For real matrices, similar notations will be used.

3 Accurate Dot Product with Error Bound

In this section, we briefly review an accurate algorithm of calculating dot products and matrix-vector products with error bounds proposed in [10].

For $a, b \in \mathbb{F}$, it is well-known that there exist algorithms to transform the sum $a+b$ into $x+y$ with $x = fl(a+b), y \in \mathbb{F}$, *i.e.*, $a+b = x+y$ (Knuth [8]) and the product $a \cdot b$ into $x+y$ with $x = fl(a \cdot b), y \in \mathbb{F}$, *i.e.*, $a \cdot b = x+y$ (Dekker [5]). We denote the algorithms as $[x, y] = \texttt{TwoSum}(a, b)$ and $[x, y] = \texttt{TwoProduct}(a, b)$, respectively (See, [10] for detail). These are so-called *error-free transformations* of floating-point arithmetic. Using these error-free algorithms, Ogita, Rump and Oishi developed the following algorithm to calculate dot product with error bound in twice the working precision even in the presence of underflow.

Algorithm 1 (Ogita, Rump and Oishi [10]) *For $x, y \in \mathbb{F}^n$, the following algorithm calculates an approximation* res *of x^Ty and its error bound* err *such that* $\texttt{res} - \texttt{err} \le x^Ty \le \texttt{res} + \texttt{err}$ *in twice the working precision.*

```
function  $[\texttt{res}, \texttt{err}] = \texttt{Dot2Err}(x, y)$
if $2n\mathbf{u} \ge 1$,  error('inclusion failed'),  end
$[p, s] = \texttt{TwoProduct}(x_1, y_1)$                % $p + s \leftarrow x_1 + y_1$
$e = |s|$
for $i = 2 : n$
   $[h, r] = \texttt{TwoProduct}(x_i, y_i)$             % $h + r \leftarrow x_i + y_i$
   $[p, q] = \texttt{TwoSum}(p, h)$                     % $p + q \leftarrow p + h$
   $t = \mathit{fl}(q + r)$;   $s = \mathit{fl}(s + t)$;   $e = \mathit{fl}(e + |t|)$
end
$\texttt{res} = \mathit{fl}(p + s)$;   $\delta = \mathit{fl}((n\mathbf{u})/(1 - 2n\mathbf{u}))$
$\alpha = \mathit{fl}(\mathbf{u}|\texttt{res}| + (\delta e + 3\underline{\mathbf{u}}/\mathbf{u}))$          % $\underline{\mathbf{u}}$: underflow unit
$\texttt{err} = \mathit{fl}(\alpha/(1 - 2\mathbf{u}))$
```

Algorithm 1 consists of only ordinary floating-point operations. Thus, we can calculate an accurate dot product and its error bound without directed rounding. We can use Algorithm 1 in the calculation of matrix-vector products. As a result, we have the following algorithm:

Algorithm 2 *For $A = (a_{ij}) \in \mathbb{F}^{m\times n}$ and $x \in \mathbb{F}^n$, the algorithm calculates an approximation y of Ax and its error bound r such that $y - r \le Ax \le y + r$.*

```
function  $[y, r] = \texttt{MV2Err}(A, x)$
$[m, n] = \texttt{size}(A)$                 % $m \times n$ matrix
for $i = 1 : m$
    $w = A(i, 1 : n)$                   % $w = (a_{i1}, \ldots, a_{in})$
    $[y_i, r_i] = \texttt{Dot2Err}(w^T, x)$
```

4 Verification Method

In this section, we shall propose a numerical method of obtaining verified error bounds for approximate solutions of the linear system (1). This method avoids using directed rounding. Thus, it is implementable on a wide range of programming languages including Java.

4.1 Ogita-Rump-Oishi's Method

Ogita, Rump and Oishi [11] have proposed a verification method for the linear system (1). Ogita-Rump-Oishi's method does not use the directed rounding of IEEE754 and use only the rounding-to-nearest mode to give verified error bounds. For the purpose, they have presented a priori error estimates for floating-point arithmetic in the estimation of (2).

From (3), it is easily derived that if α and $\beta (\alpha, \beta \in \mathbb{F})$ such that $\|RA - I\|_\infty \le \alpha < 1$ and $\|R(A\tilde{x} - b)\|_\infty \le \beta$ can be estimated, an error bound of an approximate solution $\tilde{x}$ of (1) is given as

$$\|\tilde{x} - A^{-1}b\|_\infty \le \mathit{fl}\left(\frac{\beta/(1-\alpha)}{1-3\mathbf{u}}\right) . \tag{9}$$

Ogita-Rump-Oishi's method gives a way of calculating α and β. Here, we shall briefly review the method. First, the algorithm of calculating α is as follows:

Algorithm 3 (Ogita, Rump and Oishi [11]) *For $A \in \mathbb{F}^{n\times n}$ and R being its approximate inverse, the following algorithm calculates an upper bound α of $\|RA - I\|_\infty$.*

```
function  $\alpha = \texttt{Alpha.Std}(A, R)$
if $(3n+2)\mathbf{u} \ge 1$, $\texttt{error}$('verification failed'), end
$\alpha_1 = \mathit{fl}(\|RA - I\|_\infty)$
if $\alpha_1 \ge 1$, $\texttt{error}$('verification failed'), end
$\alpha_2 = \mathit{fl}(\||R|(|A|e)\|_\infty)$      % $e = (1, \dots, 1)^T$
$\alpha = \mathit{fl}\,((\alpha_1 + \widetilde{\gamma}_{3n+2}(\alpha_2 + 2))/(1 - 2\mathbf{u}))$
```

From Algorithm 3, it holds approximately that

$$\alpha \approx \alpha_1 + n\mathbf{u}\cdot \mathrm{cond}_\infty(A) \lesssim \alpha_1 + n^2\mathbf{u}\cdot \mathrm{cond}_2(A)\ , \tag{10}$$

where $\mathrm{cond}_p(A)$ denotes the condition number defined as $\mathrm{cond}_p(A) := \|A\|_p \cdot \|A^{-1}\|_p$ for $p = 2, \infty$. From (10), one can expect that Algorithm 3 can be applicable provided $\mathrm{cond}_2(A) \lesssim (n^2\mathbf{u})^{-1}$. For instance, assume that $n = 1000$ and the use of IEEE 754 double precision. Then, the Algorithm 3 might be applicable up to the problems with $\mathrm{cond}_2(A)$ being $(1000^2 \cdot 2^{-53})^{-1} \approx 10^{10}$. This will be confirmed by numerical experiments in Section 5.

The algorithm of calculating β given by Ogita-Rump-Oishi is as follows:

Algorithm 4 (Ogita, Rump and Oishi [11]) *For $A \in \mathbb{F}^{n\times n}$, $b \in \mathbb{F}^n$, $\widetilde{x}$ being an approximate solution of $Ax = b$ and R an approximate inverse of A, the following algorithm calculates an upper bound β_1 on $\|R(A\widetilde{x} - b)\|_\infty$.*

```
function  $\beta_1 = \texttt{Beta.Std}(A, b, \widetilde{x}, R)$
$r_{\mathrm{mid}} = \mathit{fl}\,(A\widetilde{x} - b)$
$r_{\mathrm{rad}} = \mathit{fl}\,(\widetilde{\gamma}_{2n+4}(|A|\,|x| + |b|))$
$q = \mathit{fl}\,(|R|\,(\widetilde{\gamma}_{n+1}\,|r_{\mathrm{mid}}| + r_{\mathrm{rad}})/(1 - (n+3)\mathbf{u}))$
$\beta_1 = \mathit{fl}\,((\|\,|R\,r_{\mathrm{mid}}| + q\|_\infty)/(1 - 2\mathbf{u}))$
```

Here, let us characterize Algorithm 4. In case of $\widetilde{x}$ being a good approximate solution of $Ax = b$, big cancellations might occur in the course of calculating a residual $A\widetilde{x} - b$. To be precise, it can be approximately estimated that

$$|Rr_{\mathrm{mid}}| \le |R|\,|r_{\mathrm{mid}}| \approx \mathbf{u}\,|R|\,|b| \quad \text{and} \quad q \approx n\mathbf{u}\,|R|\,(|A||\widetilde{x}| + 2|b|)\ .$$

Therefore, at the very end of Algorithm 4, the second term of numerator may become much larger than the first term. This implies the existence of overestimation, which cannot be avoided if one uses a priori error estimates.

4.2 A New Method

To eliminate overestimations pointed out in the previous subsection, we here present an improved estimation method for $\|R(A\tilde{x}-b)\|_\infty$. For the purpose, we use the accurate and portable algorithm of calculating dot product with error bound discussed in Section 3. We first consider a method of inclusion for the residual $A\tilde{x}-b$. Let $\hat{A}$ and $\hat{x}$ be given by $\hat{A}=(A|b)$ and $\hat{x}=(\tilde{x}_1,...,\tilde{x}_n,-1)^T$, respectively. Obviously, $A\tilde{x}-b=\hat{A}\hat{x}$, so that Algorithm 2 (`MV2Err`) can be applied to $\hat{A}\hat{x}$. Algorithm 2 generates an approximation r_{mid} of $A\tilde{x}-b$ and its error bound r_{rad} such that $r_{\mathrm{mid}}-r_{\mathrm{rad}} \le A\tilde{x}-b \le r_{\mathrm{mid}}+r_{\mathrm{rad}}$. Then we have (cf., for example, [12])

$$t_1 - t_2 \le R(A\tilde{x}-b) \le t_1 + t_2 \ , \tag{11}$$

where t_1 and t_2 are defined by $t_1 := R\,r_{\mathrm{mid}}$ and $t_2 := |R|\,r_{\mathrm{rad}}$. It follows that

$$|R(A\tilde{x}-b)| \le |R\,r_{\mathrm{mid}}| + |R|\,r_{\mathrm{rad}} \ . \tag{12}$$

Applying the estimation (8) to the matrix-vector product $R\,r_{\mathrm{mid}}$ yields

$$|R\,r_{\mathrm{mid}}| \le s_1 + s_2 \ , \tag{13}$$

where s_1 and s_2 are defined by $s_1 := |fl(R\,r_{\mathrm{mid}})| = fl(|R\,r_{\mathrm{mid}}|)$ and $s_2 := fl(\tilde{\gamma}_{2n+1}(|R|\,|r_{\mathrm{mid}}|))$. Similarly, we have

$$|R|\,r_{\mathrm{rad}} \le fl\left(\frac{|R|\,r_{\mathrm{rad}}}{1-(n+1)\mathbf{u}}\right) =: s_3 \ . \tag{14}$$

Inserting (13) and (14) into (12), we have

$$|R(A\tilde{x}-b)| \le s_1 + s_2 + s_3 \ . \tag{15}$$

This and the results obtained by applying (4) to every components of the right hand side of (12) yield

$$|R(A\tilde{x}-b)| \le (1+\mathbf{u})^2 fl\left(s_1 + (s_2+s_3)\right) \ . \tag{16}$$

Note that it usually holds $s_2+s_3 \ll s_1$, using (5) and (6), we finally have

$$\begin{aligned}\|R(A\widetilde{x}-b)\|_\infty &\le \|(1+\mathbf{u})^2 fl\,(s_1+(s_2+s_3))\,\|_\infty \\ &\le fl\left(\frac{\|s_1+(s_2+s_3)\|_\infty}{1-3\mathbf{u}}\right) =: \beta_2\ . \end{aligned} \tag{17}$$

We can now present the following theorem:

Theorem 1 *Let $A \in \mathbb{F}^{n\times n}$ and $b \in \mathbb{F}^n$. Let R be an approximate inverse of A and $\widetilde{x}$ an approximate solution of $Ax = b$. Assume that $r_{\rm mid}$ and $r_{\rm rad}$ satisfy*

$$r_{\rm mid} - r_{\rm rad} \le A\widetilde{x} - b \le r_{\rm mid} + r_{\rm rad}\ .$$

Then the following inequality holds in the absence of underflow:

$$\|R(A\widetilde{x}-b)\|_\infty \le fl\,(\|s_1+(s_2+s_3)\|_\infty/(1-3\mathbf{u}))\ , \tag{18}$$

where s_1, s_2 and s_3 are defined by $s_1 := fl(|R\,r_{\rm mid}|)$, $s_2 := fl(\widetilde{\gamma}_{2n+1}(|R|\,|r_{\rm mid}|))$ and $s_3 := fl\,(|R|\,r_{\rm rad}/(1-(n+1)\mathbf{u}))$.

Now several estimations are in order. First, consider β_2. It can be estimated that

$$\|s_1+s_2+s_3\| \approx \|R\,r_{\rm mid}\| + n\mathbf{u}\||R|\,|r_{\rm mid}|\| + \||R|\,r_{\rm rad}\|\ .$$

If $\|r_{\rm rad}\| \approx \mathbf{u}\|r_{\rm mid}\|$, i.e. if the norm of radius $\|r_{\rm rad}\|$ is relatively small compared with the norm of midpoint $\|r_{\rm mid}\|$, then the third term can be neglected and

$$\|s_1+s_2+s_3\| \approx \|R\,r_{\rm mid}\| + n\mathbf{u}\||R|\,|r_{\rm mid}|\|\ .$$

Normally, $\|R\,r_{\rm mid}\| \gg n\mathbf{u}\||R|\,|r_{\rm mid}|\|$ holds so that

$$\|s_1+s_2+s_3\| \approx \|R\,r_{\rm mid}\|\ .$$

Since using the accurate dot product algorithm explained in the previous section one can calculate $r_{\rm mid}$ and $r_{\rm rad}$ as one uses higher precision arithmetic, $\|r_{\rm rad}\| \approx \mathbf{u}\|r_{\rm mid}\|$ might hold. Thus, it can be expected that β_2 becomes a tight upper bound of $\|R(A\widetilde{x}-b)\|_\infty$.

Using Theorem 1 and Algorithm 2, we now present the following algorithm.

Algorithm 5 *Let A, b, R and $\widetilde{x}$ be as in Theorem 1, then the following algorithm gives β_2 such that $\|R(A\widetilde{x}-b)\|_\infty \le \beta_2$.*

```
function  $\beta_2$ = Beta.New($A, \widetilde{x}, b, R$)
```

```
$[r_{\text{mid}}, r_{\text{rad}}] = \texttt{MV2Err}([A, b], [\tilde{x}; -1])$   % $\hat{A} = (A|b),\ \hat{x} = (\tilde{x}_1, ..., \tilde{x}_n, -1)^T$
$s_1 = fl(|R\, r_{\text{mid}}|)$
$s_2 = fl(\tilde{\gamma}_{2n+1}(|R|\, |r_{\text{mid}}|))$
$s_3 = fl\left((|R|\ r_{\text{rad}})/(1 - (n+1)\mathbf{u})\right)$
$\beta_2 = fl\left(\|s_1 + (s_2 + s_3)\|_\infty / (1 - 3\mathbf{u})\right)$
```

Estimating α by Algorithm 3 and β by β_2, (9) gives an error bound of an approximate solution of $Ax = b$. In this calculation, no directed rounding is used. Moreover, in this method $2n^3$ flops computation is needed for the calculation of $RA - I$ and other computations are $\mathcal{O}(n^2)$. Thus theoretical computational cost of the method is almost same as that for calculating an inverse of an $n \times n$ matrix by the Gaussian elimination. In the remaining part of this paper, using numerical experiments we will show that the method can give a very accurate error bound for an approximate solution of a linear system with almost same computational cost as that for calculating an approximate inverse of a coefficient matrix by the Gaussian elimination.

5 Numerical Experiments

We now illustrate the effectiveness of the algorithm proposed in the previous section. At present, some public domain numerical software libraries have been developed for Java, for example, JLAPACK [2], JAMA [9] and JAMPACK [14]. In Table 1, we display the comparison of computing time for matrix multiplication and that for solving linear systems by the above libraries. Here, we used a computer with Pentium IV 1.7GHz CPU, J2SDK1.4.2_06 as Java compiler and virtual machine (VM). Table 1 shows that JAMPACK is slower than the other libraries because it seems to treat even real numbers as complex numbers. From Table 1, we can confirm that JLAPACK can calculate matrix multiplication and solve a linear system fairly faster than the others. Unfortunately, JLAPACK has not supported the function of calculating the matrix inverse yet. On the other hand, JAMA has already supported it and other auxiliary functions. Considering these facts, we use JAMA for our numerical experiments as building blocks.

Following four methods are implemented on a PC with Pentium IV 1.7GHz CPU, J2SDK1.4.2_06 as Java compiler and VM with strictfp mode:

Method A Oishi-Rump method [12, Algorithms 3.1 and 3.2]
Method B Ogita-Rump-Oishi method (Algorithms 3 and 4)
Method C Method A with Algorithm 2
Method D Proposed method (Algorithms 3 and 5)

Table 1
Comparison of computing time (sec) by various matrix computation libraries.

	Matrix multiplication			Solving a linear system		
n	JLAPACK	JAMA	JAMPACK	JLAPACK	JAMA	JAMPACK
100	0.02	0.02	0.07	0.06	0.02	0.31
500	1.07	2.32	5.82	0.81	0.88	2.09
1000	8.44	17.9	47.7	5.71	6.47	12.7
2000	66.6	142	383	43.1	49.0	94.1

Table 2
Comparison of error bounds on $\|\widetilde{x} - A^{-1}b\|_\infty$ for various n.

n	**A**	**B**	**C**	**D**
100	1.43e-11	7.91e-11	2.28e-14	2.28e-14
500	2.55e-09	1.52e-08	8.75e-13	8.75e-13
1000	8.66e-09	5.16e-08	1.90e-12	1.90e-12
2000	6.10e-08	3.63e-07	3.95e-12	3.95e-12

All computations are done in double precision. In Methods A and C, we introduce JNI for switch of rounding mode. In Methods B and D, to switch rounding mode is not necessary so that Java's portability is kept. We use JAMA discussed in Section 5 with respect to calculating $\widetilde{x}$ and R in (3).

First, we choose A an $n \times n$ matrix whose entries are pseudo-random floating-point numbers uniformly distributed in $[-1, 1]$. We put $b := fl(A \cdot e)$ with $e := (1, \ldots, 1)^T$. In Tables 2 and 3, we display the error bound for $\widetilde{x}$ and its computing time by applying the each method to (1) for various n, respectively. In Table 3, the computing time for LU factorization (**LU**) and for calculating R (**INV**) are also displayed.

By Table 2, we can confirm that Methods C and D give tighter error bounds than Methods A and B. We can also confirm that Method D supplies almost same error bounds with those given by Method C. Table 3 shows that Method D is faster than Methods A and C. Moreover, it can be seen that the speed of Method D becomes approximately equal to that of Method B as n increases.

Next, we vary the condition number $\mathrm{cond}_2(A)$ of $A \in \mathbb{F}^{1000\times1000}$. In Table 4, we display error bounds for $\widetilde{x}$ when we apply the each method to (1) for various $\mathrm{cond}_2(A)$. Here, the entries of A and b are set similar to the previous example. As $\widetilde{x}$, we use the most accurate solution in double precision obtained by an iterative refinement method (cf. [3, pp. 126–127]). The notation "–" means that the verification failed.

Table 3
Comparison of computing time (sec) for various n.

n	**LU**	**INV**	**A**	**B**	**C**	**D**
100	0.02	0.05	0.07	0.04	0.10	0.07
500	0.86	2.90	4.63	2.35	4.95	2.54
1000	6.50	21.6	36.0	18.1	36.9	18.8
2000	50.0	174	284	143	287	145

Table 4
Comparison of error bounds on $\|\widetilde{x} - A^{-1}b\|_\infty$ for various $\mathrm{cond}_2(A)$ ($n = 1000$).

$\mathrm{cond}(A)$	**A**	**B**	**C**	**D**
10^2	1.44e-11	9.49e-10	1.11e-16	1.11e-16
10^4	7.19e-10	4.68e-08	1.11e-16	1.11e-16
10^6	5.52e-08	3.66e-06	1.11e-16	1.11e-16
10^8	4.22e-06	2.68e-04	1.11e-16	1.11e-16
10^{10}	3.54e-04	2.45e-02	1.11e-16	1.14e-16
10^{12}	2.98e-02	–	1.14e-16	–

Table 4 shows that Methods A and B supply coarse error bounds when $\mathrm{cond}_2(A)$ is large even if $\widetilde{x}$ has much more accuracy. As opposed to this, even if $\mathrm{cond}_2(A)$ is large, Methods C and D supply tight error bounds with almost maximum accuracy in double precision. On the other hand, the verification failed ($\alpha \geq 1$) in Methods B and D when $\mathrm{cond}_2(A) = 10^{12}$. This comes from the fact that Method D overestimates $\|RA - I\|_\infty$ as seen from (10). This result matches the discussions in Section 4.1.

In conclusion, numerical experiments using Java illustrate that although the proposed method (Method D) can apply to the problems with less condition number than Methods A and C, it can give a very accurate error bound for an approximate solution of a linear system with almost same computational cost as that for calculating an approximate inverse of a coefficient matrix by the Gaussian elimination.

Acknowledgements

This research was partially supported by CREST program, Japan Science and Technology Agency (JST), 21st Century COE Program (Productive ICT Academia Program, Waseda University) and Grant-in-Aid for Young Scientists (B) (16700017, 2004–2006) from the Ministry of Education, Science,

Sports and Culture of Japan.

References

[1] ANSI/IEEE, IEEE Standard for Binary Floating Point Arithmetic, Std 754–1985 edition, IEEE, New York, 1985.

[2] B. Blount, JLAPACK – The LAPACK library in Java.
`http://www.cs.unc.edu/Research/HARPOON/jlapack/`

[3] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd edition, Johns Hopkins University Press, Baltimore and London, 1996.

[4] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Addison-Wesley, 2nd edition, 2000.

[5] T.J. Dekker, A floating-point technique for extending the available precision, Numer. Math., 18: 224–242, 1971.

[6] S. Flynn-Hummel, V. Getov, F. Irigoin, Ch. Lengauer, High performance computing and Java, Report No. 284, Report of the Dagstuhl Seminar 00341, 2000.

[7] W. Kahan, J.D. Darcy, How Java's Floating-Point Hurts Everyone Everywhere, manuscript, 2001.
`http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf`

[8] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, volume 2, Addison-Wesley, Reading, Massachusetts, 1969.

[9] MathWorks Inc., NIST, JAMA – A Java Matrix Package.
`http://math.nist.gov/javanumerics/jama/`

[10] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput., 26(6): 1955-1988, 2005.

[11] T. Ogita, S.M. Rump, S. Oishi, Verified solution of linear systems without directed rounding, Technical Report, No. 2005-04, Advanced Research Institute for Science and Engineering, Waseda University.

[12] S. Oishi, S.M. Rump, Fast verification of solutions of matrix equations, Numer. Math., 90(4): 755–773, 2002.

[13] S.M. Rump, Verification methods for dense and sparse systems of equations, Topics in Validated Computations – Studies in Computational Mathematics (J. Herzberger ed.), 63–136, Elsevier, Amsterdam, 1994.

[14] G.W. Stewart, JAMPACK – A Java Package for Matrix Computations.
`ftp://math.nist.gov/Jampack/Jampack/AboutJampack.html`