数理解析研究所講究録 1441 巻 2005 年 75-88

Javaによる連立一次方程式の数値解の精度保証法

尾崎 克久* 荻田 武史† 宮島 信也† 大石 進一†

* 早稲田大学大学院理工学研究科 (k_ozaki@suou.waseda.jp)
† 早稲田大学理工学術院

概 要

Javaは近年の最適化技術などによって数値計算や数値シミュレーションを行う言語としても注目されてきている．従来の連立一次方程式の数値解の精度保証法は浮動小数点演算の丸めモードの変更を必要とするが，JavaではIEEE 754規格で定められている丸めモードの変更がサポートされていない．そこで本報告では丸めモードの変更を必要としない連立一次方程式の数値解の高速かつ高精度な精度保証法を提案する．最後に，数値実験によって提案手法の有効性を示す．

Numerical Verification Method for Systems of Linear Equations in Java

Katsuhisa OZAKI*, Takeshi OGITA†, Shinya MIYAJIMA†, and Shin'ichi OISHI†

* Graduate School of Science and Engineering
† Faculty of Science and Engineering

Abstract

Recently, Java seems to become a useful programming language for numerical computations by development of optimization technique. Usual methods for verifying the accuracy of numerical solutions of linear systems use the switches of rounding modes defined in IEEE 754 standard. However, such switches of rounding modes have not been supported in Java. In this paper, we will propose a verification method for linear systems without directed rounding. The method can give accurate error bounds for the numerical solutions at high speed. Finally, numerical results will be presented to show the performance of proposed method.

1 まえがき

本報告では，連立一次方程式

$$Ax = b, \quad A \in \mathbb{R}^{n\times n}, \quad b \in \mathbb{R}^n \tag{1}$$

の数値解の精度保証をJavaにおいて実行する方法について考える．ここで，数値解とは計算機によって得られる近似解のことであり，精度保証とは厳密解に対する近似解の定量的誤差評価のことを指す．

Java は OS 等に依存しないポータブルな言語であり，一度実装すれば異なるプラットフォームにおいて同一の結果を得ることができる．従来，Java は数値計算の分野においては処理速度の点で問題視されていたが，近年の Just-In-Time (JIT) コンパイラや Hot Spot VM 等の最適化技術により，その点もかなり改善することが可能となってきた [14]．したがって，Java は数値計算や数値シミュレーションを行う言語としても注目されてきている [4, 13, 15]．しかしながら，一方では Java における数値計算の様々な問題点も指摘されている [6]．

精度保証付き数値計算の分野では，近年，Oishi と Rump によって連立一次方程式のための高速な精度保証法が提案された [11]．Oishi-Rump の方法では，式 (1) の近似解を $\tilde{x}$，A の近似逆行列を R，I を $n \times n$ 単位行列とすると

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

を証明できた場合に

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

という誤差評価式を用いて精度保証を行っている．このとき，IEEE 754 規格 [1] に定められている 4 つの丸めモード

- 最近点への丸め (デフォルト)
- $+\infty$ 方向への丸め
- $-\infty$ 方向への丸め
- 原点方向への丸め

を適宜変更する必要がある．Oishi-Rump の方法では「$+\infty$ 方向への丸め」及び「$-\infty$ 方向への丸め」を用いている．ところが，現在の Java における浮動小数点演算の規格では，丸めモードの変更が定められていない．そのため，Java で丸めモードの変更を実装するには，C 言語などで丸めモードの変更を行うライブラリを生成し，Java Native Interface (JNI) を用いなければならない．しかしながら，他言語で生成したライブラリを用いると，Java の大きな利点の 1 つであるソースコードのポータビリティ(アーキテクチャ，OS，コンパイラなどに依存しない) が失われてしまう．Java のポータビリティを保持するためにはデフォルトの丸めモードである「最近点への丸め」のみで精度保証を行う必要があるため，従来の丸めモードの変更を必要とする精度保証法を用いることができない．

これに対し，浮動小数点演算の事前誤差評価を用いて丸めモードを変更せずに最近点への丸めのみで高速に連立一次方程式の数値解を精度保証する手法を Ogita らが提案している [10]．我々はその手法を拡張し，高精度内積計算アルゴリズム [9] を用いた新しい手法を提案する．これにより Java のポータビリティが損なわれない上に，精度保証が失敗しない限り，従来の手法よりも数値解の持つ精度をできるだけ過大評価することなく保証することが可能となる．本報告の最後では，数値実験によって提案手法の有効性を示す．

2 準備

本章では，本報告で用いるJavaの浮動小数点システム，各種記号，浮動小数点演算に関するいくつかの公式について説明をする．

まずJavaにおける浮動小数点システム[5]について簡潔に述べる．Javaでは浮動小数点数にIEEE 754の単精度及び倍精度の表現形式を用いている．デフォルトの丸めモードはIEEE 754の最近点への丸めモードであり，Java単独では丸めモードの切り替えが出来ない．またIEEE 754は浮動小数点演算の計算過程において，単精度もしくは倍精度を超えた拡張精度での内部演算を認めている．拡張精度に関してはCPUに依存するため，浮動小数点演算の結果が異なることがある．現在のJavaにおける浮動小数点演算の規格では，この拡張精度で内部演算を行うwidefpモードがデフォルトとなっている．ただしJavaのstrictfpモードを使用すると，拡張精度を用いずにすべてIEEE 754の単精度及び倍精度で演算が行われる．そのためstrictfpモードを用いて得られる演算結果はCPUに依存しないためポータビリティを持つ．

次に各種記号についての説明をする．$\mathbb{F}$を浮動小数点数の集合とし，$\mathrm{fl}(\cdot)$は括弧中の演算をすべて浮動小数点演算によって実行したときの結果を表す．たとえば，$x, y \in \mathbb{F}^n$に対して$\mathrm{fl}(x^T y)$は浮動小数点演算によって計算した内積の値を意味する．浮動小数点演算において，アンダーフローは起きないと仮定する[1]．また，$\mathbf{u}$を浮動小数点演算の相対精度(unit roundoff)とする[2]．

次に，本報告で利用する式を以下に示す[10]．$a, b \in \mathbb{F}$に対して

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

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

が成り立つ．また，$x, y \in \mathbb{F}^n$に対して

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

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

$$|x^T y| \leq |\mathrm{fl}(x^T y)| + \mathrm{fl}(\tilde{\gamma}_{n,2n+2}|x^T||y|) \tag{8}$$

が成り立つ．ただし，$m, n \in \mathbb{N}$に対して$\tilde{\gamma}_{m,n}$を

$$\tilde{\gamma}_{m,n} := \mathrm{fl}\left(\frac{m\mathbf{u}}{1-n\mathbf{u}}\right) \tag{9}$$

と定義する．また，ベクトルと行列の最大値ノルムについては，$x = (x_1, \ldots, x_n)^T$及び$A = (a_{ij}) \in \mathbb{F}^{m\times n}$に対して

$$\|x\|_\infty := \max_{1\leq i\leq n} |x_i|, \qquad \|A\|_\infty := \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}| \tag{10}$$

である．

[1]アンダーフローを考慮しても議論は大きくは変わらない．

[2]IEEE 754倍精度では，$\mathbf{u} = 2^{-53}$となる．

3 高精度演算アルゴリズム

本章では，提案する精度保証法において用いる内積や行列・ベクトル積の高精度演算について述べる．

まず，浮動小数点演算の加算及び乗算に対する誤差の計算方法について説明する．これらは，内積計算や行列・ベクトル積を高精度に計算するアルゴリズムの中で用いる．Knuthは2つの浮動小数点数の和 $a+b$ を近似値 x と誤差 y の和 $x+y$ $(x,y\in\mathbb{F})$ に変換するアルゴリズムを提案している [7]．このとき，$a+b=x+y$ が厳密に成り立つ．そのアルゴリズムを以下に記す．本報告中のアルゴリズムはコード記述の簡略化のために Matlab に似たコードで記述する．

Algorithm 1 (Knuth [7]) $a,b\in\mathbb{F}$ に対して $a+b$ の近似値 x とその誤差 y $(x,y\in\mathbb{F})$ を求めるアルゴリズム．

$$
\begin{aligned}
&\text{function}\quad [x,y]=\texttt{TwoSum}(a,b)\\
&x=\mathrm{fl}(a+b)\\
&z=\mathrm{fl}(x-a)\\
&y=\mathrm{fl}((a-(x-z))+(b-z))
\end{aligned}
$$

□

次に Dekker による仮数部が p ビットの浮動小数点数 a に対し $s=\lceil p/2\rceil$ とし上位 s ビットに対応する x と下位 $p-s$ ビットに対応する y $(x,y\in\mathbb{F})$ に誤差なく分解するアルゴリズム [3] を以下に記す[3]．

Algorithm 2 (Dekker [3]) $a\in\mathbb{F}$ に対して，仮数部が p ビットである浮動小数点を上位 $s=\lceil p/2\rceil$ ビットに対応する x と下位 $p-s$ ビットに対応する y $(x,y\in\mathbb{F})$ に分解するアルゴリズム．

$$
\begin{aligned}
&\text{function}\quad [x,y]=\texttt{Split}(a)\\
&c=\mathrm{fl}(\texttt{factor}\cdot a)\quad \%\ \texttt{factor}=2^s+1\\
&x=\mathrm{fl}(c-(c-a))\\
&y=\mathrm{fl}(a-x)
\end{aligned}
$$

□

上記アルゴリズムを利用して，Dekker と Veltkamp は2つの浮動小数点数の積 $a\cdot b$ をその近似値 x と誤差 y $(x,y\in\mathbb{F})$ の和に変換するアルゴリズムを提案している [3]．このとき，$a\cdot b=x+y$ が厳密に成り立つ．そのアルゴリズムを以下に記す．

Algorithm 3 (Dekker [3]) $a,b\in\mathbb{F}$ に対し $a\cdot b$ の近似値 x と誤差 y $(x,y\in\mathbb{F})$ を求めるアルゴリズム．

[3]たとえば，IEEE 754 倍精度の場合，$p=53$, $s=27$ となる．

function $[x, y] = \texttt{TwoProduct}(a, b)$
$x = \mathrm{fl}(a \cdot b)$
$[a_1, a_2] = \texttt{Split}(a)$
$[b_1, b_2] = \texttt{Split}(b)$
$y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$

□

上記アルゴリズムを用いてOgitaらはハードウェアでサポートされている通常の浮動小数点演算，特に，加減算及び乗算のみを用いて内積計算を高速かつ高精度に行う手法を提案している[9]．これはTwoSumやTwoProductにより得られる誤差を計算値に還元していく手法であり，実行レベルでの最適化に優れていることから高速に行えることが示されている．その中で，提案手法で用いるアルゴリズムDot2Errを以下に記す．Dot2Errは，使用している精度に対してその倍の精度(倍精度であれば4倍精度)で内積を計算し，その計算値と誤差の上限を返す関数である．

Algorithm 4 (Ogita-Rump-Oishi [9]) $x, y \in \mathbb{F}^n$に対し，内積$x^T y$を2倍の精度で計算したときの計算値$res \in \mathbb{F}$と誤差の上限$err \in \mathbb{F}$を求めるアルゴリズム．

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

□

表1は，各成分が$[-1, 1]$の一様乱数であるベクトル$x, y \in \mathbb{F}^n$に対してnを10から10000まで変化させてDot2Errを用いたときの，相対誤差$err/|res|$の値である．数値実験ではJavaのstrictfpモードで倍精度を使用した．
この結果から，内積計算における誤差の上限を相対的に非常に小さく計算できることがわかる．

表 1: 乱数ベクトルに対する Dot2Err の相対誤差

n	$err/\|res\|$
10	1.12e-16
100	1.12e-16
1000	1.12e-16
10000	1.12e-16

次に，Algorithm 4 (Dot2Err) を行列とベクトルの積に適用できるように拡張し，$A \in \mathbb{F}^{n\times n}$，$x \in \mathbb{F}^n$ に対して

$$y - r \leq Ax \leq y + r \tag{11}$$

を満たすような $y, r \in \mathbb{F}^n$ を求めるアルゴリズムを以下に記す．このとき y, r は

$$y = \mathrm{fl}(Ax), \quad |Ax - \mathrm{fl}(Ax)| \leq r \tag{12}$$

を満たす．ただし，ベクトル $v = (v_1, \ldots, v_n)^T$ に対して $|v|$ を $|v| := (|v_1|, \ldots, |v_n|)^T$ と定義し，ベクトルにおける不等式は各成分ごとに不等式が成立することを意味する．また，行列に対しても同様の表記を用いる．

Algorithm 5 $A \in \mathbb{F}^{m\times n}$, $x \in \mathbb{F}^n$ に対し $y - r \leq Ax \leq y + r$ を満たす Ax の近似ベクトル y と誤差ベクトル r $(y, r \in \mathbb{F}^n)$ を求めるアルゴリズム．

function $[y, r] = \mathtt{MVErr}(A, x)$
$[m, n] = \mathtt{size}(A)$ % m 行 n 列
for $i = 1 : m$
  $w = A(i, :)$
  $[y_i, r_i] = \mathtt{Dot2Err}(w^T, x)$
end

□

4 提案手法による精度保証

本章では，Java において連立一次方程式の数値解の精度保証をするために，丸めモードの制御なしで行う高速かつ高精度な精度保証法を提案する．

4.1 従来手法

Ogita らは，丸めモードの変更なしで連立一次方程式 $Ax = b$ の数値解の精度保証をする手法を提案している [10]．この手法は浮動小数点演算の事前誤差評価を用いた手法で，

誤差の高速な見積もりができることが示されている．式 (3) から

$$\|RA - I\|_\infty \leq \alpha, \qquad \|R(Ax - b)\|_\infty \leq \beta$$

のような α, β が得られれば

$$\|\tilde{x} - A^{-1}b\|_\infty \leq \mathrm{fl}\left(\frac{\beta/(1-\alpha)}{1-3\mathbf{u}}\right) \tag{13}$$

によって精度保証をすることができることがわかるので，以下では α, β の計算方法について考える．

まず，α を求めるアルゴリズムを以下に記す．

Algorithm 6 (Ogita-Rump-Oishi [10]) $A \in \mathbb{F}^{n\times n}$, R を A の近似逆行列としたとき，$\|RA - I\|_\infty$ の上限 α を求めるアルゴリズム．

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

□

Algorithm 6 による α の評価は，基本的に

$$\begin{aligned} \alpha &\approx \mathrm{fl}(\|RA - I\|_\infty) + n\mathbf{u} \cdot \mathrm{cond}_\infty(A) \\ &\lesssim \mathrm{fl}(\|RA - I\|_\infty) + n^{1.5}\mathbf{u} \cdot \mathrm{cond}_2(A) \end{aligned} \tag{14}$$

である．ただし，$\mathrm{cond}_p(A)$ は行列の条件数であり

$$\mathrm{cond}_p(A) := \|A\|_p \cdot \|A\|_p^{-1}, \quad p = 2, \infty \tag{15}$$

によって定義される．一般的に，条件数が大きいほど問題の性質は悪条件となる．よって，このアルゴリズムは

$$\mathrm{cond}_2(A) \lesssim (n^{1.5}\mathbf{u})^{-1} \tag{16}$$

であるような連立一次方程式 $Ax = b$ に適用可能である．たとえば，IEEE 754 の倍精度であれば，$n = 1000$ のとき $\mathrm{cond}_2(A)$ が $(1000^{1.5} \cdot 2^{-53})^{-1} \approx 2.8 \times 10^{11}$ 程度までの問題が，Algorithm 6 を適用可能な限度であることが推定できる．これについては 5 章の数値実験においても考察する．

次に，β を求めるアルゴリズムを以下に記す．

Algorithm 7 (Ogita-Rump-Oishi [10]) $A \in \mathbb{F}^{n\times n}$, $b \in \mathbb{F}^n$, $\tilde{x}$ を $Ax = b$ の近似解，R を A の近似逆行列としたとき，$\|R(A\tilde{x} - b)\|_\infty$ の上限 β_1 を求めるアルゴリズム．

$$\begin{array}{l}\text{function}\ \ \beta_1 = \texttt{Beta.Std}(A, b, \tilde{x}, R)\\ \gamma = \mathrm{fl}\,(((n+1)\mathbf{u})/(1-(2n-4)\mathbf{u}))\\ r = \mathrm{fl}(|A\tilde{x}-b| + \gamma(|A||\tilde{x}|+|b|))\\ \beta_1 = \mathrm{fl}\,((\|\,|R|\,r\|_\infty)/(1-(n+3)\mathbf{u}))\end{array}$$

□

Algorithm 7による β の評価に関して，$\tilde{x}$ が良い近似解である場合，残差 $A\tilde{x}-b$ は大きな桁落ちが起こりやすい計算となり，3行目

$$r = \mathrm{fl}(|A\tilde{x}-b| + \gamma(|A||\tilde{x}|+|b|))$$

の右辺第1項及び第2項は

$$\begin{aligned}\mathrm{fl}(|A\tilde{x}-b|) &\approx \mathbf{u}\,|b|\\ \mathrm{fl}(\gamma(|A||\tilde{x}|+|b|)) &\approx n\mathbf{u}(|A||\tilde{x}|+|b|)\end{aligned}$$

となる．つまり，n が大きい場合には，本来は誤差項であるはずの第2項が支配的となり，十分な精度が得られない可能性がある．これは，事前誤差評価を用いているために避けられないものである．また，4行目

$$\beta_1 = \mathrm{fl}\,((\|\,|R|\,r\|_\infty)/(1-(n+2)\mathbf{u}))$$

では，$\|R(A\tilde{x}-b)\|_\infty$ の評価を $\||R|r\|_\infty$ として計算しているため，結果が過大評価となる場合がある．

4.2 提案手法

本節では高精度内積計算アルゴリズムを用いて従来手法における Algorithm 7 の β_1 の見積もりを改善し，過大評価をできるだけ抑制する手法を提案する．

まず，残差 $A\tilde{x}-b$ の評価について考える．$A, \hat{A}, \tilde{x}, \hat{x}$ をそれぞれ

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}, \quad \hat{A} = (A|b) = \begin{pmatrix} a_{11} & \cdots & a_{1n} & b_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n1} & \cdots & a_{nn} & b_n \end{pmatrix}$$

及び

$$\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)^T, \quad \hat{x} = (\tilde{x}_1, \ldots, \tilde{x}_n, -1)^T$$

とすると

$$A\tilde{x} - b = \hat{A}\hat{x} \tag{17}$$

となる．次に，$\hat{A}\hat{x}$ に対して Algorithm 5 (`MVErr`) を適用し

$$r_{\mathrm{mid}} - r_{\mathrm{rad}} \le A\tilde{x} - b \le r_{\mathrm{mid}} + r_{\mathrm{rad}} \tag{18}$$

を満たす $A\tilde{x}-b$ の近似ベクトル r_{mid} と誤差ベクトル r_{rad} $(r_{\mathrm{mid}}, r_{\mathrm{rad}} \in \mathbb{F}^n)$ を求める．これを，$A\tilde{x}-b$ を包み込む中心・半径型の区間ベクトル $\langle r_{\mathrm{mid}}, r_{\mathrm{rad}} \rangle$ と表現し，A の近似逆行列 $R \in \mathbb{F}^{n\times n}$ と区間ベクトル $\langle r_{\mathrm{mid}}, r_{\mathrm{rad}} \rangle$ の積をとると

$$t_1 - t_2 \le R(A\tilde{x}-b) \le t_1 + t_2 \tag{19}$$

となることがわかる (たとえば，文献 [11] を参照)．ただし

$$t_1 := R\,r_{\mathrm{mid}}, \qquad t_2 := |R|\,r_{\mathrm{rad}}$$

である．これにより $R(A\tilde{x}-b)$ の絶対値について

$$|R(A\tilde{x}-b)| \le |R\,r_{\mathrm{mid}}| + |R|\,r_{\mathrm{rad}} \tag{20}$$

が成立する．式 (8) の評価を行列とベクトルの積に拡張して $|R\,r_{\mathrm{mid}}|$ に適用すると

$$|R\,r_{\mathrm{mid}}| \le s_1 + s_2 \tag{21}$$

を得る．ただし

$$s_1 := |\mathrm{fl}(R\,r_{\mathrm{mid}})| = \mathrm{fl}(|R\,r_{\mathrm{mid}}|), \qquad s_2 := \mathrm{fl}(\tilde{\gamma}_{n,2n+2}(|R|\,|r_{\mathrm{mid}}|))$$

である．次に式 (7) の評価を行列とベクトルの積に拡張して $|R|\,r_{\mathrm{rad}}$ に適用すると

$$|R|\,r_{\mathrm{rad}} \le \mathrm{fl}\left(\frac{|R|\,r_{\mathrm{rad}}}{1-(n+1)\mathbf{u}}\right) =: s_3 \tag{22}$$

となる．よって，式 (21)，(22) を式 (20) に代入することにより

$$|R(A\tilde{x}-b)| \le s_1 + s_2 + s_3 \tag{23}$$

となる．ここで，式 (4) はベクトルに対しても要素ごとに成り立つため

$$|R(A\tilde{x}-b)| \le (1+\mathbf{u})^2\mathrm{fl}\,(s_1 + (s_2+s_3)) \tag{24}$$

となる．最後に，式 (5), (6) より

$$\begin{aligned} \|R(A\tilde{x}-b)\|_\infty &\le \left\|(1+\mathbf{u})^2\mathrm{fl}\,(s_1+(s_2+s_3))\right\|_\infty \\ &\le \mathrm{fl}\left(\frac{\|s_1+(s_2+s_3)\|_\infty}{1-3\mathbf{u}}\right) =: \beta_2 \end{aligned} \tag{25}$$

が成り立つ．

ここで，この β_2 の誤差評価方法について考えると

$$\|s_1+s_2+s_3\| \approx \|R\,r_{\mathrm{mid}}\| + n\mathbf{u}\||R|\,|r_{\mathrm{mid}}|\| + \||R|\,r_{\mathrm{rad}}\| \tag{26}$$

であるから

$$\|r_{\mathrm{rad}}\| \approx \mathbf{u}\|r_{\mathrm{mid}}\| \tag{27}$$

であれば，式 (26) の第 3 項はほとんど無視できて

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

となる．通常

$$\|R r_{\mathrm{mid}}\| \gg n\mathbf{u}\||R|\,|r_{\mathrm{mid}}|\|$$

であることが期待できるから，結局

$$\|s_1 + s_2 + s_3\| \approx \|R r_{\mathrm{mid}}\|$$

となる．したがって，Algorithm 5 を用いて式 (27) を満たすように r_{mid} 及び r_{rad} を計算すれば，β_2 がほとんど過大評価のない誤差の上限となることが期待できる．

以上の議論により，以下の定理を得る．

Theorem 1 $A \in \mathbb{F}^{n\times n}$，$b \in \mathbb{F}^n$，連立一次方程式 $Ax = b$ の近似解を $\tilde{x} \in \mathbb{F}^n$，$R$ を A の近似逆行列とする．また残差 $A\tilde{x} - b$ の近似ベクトル r_{mid}，誤差ベクトル r_{rad} は

$$r_{\mathrm{mid}} - r_{\mathrm{rad}} \le A\tilde{x} - b \le r_{\mathrm{mid}} + r_{\mathrm{rad}}$$

を満たすものとする．このとき

$$\|R(A\tilde{x} - b)\|_\infty \le \mathrm{fl}\left(\frac{\|s_1 + (s_2 + s_3)\|_\infty}{1 - 3\mathbf{u}}\right) \tag{28}$$

が成立する．ただし

$$s_1 := \mathrm{fl}(|R r_{\mathrm{mid}}|), \quad s_2 := \mathrm{fl}(\tilde{\gamma}_{n,2n+2}(|R|\,|r_{\mathrm{mid}}|)), \quad s_3 := \mathrm{fl}\left(\frac{|R|\,r_{\mathrm{rad}}}{1 - (n+1)\mathbf{u}}\right)$$

である． □

Theorem 1 及び Algorithm 5 を用いて $\|R(A\tilde{x} - b)\|_\infty \le \beta_2$ を満たす上限 β_2 を計算するアルゴリズムを以下に示す．

Algorithm 8 $A \in \mathbb{F}^{n\times n}$，$b \in \mathbb{F}^n$，連立一次方程式 $Ax = b$ の近似解を $\tilde{x}$，R を A の近似逆行列としたとき，$\|R(A\tilde{x} - b)\|_\infty \le \beta_2$ を満たす 上限 β_2 を計算するアルゴリズム．

function $\beta_2 = \mathtt{Beta.New}(A, \tilde{x}, b, R)$
$[r_{\mathrm{mid}}, r_{\mathrm{rad}}] = \mathtt{MVErr}([A, b], [\tilde{x}; -1])$ % $\hat{A} = (A|b)$, $\hat{x} = (\tilde{x}_1, \ldots, \tilde{x}_n, -1)$
$s_1 = \mathrm{fl}(|R r_{\mathrm{mid}}|)$
$s_2 = \mathrm{fl}(\tilde{\gamma}_{n,2n+2}(|R|\,|r_{\mathrm{mid}}|))$
$s_3 = \mathrm{fl}\,((|R|\,r_{\mathrm{rad}})/(1 - (n+1)\mathbf{u}))$
$\beta_2 = \mathrm{fl}\,((s_1 + (s_2 + s_3))/(1 - 3\mathbf{u}))$

□

α の評価については従来手法と同様に Algorithm 6 を用い，$\beta = \beta_2$ として最終的な数値解の精度を式 (13) により評価する．

5 数値実験

計算機上で連立一次方程式を解いたり行列計算をする場合，高速性や信頼性の点から数値計算ライブラリを用いることが多い．Java で利用可能な数値計算ライブラリとしては，たとえば以下のようなものがある．

- JLAPACK (B. Blount [2])
- JAMA (MathWorks, NIST [8])
- JAMPACK (G.W. Stewart [12])

表 2 は上記のライブラリを用いて行列乗算に要した計算時間の比較であり，表 3 は連立一次方程式を解いたときの計算時間を比較した結果である．数値実験に用いた計算機の CPU は Pentium IV 1.7GHz であり，Java のバージョンは J2SDK1.4.2_06 である．

表 2: 行列乗算に対する計算時間 (秒) の比較

n	JLAPACK	JAMA	JAMPACK
100	0.02	0.02	0.07
500	1.07	2.32	5.82
1000	8.44	17.9	47.7
2000	66.6	142	383

表 3: 連立一次方程式の求解に対する計算時間 (秒) の比較

n	JLAPACK	JAMA	JAMPACK
100	0.06	0.02	0.31
500	0.81	0.88	2.09
1000	5.71	6.47	12.7
2000	43.1	49.0	94.1

JAMPACK は行列の要素をすべて複素数として計算するため，実数のみを扱う場合には他のライブラリに比べて遅くなる．JLAPACK では行列乗算や連立一次方程式の求解を高速に行えるが，逆行列を求める関数などが現段階ではサポートされていない．それに対し，JAMA は JAMPACK より高速であり，逆行列を求める関数やノルムの計算なども整備されている．よって，以下では JAMA を用いる．

次に，以下に示す 4 つの手法による精度保証結果及び計算時間の比較を行う．

手法 A Oishi-Rump の手法 [11, Algorithm 3.1, 3.2]

手法 B Ogita らの手法 [10] (Algorithm 6, 7)

手法 C 手法 A に Algorithm 5 を適用

手法 D 提案手法 (Algorithm 6, 8)

手法 A, C は有向丸めを用いた精度保証法であるため，JNI を利用して実装した．手法 B, D は有向丸めを用いないため，Java のポータビリティを活かせる精度保証法である．連立一次方程式の近似解の計算，近似逆行列及び行列乗算には，前節で紹介した Java の線形計算パッケージである JAMA を使用した．数値実験に用いる計算機の CPU は Pentium IV 1.7GHz であり，Java のバージョンは J2SDK1.4.2_06 である．また実験ではすべて倍精度を用いた．

まず，A を各要素が $[-1,1]$ の一様乱数である $n \times n$ 行列とし，$b = \mathrm{fl}(A \cdot e)$ とする．ただし，$e = (1,\ldots,1)^T$ である．このとき，各手法による精度保証の結果を表 4 に示す．また，各手法に要した計算時間を表 5 に示す．参考のために，表 5 には LU 分解及び近似逆行列の計算に要した計算時間 (LU, 逆行列) も載せる．

表 4: 各手法による精度保証結果の比較

n	手法 A	手法 B	手法 C	手法 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

表 5: 各手法による計算時間 (秒) の比較

n	LU	逆行列	手法 A	手法 B	手法 C	手法 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

表 4 から，手法 C,D は手法 A,B と比較して誤差の過大評価を抑えることができ，手法 D は手法 C と同等な評価結果を得られたことがわかる．また，表 5 から，手法 D は手法 A, C より高速であり，次元数 n が大きい場合には手法 B とほぼ同等な計算時間で実行できることがわかる．

次に，A の条件数 $\mathrm{cond}_2(A)$ を 10^2 から 10^{12} まで変化させた場合について考える．A の各成分は $[-1,1]$ の乱数であり，$b = \mathrm{fl}(A \cdot e)$ とする．また，あらかじめ残差反復を用いて十分に高精度な近似解 $\tilde{x}$ が得られているものとする．このときの各手法による精度保証結果が表 6 である．

表 6 から，手法 A, B では解の精度が良くても条件数 $\mathrm{cond}_2(A)$ が大きいときには精度保証結果はかなり過大評価となる．これに対し，手法 C, D では条件数が悪い場合でも倍精度のほぼ限界まで精度保証ができている．ただし，手法 B, D では A の正則性の検証に

表 6: 条件数を変えて高精度な解を与えた場合における精度保証結果の比較 ($n = 1000$)

$\mathrm{cond}_2(A)$	手法 A	手法 B	手法 C	手法 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	–

–は精度保証失敗 ($\alpha \geq 1$)

おいて $\|RA - I\|_\infty$ を過大評価してしまうために，手法 A, C に比べて精度保証が成功する範囲が狭くなっている．これは，式 (16) で示したように，Algorithm 6 のような事前誤差評価を用いた場合には避けられないものである．

6 むすび

本報告では，連立一次方程式に対する Java 向けの精度保証法を提案した．提案手法を用いると計算量の少ない部分に高精度なアルゴリズム (Algorithm 5) を用いることにより，わずかな手間を追加するだけで，Ogita らの手法よりも大幅に精度保証の結果が改善され，Oishi-Rump の有向丸めを用いた手法に Algorithm 5 を適用した手法と同等な結果を得られた．

提案手法により，Java のポータビリティを損なうことなく高速かつ高精度な精度保証が可能となった．提案手法を用いると，異なるプラットフォームにおいても同一の結果を得ることができる．

本報告では，使用する言語が Java である場合に適した精度保証法を提案した．本手法は，Java 以外の言語を用いた場合でも実装可能であるが，丸めモードの変更が可能な場合は，それを適宜用いたほうが精度保証のアルゴリズムは簡潔になる．したがって，計算する環境に適応した精度保証法を用いることが重要である．

参考文献

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

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

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

[4] S. Flynn-Hummel, V. Getov, F. Irigoin, Ch. Lengauer: High Performance Computing and Java, Report No. 284, Report of the Dagstuhl Seminar 00341, 2000.

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

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

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

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

[9] T. Ogita, S. M. Rump, S. Oishi: Accurate sum and dot product, SIAM J. Sci. Comput., to appear.

[10] T. Ogita, S. M. Rump, S. Oishi: Verified Solution of Linear Systems without Directed Rounding, *11th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics*, Fukuoka, Japan, 2004.

[11] S. Oishi, S. M. Rump: Fast Verification of Solutions of Matrix Equations, Numer. Math., **90**:4 (2002), 755–773.

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

[13] 内山 知実: Javaによる連続体力学の有限要素法, 森北出版, 2001.

[14] 首藤 一幸: 最適化の手引き, 月刊 JavaWorld, 2000年9月号, pp.58–75, IDG ジャパン, 2000.

[15] 峯村 吉泰: Javaによる流体・熱流動の数値シミュレーション, 森北出版, 2001.