数理解析研究所講究録
第2054巻 2017年 14-24

安定化手法を用いた有理標準形の導出
Computation of Frobenius Canonical Form with Stabilization Techniques

片山 彰之
Akiyuki Katayama *
東邦大学大学院 理学研究科
Gradiate School of Science, Toho University
白柳 潔
Kiyoshi Shirayanagi †
東邦大学 理学部
Faculty of Science, Toho University

Abstract

Frobenius canonical form is a canonical form of a square matrix over a field. An algorithm of computing it has instability. In this paper, we applied the stabilization techniques proposed by Sweedler and the second author to the algorithm, and obtained efficacy for matrices containing complicated elements.

1 はじめに

一般に, 計算機では浮動小数計算が用いられる. これは有限の固定された桁数で計算を行えるので, 計算機と相性が良い. ところが, 有限精度の浮動小数は誤差を含むことがある. これによって, あるアルゴリズムは, 計算途中で異なる分岐を辿ってしまい, 全く正しくない出力を返してしまう. この現象をアルゴリズムの不安定性と呼ぶ. 本稿では, 有理標準形 (Frobenius 標準形) を計算するアルゴリズムに対して, Moss Sweedler と本稿の第二著者によって提案された安定化手法 [4] を適用した. 有理標準形は, 体上の正方行列に対する標準形の一種であり, 四則演算のみで計算できる. 典型的な行列の標準形は Jordan 標準形であるが, これは一般に体の拡大を必要とする. また, 有理標準形は Jordan 標準形と同程度の情報を持つため, 計算機の観点から, 有理標準形を計算するべきである [3].

本稿は次のように構成される. まず, 第 2 節で安定化手法について簡単に復習をする. 第 3 節では, 有理標準形についての復習と, 本稿で用いるアルゴリズムについての説明をする. 第 4 節で, 計算機実験の結果について述べ, 第 5 節で本稿をまとめる.

*6515005k@nc.toho-u.ac.jp
†kiyoshi.shirayanagi@is.sci.toho-u.ac.jp

2 復習

この節では, 安定化手法について簡単な復習を行う. 簡単のため, 次の制限されたクラスのアルゴリズムだけを考える. より一般的な仮定については [4] を参照されたい.

不連続点 0 の代数的アルゴリズムとは, 次の条件を全て満たすアルゴリズムである:

- アルゴリズム中における任意のデータは, すべて多項式環 $\mathbf{C}[x_1, \cdots, x_\nu]$ の中で閉じている.
- アルゴリズム中の演算は $\mathbf{C}[x_1, \cdots, x_\nu]$ で閉じている.
- すべての述語は, 不連続点を持つならば 0 のみである.

最後の条件は詳しく述べるべきであろう. 述語は R から $\{true, false\}$ への写像と定義される. 但し, R は $\mathbf{C}[x_1, \cdots, x_\nu]$ の部分環である. 述語 p が**不連続点 0 の述語**であるとは, $p(\alpha)$ の真偽が $\alpha = 0$ かそうでないかで変わるときにいう. 次の例について考える. $R := \mathbf{R}$ とし, $pred$ を次のように定義する:

$$pred(x) := \begin{cases} true & (x = 4), \\ false & (otherwise). \end{cases}$$

これは不連続点 4 の述語である. しかし, これは明らかに次の述語 $pred'$ と同値である:

$$pred'(x) := \begin{cases} true & (x - 4 = 0), \\ false & (otherwise). \end{cases}$$

$pred'$ は不連続点 0 の述語である. 述語をこのように変換することで, 数式処理に使われるアルゴリズムの大半は, 不連続点 0 の代数的アルゴリズムに変換することができる.

不連続点 0 の代数的アルゴリズム $\mathcal{A}$ に対して, **区間化**, **区間演算**, **ゼロ書き換え**を施したものを $\mathcal{A}$ の**安定化されたアルゴリズム**と呼び, Stab($\mathcal{A}$) と表す. 区間化, 区間演算, ゼロ書き換えについては後の小節で詳しく述べることにし, ここでは主定理を掲載する. この定理の証明及び, より一般的な仮定については [4] を参照せよ.

定理 1 (Shirayanagi-Sweedler, 1995, [4])
$\mathcal{A}$ を不連続点 0 の代数的アルゴリズム, I を $\mathcal{A}(I)$ が正常終了するような入力, $\{I_j\}_j$ を I の近似列 [1] とする. このとき, 安定化されたアルゴリズム Stab($\mathcal{A}$) に対して次が成り立つ. ある自然数 N が存在し, $k \geq N$ ならば,

1. Stab($\mathcal{A}$)(I_k) は正常終了し,
2. Stab($\mathcal{A}$)(I_k) は $\mathcal{A}(I)$ に収束する.

一般に, $\mathcal{A}(I_k)$ は**必ずしも** $\mathcal{A}(I)$ **へ収束しない**ことに注意せよ. これは, 2 節の冒頭で説明したような述語を含むアルゴリズムを考えれば明白であろう.

---

[1] 簡明のため, 本稿では近似列を次のように定義する. 入力 I に対して, I_j を I のすべての要素を小数 j 桁で近似したものと定義し, 列 $\{I_j\}_j$ を I の**近似列**と呼ぶ. より一般的な仮定は [4] を参照されたい.

2.1 区間化

区間化とは, アルゴリズムの入力を全て対応する区間に変換することを指す.

まず便宜上, 区間について定義する. $a,b,c,d \in \mathbf{R}$ に対して, $b,d \geq 0$ を満たすとき, $[[a,b][c,d]]$ を**区間**と呼ぶ. $\alpha = a+bi \in \mathbf{C}$ に対して, 次を満たす区間を α **の区間**と呼び, (α) で表す:

$$[[a',\epsilon],[b',\delta] \ \ where \ \ a' \in U(a;\epsilon) \ \ and \ \ b' \in U(b;\delta).$$

但し, $U(x;y) := \{z \in \mathbf{R} \mid |x-z| \leq y\}$ とする. 典型的には, $\epsilon = \delta = 10^{-j}$ とする. このとき, (α) は**精度** j **の区間**と呼ぶ.

I を $\mathcal{A}$ の入力とする. このとき, I のすべての成分をある精度の区間にすることを**区間化**と呼ぶ. 本稿で用いる複素数に対する区間は通常のものではない. この区間は, 実部と虚部とを分けることによって, 通常の区間よりも多くの情報を持つことができる. 則ち, 実部と虚部とを分けてゼロ書き換えを行える. ゼロ書き換えは後の小節で説明する. また, 複素数に対する通常の区間は [1] を参照されたい.

2.2 区間演算

区間化の後, 入力がすべて区間になっているため, アルゴリズムを進める際, 元の演算に対応するような区間同士の演算が必要となる. これが区間演算である. 今, $arith$ を $\mathbf{C}$ 上の二項演算とする. 複素数 $\alpha_1 = a_1+b_1i$, $\alpha_2 = a_1+b_2i$ に対して, 対応する区間をそれぞれ $(\alpha_1) = [[a_1',\epsilon_1],[b_1',\delta_1]]$, $(\alpha_2) = [[a_2',\epsilon_2],[b_2',\delta_2]]$ とする. このとき, $arith$ **に対する二項区間演算** $itarith$ を以下のように定義する:

$$\begin{aligned} &itarith((\alpha),(\beta)) \\ &:= [[f_{\Re}(a_1',b_1',a_2',b_2'), f_\epsilon(a_1',b_1',\epsilon_1,\delta_1,a_2',b_2',\epsilon_2,\delta_2)],[f_{\Im}(a_1',b_1',a_2',b_2'), f_\delta(a_1',b_1',\epsilon_1,\delta_1,a_2',b_2',\epsilon_2,\delta_2)]]. \end{aligned}$$

但し, $f_{\Re} \in U(\Re(arith(\alpha,\beta)); f_\epsilon)$, $f_{\Im} \in U(\Im(arith(\alpha,\beta)); f_\delta)$ を満たす. ここで, $\Re(\alpha)$ は α の実部, $\Im(\alpha)$ は α の虚部をそれぞれ表す. 簡明のため, $f_{\Re}, f_\epsilon, f_{\Im}, f_\delta$ の引数は省略した. $f_{\Re}, f_{\Im}$ は四変数関数で, a_1', b_1', a_2', b_2' によって定まり, f_ϵ, f_δ は八変数関数で, $a_1', b_1', \epsilon_1, \delta_1, a_2', b_2', \epsilon_2, \delta_2$ によって定まる.

2.3 ゼロ書き換え

安定化手法はゼロ書き換えという処理を持つ. これは, 他の区間法には無いもので, 安定化手法の大きな特徴である. ゼロ書き換えとは, 不連続点 0 の述語の直前に, 区間内に 0 が含まれれば, その区間を零区間に書き換えることである. ゼロ書き換えを説明する前に, 通常の述語から, 引数に区間を持つような述語へ拡張する方法について述べる. $\alpha = a+bi$ を複素数, α の区間を $(\alpha) = [[a',\epsilon],[b',\delta]]$, $\mathbf{C}$ 上の一引数述語を $predicate$ とする. このとき, 複素数区間上の述語 $it\_predicate$ を次のように定義する.

$$it\_predicate((\alpha)) := predicate(a'+b'i).$$

このようにすれば, 区間引数の述語を定義することができる. 引数が複数個存在する述語も同様に定義できる. 次にゼロ書き換えを定義する. 不連続点 0 の述語を評価する前に, その述語の引数に対して, 次の写像 ZF を施すことを**ゼロ書き換えを施す**という. 複素数 $\alpha = a+bi$ に対して, その区間を $(\alpha) = [[a',\epsilon],[b',\delta]]$ と表す. このとき,

$$ZF((\alpha)) := \begin{cases} [[a',\epsilon],[b',\delta]] & (|a'| > \epsilon \ \ and \ \ |b'| > \delta), \\ [[a',\epsilon],[0,0]] & (|a'| > \epsilon \ \ and \ \ |b'| \leq \delta), \\ [[0,0],[b',\delta]] & (|a'| \leq \epsilon \ \ and \ \ |b'| > \delta), \\ [[0,0][0,0]] & (|a'| \leq \epsilon \ \ and \ \ |b'| \leq \delta). \end{cases}$$

3 有理標準形

この節では, 有理標準形乃至 Frobenius 標準形について復習する. 有理標準形は一般の体上で定義できるが, 本稿では **C** 上で議論を進める. 尚, 以下の定義や定理は, 一般の体上でも成り立つ. 有理標準形を定義する前に数学的な準備を行う.

定義 2 (companion 行列と最小多項式)
n 次正方行列 C が **companion** であるとは, C が次の構造を持つときである.

$$C = \begin{bmatrix} 0 & 0 & \cdots & 0 & a_0 \\ 1 & 0 & \cdots & 0 & a_1 \\ 0 & 1 & \cdots & 0 & a_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & a_{n-1} \end{bmatrix}.$$

但し, $a_0, \cdots, a_{n-1}$ はいずれも複素数である. また, C **に付随する最小多項式** $\phi(t)$ を

$$\phi(t) := t^n - a_{n-1}t^{n-1} - \cdots - a_1 t - a_0$$

とする.

次の定理が有理標準形について述べている. 証明や詳細な情報については [2] を参照されたい.

定理 3 (有理標準形の存在と一意性)
A を **C** 上の n 次正方行列とする. このとき, 次を満たす C が一意的に存在する.

$$C = PAP^{-1}, \quad C = \bigoplus_{i=1}^{r} C_i, \quad \phi_{i+1}(t) \mid \phi_i(t) \ \ for \ \ 1 \le i \le r-1.$$

但し, r はある自然数, P は正則行列, C_i は *companion* 行列, $\phi_i(t)$ は C_i に付随する最小多項式である. C を A の**有理標準形**乃至 **Frobenius 標準形**と呼ぶ.

3.1 アルゴリズム

有理標準形を計算するアルゴリズムはいくつか存在する [3], [5]. しかし, これらは次の基本的なアルゴリズムがベースとなっている. 従って, 私たちは基本的なアルゴリズムに対して安定化手法を適用する. アルゴリズムは以下のステップで実行される. 尚, このアルゴリズムは [2] からの引用である.

アルゴリズム 1
Input: n 次正方行列 A
Output: A の *Frobenius* 標準形 C

1. $flag := true$, $l := 1$, $m := 1$, $lst := []$, $E := A$
2. E の第 l 列中で, e_{il}(E の第 (i,l) 成分) が非零になるような i を第 $l+1$ 行から第 n 行の中から選ぶ. **if** そのような i が見つからない **then go to 5.**.
3. E の第 $l+1$ 行と第 i 行を入れ替え, 同時に第 $l+1$ 列と第 i 列を入れ替える. その後 $\alpha := e_{l+1,l}$ とし,E の第 $l+1$ 行を α で割り, 同時に第 $l+1$ 列を α 倍する.

4. **for** $i = 1, 2, \cdots, l, l+2, \cdots, n$ **do**

$\alpha := e_{il}$ とし, E の第 i 列から第 $l+1$ 列を α したものを引き, 同時に第 $l+1$ 列に第 i 列を α 倍したものを加える.

その後, $l := l+1$, $m := m+1$ とし, **go to 2.**.

5. **if** *lst* の成分の和が n **then return** E.
else if *flag* $=$ *true* **then**

m を *lst* に加え, *flag* $:=$ *false*, $l := l+1$, $m := 1$ とし, **go to 2.**.

else

I. E の第 l 列中で, e_{il} が非零になるような i を第 1 行から第 $l-m$ 行の中から選ぶ.
if そのような i が見つからない **then go to 6.**.

II. 今, $lst = [m_1, \cdots, m_r]$ と仮定する. また, $a_0 := 0$, $a_i := m_1 + \cdots + m_i$ とする. このとき, $a_{p-1} < i \leq a_p$ を満たす p を見つけられる.
if $m_p \leq m$ **then**

E の第 $a_{p-1}+1$ 行と第 a_r+1 行を入れ替え, 同時に第 $a_{p-1}+1$ 列と第 a_r+1 列を入れ替える. その後, $m_p, m_{p+1}, \cdots, m_r$ を *lst* から取り除き, $l := m_0 + m_1 + \cdots + m_{p-1} + 1$ [2], $m := 1$, もし $lst = []$ ならば *flag* $:=$ *true* とし, **go to 2.**.

III. **for** $j = 1, 2, \cdots, m_p - m$ **do**

$\alpha := e_{a_p - j + 1, l}$ とする.
for $k = 1, 2, \cdots, m$ **do**

E の第 $l-k+1$ 列から第 $a_p - j - k + 1$ 列の α 倍を引き, 同時に第 $a_p - j - k + 1$ 行に第 $l-k+1$ 行の α 倍を加える.

E の第 l 列中で, e_{ql} が非零になるような q を第 $a_p - r + 1$ 行から第 a_p 行の中から選ぶ.
if そのような q が見つからない **then**

m を *lst* に加え, $l := l+1$, $m := 1$ とし, **go to 2.**.

IV. E の第 $a_{p-1}+1$ 行と第 a_r+1 行を入れ替え, 同時に第 $a_{p-1}+1$ 列と第 a_r+1 列を交換する. その後, $m_p, \cdots, m_r$ を *lst* から取り除き, $l := m_0 + \cdots + m_{p-1} + 1$, $m := 1$, もし $lst = []$ ならば *flag* $:=$ *true* とし, **go to 2.**.

6. $lst = [m_1, \cdots, m_r]$ と仮定する. このとき, E は次の形をしている.

$$E = \begin{bmatrix} C_1 & & & & * & \cdots & * \\ & \ddots & & & * & \cdots & * \\ & & C_r & & * & \cdots & * \\ & & & C_{r+1} & * & \cdots & * \\ & & & & \vdots & & \vdots \\ & & & & * & \cdots & * \end{bmatrix},$$

[2] $m_0 := 0$ と定義する.

但し, C_i はコンパニオン行列, 空欄は適当な零行列, $*$ は適当な行列である. 今, $\phi_r(t), \phi_{r+1}(t)$ をそれぞれ, C_r, C_{r+1} の付随する最小多項式とする.
if $\phi_{r+1}(t) \mid \phi_r(t)$ **then**

m を lst に加え, $l := l+1$, $m := 1$ とし, **go to 2.**.

else

E の第 $m_1 + \cdots + m_r + 1$ 列を第 $m_1 + \cdot + m_{r-1} + 1$ 列に加え, 同時に, 第 $m_1 + \cdots + m_{r-1} + 1$ 行を第 $m_1 + \cdots + m_r + 1$ から引く. その後, $l := m_1 + \cdots + m_r + 1$, $m := 1$ とし, **go to 2.**.

このアルゴリズムは基本変形のみで計算されていることに注意せよ. また, 述語の不連続点も 0 に限るので, このアルゴリズムは不連続点 0 の代数的アルゴリズムである. 上記のアルゴリズムを `efrob` で表す.

4 計算機実験

私たちは, 不安定なアルゴリズムである `efrob` に対して, 計算機を用いて比較実験を行った. 詳細な設定は以下の通りである.

- 環境: iMac27(Late 2013), CPU: Intel Core i5(3.4GHz), 物理メモリ: 8GB.
- 言語: Maple18, Python3.5.
- 使われた関数:
  - Maple の組み込み関数 (`FrobeniusForm`),
  - 自作の厳密関数 (`efrob`),
  - 自作の近似関数 [3] (`ffrob`),
  - 自作の安定化された関数 1 [4] (`itfrob_R`),
  - 自作の安定化された関数 2 [5] (`itfrob_C`).
- 入力行列はすべて, 多重固有値を持つとする. [6]
- 浮動小数を用いる場合, 精度桁は 500 桁で行う.
- これらの関数に対して, 出力の正当性, CPU 時間, 総メモリ使用量の観点から比較実験.

---

[3] この関数は, まず入力を浮動小数近似してから `efrob` を走らす関数である.

[4] この関数は, 実数上の区間を用いた安定化された関数である. そのため, 虚数の含まれるような入力に対しては, 計算することができない.

[5] この関数は, 複素数上の区間を用いた安定化された関数である. 勿論, 入力がすべて実数成分でも走るが, 無駄な処理が多くなってしまう.

[6] これは, 入力行列の特性多項式が $(t-\alpha)^n$ の形を持つことを意味する. このとき, 行列の構造が不安定になる. この問題は, 行列の構造安定性問題として, 古典的に研究されている [2].

4.1 実数かつ代数的数

本小節では, 特性多項式が $(t-2^{1/17})^n$ となる行列に対して実験を行った. 今回は実験の精度を高めるために, 各次の行列に対して, 10 個の行列を用い, それらの出力の平均値をとった. 表 1, 2 が結果である. まず, 表中にある, *invalid* は出力があったが, それが正当でないことを表す. >1000 は 1000 秒経っても出力が返らなかったことを表す. *no_data* は, 何らかの原因で, 対応するデータが存在しないことを表す. *miss* は, 精度桁不足で処理が途中で止まってしまったことを表す. まず, 計算速度について考察する. 比較的小さい行列, 則ち, 3 次から 5 次の行列では, Maple の組み込み関数が最も速く, メモリ効率も良い. しかし, それ以上の大きさの行列になると, 急激に遅くなっていることがわかる. 従って, Maple の組み込み関数は, 比較的小さい行列に対して最適化されていることがわかる. 自作の厳密関数は行列が大きくなるにつれて時間がかかっている. そして, 13 次以降から, 急激に遅くなっている. これは, 計算量の問題もあるが, 内部で係数膨張が起きていると示唆する. 安定化手法を用いた関数はいずれも高速に計算することができている. 只, 実数上では, 実数に特化した方が効率的であることがわかる. 素朴な浮動小数近似は, 高速に計算できていたが, 今回の実験では不当な出力を返していた. 次に, メモリの使用量であるが, これらは計算速度とほぼ同様の結果になっている.

表 1: CPU 時間 (sec), $\mathbf{Q}(2^{1/17})$

行列の大きさ	FrobeniusForm	efrob	ffrob	itfrob_R	itfrob_C
3	0.0056	0.7169	*invalid*	0.129	0.1032
4	0.0518	2.968	*invalid*	0.1326	0.3342
5	0.2806	7.7452	*invalid*	0.1478	0.7916
6	6.1065	14.962	*invalid*	0.3602	1.849
7	20.43	32.839	*invalid*	0.7225	3.6246
8	175.79	49.131	*invalid*	1.0758	6.9775
9	238.01	77.511	*invalid*	1.8777	11.365
10	>1000	123.63	*invalid*	3.31	18.691
11	>1000	137.86	*invalid*	6.404	29.976
12	>1000	186.01	*invalid*	8.4946	54.06
13	>1000	254.94	*invalid*	15.119	*miss*
14	>1000	640.95	*invalid*	25.181	*miss*
15	>1000	878.23	*invalid*	31.720	*miss*
16	>1000	>1000	*invalid*	50.417	*miss*
17	>1000	>1000	*invalid*	71.697	*miss*
18	>1000	>1000	*invalid*	*miss*	*miss*
19	>1000	>1000	*invalid*	*miss*	*miss*
20	>1000	>1000	*invalid*	*miss*	*miss*

表.2: 総メモリ使用量 (GiB), $\mathbf{Q}(2^{1/17})$

行列の大きさ	FrobeniusForm	efrob	ffrob	itfrob_R	itfrob_C
3	0.00063169	0.09412	*invalid*	0.0039623	0.019643
4	0.0091878	0.37536	*invalid*	0.012171	0.070373
5	0.052897	0.91178	*invalid*	0.034206	0.16928
6	0.60191	1.559	*invalid*	0.066279	0.41059
7	2.7462	2.8354	*invalid*	0.13155	0.83222
8	12.416	4.3809	*invalid*	0.22608	1.5578
9	17.641	6.173	*invalid*	0.38845	2.5366
10	*no_data*	8.7631	*invalid*	0.62661	4.2236
11	*no_data*	12.605	*invalid*	1.1774	6.4194
12	*no_data*	27.228	*invalid*	1.4828	10.657
13	*no_data*	33.316	*invalid*	2.4587	*miss*
14	*no_data*	63.979	*invalid*	3.7983	*miss*
15	*no_data*	97.628	*invalid*	4.5802	*miss*
16	*no_data*	*no_data*	*invalid*	6.2730	*miss*
17	*no_data*	*no_data*	*invalid*	7.8156	*miss*
18	*no_data*	*no_data*	*invalid*	*miss*	*miss*
19	*no_data*	*no_data*	*invalid*	*miss*	*miss*
20	*no_data*	*no_data*	*invalid*	*miss*	*miss*

4.2 虚数かつ代数的数

本小節では, 特性多項式が $(t-13^{1/7}i)^n$ となる行列に対して実験を行った. 今回の実験も精度を高めるために, 各次の行列に対して, 10 個の行列を用い, それらの出力の平均値をとった. 表 3, 4 が結果である. まず, Maple の組み込み関数は, 3 次乃至 4 次の行列については非常に高速に計算することができた. しかし, 5 次から途端に遅くなり, 6 次以降は 1000 秒以上の時間がかかった. 素朴な近似は 4.1 節と同様であった. また, 実数に特化して安定化された関数 (`itfrob_R`) は対応していないため, 実験できなかった. 安定化された関数は, いずれも自作の厳密関数よりも高速に計算することができ, メモリ使用量の観点からも優れていた. 9 次以降の行列は, いくつかは生成することができたが, 10 種類生成することができなかった. これは, 今回用いた計算機の性能が低かったためであると考えられる.

表 3: CPU 時間 (sec), $\mathbf{Q}(13^{1/7}, i)$

行列の大きさ	`FrobeniusForm`	`efrob`	`ffrob`	`itfrob_R`	`itfrob_C`
3	0.0109	0.4745	*invalid*	*no_data*	0.1414
4	0.525	1.2923	*invalid*	*no_data*	0.4901
5	133.35	3.029	*invalid*	*no_data*	1.1263
6	> 1000	6.6365	*invalid*	*no_data*	2.7333
7	> 1000	14.806	*invalid*	*no_data*	4.6702
8	> 1000	25.659	*invalid*	*no_data*	8.7635

4.3 Python

本節では, Python を用いた計算機実験の結果について報告する. Python はオブジェクト指向型スクリプト言語 (軽量言語) である. 元来, Python のようなスクリプト言語は, サーバーの管理などの短いプログラムを書くために使われてきたが, 近年では, 多様な場面で使われている. 例えば, クラウドストレージである「Dropbox」のアプリケーションや google 社の「YouTube」は Python を用いて書かれている.

さて, 本稿では, 数式処理システムではない [7] Python に対して, 有理標準形を計算するアルゴリズム

表 4: 総メモリ使用量 (sec), $\mathbf{Q}(13^{1/7}, i)$

行列の大きさ	`FrobeniusForm`	`efrob`	`ffrob`	`itfrob_R`	`itfrob_C`
3	0.0011437	0.064252	*invalid*	*no_data*	0.022905
4	0.075563	0.16989	*invalid*	*no_data*	0.086722
5	4.8871	0.40355	*invalid*	*no_data*	0.20545
6	*no_data*	0.90557	*invalid*	*no_data*	0.51969
7	*no_data*	2.0625	*invalid*	*no_data*	0.87205
8	*no_data*	3.1347	*invalid*	*no_data*	1.7336

[7] Python は SymPy というライブラリーを持っている. これを用いれば Python 上で記号処理を行えるが, 今回の研究では使用していない. 尚, 正確な浮動小数を使用するために, Decimal モジュールを用いたことに注意せよ.

efrob を実装し, 正しく計算することができるかを報告する.

まず, 次のような行列を計算した.

$$A := \begin{bmatrix} 3 & 3 & -4 & 4 & -3 \\ 0 & -6 & -6 & 0 & 2 \\ 2 & 4 & 1 & 2 & -3 \\ 1 & 7 & 4 & 2 & -3 \\ 7 & 4 & -4 & 5 & -7 \end{bmatrix}.$$

この行列の有理標準形は

$$\begin{bmatrix} 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 1 & -3 \end{bmatrix}.$$

となり, Python 上で計算できた. 次に,

$$\frac{1}{7}A = \begin{bmatrix} 3/7 & 3/7 & -4/7 & 4/7 & -3/7 \\ 0 & -6/7 & -6/7 & 0 & 2/7 \\ 2/7 & 4/7 & 1/7 & 2/7 & -3/7 \\ 1/7 & 1 & 4/7 & 2/7 & -3/7 \\ 1 & 4/7 & -4/7 & 5/7 & -1 \end{bmatrix}$$

を計算した. もちろん, これは, 行列 A のスカラー倍であるから

$$\begin{bmatrix} 0 & 0 & 2/7^3 & 0 & 0 \\ 1 & 0 & -1/7^2 & 0 & 0 \\ 0 & 1 & -4/7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2/7^2 \\ 0 & 0 & 0 & 1 & -3/7 \end{bmatrix}$$

となるはずだが, 実際には,

$$\begin{bmatrix} 0 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -11/7^2 \\ 0 & 1 & 0 & 0 & 1/7^2 \\ 0 & 0 & 1 & 0 & 8/7^4 \\ 0 & 0 & 0 & 1 & -4/7^5 \end{bmatrix}$$

の近似か, 無限ループに陥った. これは, 微小な誤差のために, アルゴリズムはゼロ判定を誤り, 結果として, まったく異なる構造の行列を返してしまったと考えられる. 一方, 安定化手法を用いたところ, 精度 50 桁以上で, 正しい行列の構造に収束した.

5 おわりに

本稿では有理標準形を求めるアルゴリズムに対して, 安定化手法を用いた計算の高速化を行った. その結果, 次が得られた.

- 安定化手法を用いた関数は, 計算速度, メモリ使用量ともに厳密な関数より優れていた.
- 素朴な浮動小数近似を用いた関数は, 計算速度, メモリ使用量ともに優れていたが, 今回のような多重固有値を持つ行列に対しては正当な出力を返さなかった.
- 実数上では, *itfrob_R* の方が *itfrob_C* より優れていた.
- 数式処理システムでなくとも, 安定化手法を用いれば, 十分大きい精度のもとで, 正当な出力を返す.

本稿では, 整数成分行列や有理成分行列は対象としていない. これは, これらの行列を計算するときには, モジュラー計算 [3] などの優れた計算法を用いればよいからである. 安定化手法は, これらの計算法では対処できないような行列で, かつ各成分が複雑であるようなものに対して威力を発揮できるはずである.

また, 今後の課題として次が挙げられる.

1. 本稿で用いられたアルゴリズムの精度桁問題の肯定的解決.
2. 区間有理標準形の正当性判定法.

1. について説明する. 主定理 (定理 1) によって, ある有限の N が存在し, それ以上の精度であれば, 安定化された関数の出力は常に正当であることが保証されている. しかし, そのような N を計算する手法は, まだ確立していない. このような N を求める問題を精度桁問題とよぶ. この問題が解決すれば, 安定化手法は非常に使いやすくなる. また, 2. については, 出力として表れた行列が, 正当なものかを判定する手法のことである. この判定法があれば, 不当な出力を検出することができる. もし, 不当ならば, それは精度桁が足りていないことを意味するので, より高い精度で計算することにより, いつかは正当な出力が返るのである. 今後は, これらの問題の解決に勤しみたい.

参 考 文 献

[1] G. Alefeld and J. Herzberger: *Introduction to Interval Computations, Computer Science and Applied Mathematics*, Académic Press, 1983.

[2] 韓 太舜, 伊理 正夫: ジョルダン標準形, UP 応用数学選書, **8**, 東京大学出版会, 1982.

[3] 森継 修一: 有理数行列の Frobenius 標準形のモジュラー計算法, 数理解析研究所講究録, **1335**, 2003, 33–40.

[4] K. Shirayanagi and M. Sweedler: A Theory of Stabilizing Algebraic Algorithms, *Technical Report 95-28, Mathematical Sciences Institute, Cornell University*, 1995, 92 pages.

[5] A. Storjohann, An $O(n^3)$ Algorithm for the Frobenius Normal Form, *ISSAC '98*, ACM, N.Y., 1998, 101–105.