信州大学工学部

学士論文

有理表現による標数 2 の
有限体演算の高速化に関する検討

指導教員　西新 幹彦 准教授

学科　電気電子工学科
学籍番号　12T2087F
氏名　宮本 健太

2017 年 2 月 18 日

目次

1　序論

1.1　研究の背景と目的

今日の社会では，スマートフォンやコンピュータなどの所有率は非常に高く，誰もが情報という分野と深い関わりを持っている．この情報という分野において通信を行う際に送信する情報源を符号化し受信する際に復号化が行われる．このとき，剰余計算を行うことで小数を取らず誤差が出ない利点を持つ有限体を用いることで符号化された情報源を誤りなく復号化することができる．

標数 2 の有限体において，正負の概念はなく加算と減算はどちらも排他的論理和として扱われる．また乗算は，有限体を多項式を用いて表し原始多項式による剰余計算を行うことで積を導くことができる．さらに除算では，既知数と未知数の乗算を行うことで導かれる連立方程式を解くことによって商を導くことができる．したがって，乗算に比べて除算は，連立方程式を解かなければならない分演算時間が長くなる．これに関して，従来研究 [1] では乗算と比較して除算の演算時間は 2 倍以上にもなることが報告されている．そこで本研究はこの除算の演算時間の増加に着目し，単一の有限体を分母分子それぞれに単一の有限体を持つ分数として表現して演算する方法を提案する．この方法を本研究では有理法と呼び，有理法による有限体の表現を有理表現と呼ぶ．有理表現された有限体の除算は，通常の有限体の加算と乗算のみで実現できるため，従来の方法よりも演算時間を短縮できることが期待される．本研究では，有理法の効果を検証するために，有理法を適用した場合と適用しない場合のそれぞれで逆行列を求める演算の演算時間を測定し比較する実験を行った．その結果，有限体のビット長が大きいほど有理法の効果が強いことが分かった．また，一連の演算の中で多くの除算が必要になる場合に有理法の効果が大きいことが確認された．

1.2　本論文の構成

以降，本論文は次のように構成されている．2 章では，有限体の乗算と除算の方法を説明する．3 章では本研究で提案する有理法について説明する．4 章では計算時間の計測実験の結果を見ながら，提案法の有効性を検証する．5 章では本論文のまとめを記す．

2　有限体

有限集合で且つ四則演算可能で分配法則を満たす代数系を有限体と呼ぶ．また，位数 q^n の有限体を位数 q の有限体の n 次拡大体と呼ぶ．ここで，次数が n の有限体の元 a は位数 q の

有限体を係数とする多項式を用いて，

$$f_a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$$

のように表すことができる．

なお，x は不定元である．また，本稿では，このような有限体を

$$\begin{pmatrix} a_{n-1} & a_{n-2} & \cdots & a_1 & a_0 \end{pmatrix}$$

のようにベクトルとして表現する．

本研究では標数 2 の有限体を使用する．位数 2^n の有限体を考えるとき，n をその有限体のビット長と呼ぶこととする．

2.1 有限体の乗算

本節では，有限体の乗算の理論的枠組みと数値計算の方法を説明する．位数 2^n の有限体の元 a, b に対し，それぞれを $n-1$ 次の多項式 $f_a(x), f_b(x)$ として表現し，$f_a(x)$ と $f_b(x)$ の積を求める．積は $2(n-1)$ 次多項式となるが扱っている有限体が n 次拡大体のため積も $n-1$ 次多項式で表現する必要がある．そこで，原始多項式を法とする剰余計算を行うことで n 次以上の項をそれより小さい次数の項に置き換えることができ，積を導くことができる．以下に位数 2^3 と 2^4 の場合の例を示す．

2.1.1 位数 2^3 の有限体の乗算

有限体の乗算について位数 2^3 の有限体を例に説明する．なお，原始多項式は x^3+x+1 を使用する．位数 2^3 の有限体の場合，x の次数は 2 次までしか扱えない．そこで，原始多項式を用いることで 3 次以上の項をそれより小さい次数の項にする．

[手順 1]　位数 2^3 であることから，$a_2x^2+a_1x+a_0$ と $b_2x^2+b_1x+b_0$ の 2 つの多項式を用意し，積 $c_2x^2+c_1x+c_0$ を導くために多項式同士を乗算すると，

$$\begin{aligned}
&(a_2x^2+a_1x+a_0)\times(b_2x^2+b_1x+b_0)\\
&= a_2b_2x^4+a_2b_1x^3+a_2b_0x^2+a_1b_2x^3+a_1b_1x^2+a_0b_2x^2+a_1b_0x+a_0b_1x+a_0b_0\\
&= c_2x^2+c_1x+c_0
\end{aligned}$$

となる．ここで，x の次数で整理すると，

$$\begin{aligned}
&a_2b_2x^4+(a_2b_1+a_1b_2)x^3+(a_2b_0+a_1b_1+a_0b_2)x^2+(a_1b_0+a_0b_1)x+a_0b_0\\
&= c_2x^2+c_1x+c_0
\end{aligned} \tag{2.1.1}$$

となる．

[手順２]　位数 2^3 のため x の次数は 2 次までしか扱えない．そこで，原始多項式を用いて 3 次以上の項をそれより小さい次数の項に置き換えると，

$$x^3 \to x+1 \text{ より} \qquad (a_2b_1+a_1b_2)x^3 \to (a_2b_1+a_1b_2)x+(a_2b_1+a_1b_2) \tag{2.1.2}$$

$$x^4 \to x(x+1)$$
$$= x^2+x \text{ より} \qquad a_2b_2x^4 \to a_2b_2x^2+a_2b_2x \tag{2.1.3}$$

となる．よって，式 (2.1.2)，式 (2.1.3) を式 (2.1.1) に代入し x の次数で整理すると，

$$(a_2b_0+a_1b_1+a_0b_2+a_2b_2)x^2+(a_1b_0+a_0b_1+a_2b_1+a_1b_2+a_2b_2)x+(a_0b_0+a_2b_1+a_1b_2)$$
$$= c_2x^2+c_1x+c_0 \tag{2.1.4}$$

となる．

[手順３]　式 (2.1.4) の両辺の x の係数同士で等式が成立するので，

$$\begin{cases} c_2 = a_2b_0+a_1b_1+a_0b_2+a_2b_2 \\ c_1 = a_1b_0+a_0b_1+a_2b_1+a_1b_2+a_2b_2 \\ c_0 = a_0b_0+a_2b_1+a_1b_2 \end{cases}$$

であり，任意の $(a_2\ a_1\ a_0),(b_2\ b_1\ b_0)$ に 0 または 1 を代入することで積 $(c_2\ c_1\ c_0)$ を導くことができる．

ここで，$(a_2\ a_1\ a_0)=(1\ 0\ 1),(b_2\ b_1\ b_0)=(0\ 1\ 1)$ の場合を具体的に計算すると，

$$\begin{cases} c_2 = 1\times1+0\times1+1\times0+1\times0=1 \\ c_1 = 0\times1+1\times1+1\times1+0\times0+1\times0=0 \\ c_0 = 1\times1+1\times1+0\times0=0 \end{cases}$$

となる．$(c_2\ c_1\ c_0)=(1\ 0\ 0)$ であり，積を導くことができた．2 進数である $(1\ 0\ 0)$ を 10 進数に変換すると 4 であり，同様にして任意の $(a_2\ a_1\ a_0),(b_2\ b_1\ b_0)$ の組み合わせによる積を 10 進数表記したものを表 1 に示す．

2.1.2　位数 2^4 の有限体の乗算

次に有限体の乗算について位数 2^4 の有限体を例に説明する．なお, 原始多項式は x^4+x+1 を使用する．位数 2^4 の有限体の場合．x の次数は 3 次までしか扱えない．そこで，原始多項式を用いることで 4 次以上の項をそれより小さい次数の項にする．

[手順１]　位数 2^4 であることから，$a_3x^3+a_2x^2+a_1x+a_0$ と $b_3x^3+b_2x^2+b_1x+b_0$ の２つの多項式を用意し，積 $c_3x^3+c_2x^2+c_1x+c_0$ を導くために多項式同士を乗算すると，

$$(a_3x^3+a_2x^2+a_1x+a_0)\times(b_3x^3+b_2x^2+b_1x^1+b_0)$$
$$= a_3b_3x^6+a_3b_2x^5+a_3b_1x4+a_3b_0x^3+a_2b_3x^5+a_2b_2x^4+a_2b_1x^3+a_2b_0x^2+a_1b_3x^4+a_1b_2x^3$$
$$+a_1b_1x^2+a_1b_0x+a_0b_3x^3a_0b_2x^2+a_0b_1x+a_0b_0 = c_3x^3+c_2x^2+c_1x+c_0$$

表 1　位数 2^3 の乗算 $(a \times b)$

b \ a	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7
2	0	2	4	6	3	1	7	5
3	0	3	6	5	7	4	1	2
4	0	4	3	7	6	2	5	1
5	0	5	1	4	2	7	3	6
6	0	6	7	1	5	3	2	4
7	0	7	5	2	1	6	4	3

となる．ここで，x の次数で整理すると，

$$a_3b_3x^6+(a_3b_2+a_2b_3)x^5+(a_3b_1+a_2b_2+a_1b_3)x^4+(a_3b_0+a_2b_1+a_1b_2+a_0b_3)x^3+(a_2b_0 +a_1b_1+a_0b_2)x^2+(a_1b_0+a_0b_1)x+a_0b_0=c_3x^3+c_2x^2+c_1x+c_0 \tag{2.1.5}$$

となる．

[手順２]　原始多項式を用いて 4 次以上の項をそれより小さい次数の項に置き換えると，

$$x^4 \to x+1 \text{ より} \quad (a_3b_1+a_2b_2+a_1b_3)x^3 \to (a_3b_1+a_2b_2+a_1b_3)x+(a_3b_1+a_2b_2+a_1b_3) \tag{2.1.6}$$

$$x^5 \to x(x+1)$$
$$=x^2+x \text{ より} \qquad (a_3b_2+a_2b_3)x^5 \to (a_3b_2+a_2b_3)x^2+(a_3b_2+a_2b_3)x \tag{2.1.7}$$

$$x^6 \to x(x^2+x)$$
$$=x^3+x^2 \text{より} \qquad a_3b_3x^6 \to a_3b_3x^3+a_3b_3x^2 \tag{2.1.8}$$

となる．よって，式 (2.1.6)〜式 (2.1.8) を式 (2.1.5) に代入し x の次数で整理すると，

$$(a_3b_0+a_2b_1+a_1b_2+a_0b_3+a_3b_3)x^3+(a_2b_0+a_1b_1+a_0b_2+a_3b_3+a_3b_2+a_2b_3)x^2+(a_1b_0 +a_0b_1+a_3b_2+a_2b_3+a_3b_1+a_2b_2+a_1b_3)x+(a_0b_0+a_3b_1+a_2b_2+a_1b_3) =c_3x^3+c_2x^2+c_1x+c_0 \tag{2.19}$$

となる．

[手順３]　式 (2.1.9) の両辺の x の係数同士で等式が成立するので，

$$\begin{cases} c_3 = a_3b_0 + a_2b_1 + a_1b_2 + a_0b_3 + a_3b_3 \\ c_2 = a_2b_0 + a_1b_1 + a_0b_2 + a_3b_3 + a_3b_2 + a_2b_3 \\ c_1 = a_1b_0 + a_0b_1a_3b_2 + a_2b_3 + a_3b_1 + a_2b_2 + a_1b_3 \\ c_0 = a_0b_0 + a_3b_1 + a_2b_2 + a_1b_3 \end{cases}$$

であり，任意の $(a_3\ a_2\ a_1\ a_0), (b_3\ b_2\ b_1\ b_0)$ に 0 または 1 を代入することで積 $(c_3\ c_2\ c_1\ c_0)$ を導くことができる．

2.2　有限体の除算

次に，有限体の除算の理論的枠組みと数値計算の方法を説明する．位数 2^n の有限体の元 a, b に対し，それぞれを $n-1$ 次の多項式 $f_a(x), f_b(x)$ として表現し，原始多項式を法として $f_a(x)$ を $f_b(x)$ で割った商 $f_c(x)$ を求める． ここで，既知数同士の除算は未知数である商を用いることで既知数と未知数の乗算に書き換えることができる．すなわち，$f_b(x)$ と $f_c(x)$ の積は $f_a(x)$ である．この乗算の積は 2.1 節と同様に，$2(n-1)$ 次多項式となるが扱っている有限体が n 次拡大体のため $n-1$ 次多項式で表現する必要がある．そこで，原始多項式を法とする剰余計算を行うことで n 次以上の項をそれより小さい次数の項に置き換えることができ，既知数と未知数の積を求めることができる．しかしこのままでは既知数と未知数が混在したままであり未知数を導くことができない．そこで，x の係数をベクトル表現を用いることで，既知数と未知数のそれぞれのベクトルの積に書き換える．さらに，既知数の逆行列を両辺に掛けることで未知数のベクトルを既知数同士のベクトルの積で表すことができ，商を導くことができる．以下に位数 2^3 と 2^4 の場合の例を示す．

2.2.1　位数 2^3 の有限体の除算

有限体の除算について位数 2^3 の有限体を例に説明する．なお，原始多項式は x^3+x+1 を使用する．位数 2^3 の有限体の場合，x の次数は 2 次までしか扱えない．そこで，原始多項式を用いることで 3 次以上の項をそれより小さい次数の項にする．

[手順１]　位数 2^3 であることから，$a_2x^2+a_1x+a_0$ と $b_2x^2+b_1x+b_0$ の 2 つの多項式を用意し，商 $c_2x^2+c_1x+c_0$ を導くために多項式同士で除算すると，

$$(a_2x^2+a_1x+a_0) \div (b_2x^2+b_1x+b_0) = c_2x^2+c_1x+c_0$$

であり，$b_2x^2+b_1x+b_0$ を右辺に移項すると，

$$\begin{aligned} &a_2x^2+a_1x+a_0 = (c_2x^2+c_1x+c_0) \times (b_2x^2+b_1x+b_0) \\ &= c_2b_2x^4+c_2b_1x^3+c_2b_0x^2+c_1b_2x^3+c_1b_1x^2+c_1b_0x+c_0b_2x^2+c_0b_1x+c_0b_0 \end{aligned}$$

のように未知数と既知数の乗算となる．さらに x の次数で整理すると，

$$a_2x^2+a_1x+a_0=c_2b_2x^4+(c_2b_1+c_1b_2)x^3+(c_2b_0+c_1b_1+c_0b_2)x^2+(c_1b_0+c_0b_1)x+c_0b_0 \tag{2.2.1}$$

となる．

[手順２] 原始多項式を用いて 3 次以上の項をそれより小さい次数の項に置き換えると，

$$x^3 \to x+1 \text{ より} \qquad (c_2b_1+c_1b_2)x^3 \to (c_2b_1+c_1c_2)x+(c_2b_1+c_1b_2) \tag{2.2.2}$$

$$x^4 \to x(x+1)$$
$$=x^2+x \text{ より} \qquad c_2b_2x^4 \to c_2b_2x^2+c_2b_2x \tag{2.2.3}$$

となる．よって，式 (2.2.2)，式 (2.2.3) を式 (2.2.1) に代入し x の次数で整理すると，

$$(c_2b_0+c_1b_1+c_0b_2+c_2b_2)x^2+(c_1b_0+c_0b_1+c_2b_1+c_1b_2+c_2b_2)x+(c_0b_0+c_2b_1+c_1b_2)$$
$$=a_2x^2+a_1x+a_0 \tag{2.2.4}$$

となる．

[手順３] 式 (2.2.4) の両辺の x の係数同士で等式が成立するので，

$$\begin{cases} a_2=c_2b_0+c_1b_1+c_0b_2+c_2b_2 \\ a_1=c_1b_0+c_0b_1+c_2b_1+c_1b_2+c_2b_2 \\ a_0=c_0b_0+c_2b_1+c_1b_2 \end{cases}$$

となるが，未知数と既知数が混在している．そこで，ベクトル表現を用いて整理すると，

$$\begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} b_2c_0 & b_1c_1 & (b_0+b_2)c_2 \\ b_1c_0 & (b_0+b_2)c_1 & (b_1+b_2)c_2 \\ b_0c_0 & b_2c_1 & b_1c_2 \end{bmatrix} = \begin{bmatrix} b_2 & b_1 & b_0+b_2 \\ b_1 & b_0+b_2 & b_1+b_2 \\ b_0 & b_2 & b_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

となる．さらに，両辺に b を成分とする行列の逆行列をかけると，

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} b_2 & b_1 & b_0+b_2 \\ b_1 & b_0+b_2 & b_1+b_2 \\ b_0 & b_2 & b_1 \end{bmatrix}^{-1} \begin{bmatrix} a_2 \\ a_1 \\ a_0 \end{bmatrix}$$

となり，既知数同士の乗算となるため商を導くことができる．

ここで，$(a_2\ a_1\ a_0)=(1\ 0\ 1),(b_2\ b_1\ b_0)=(0\ 1\ 1)$ の場合を具体的に計算すると，

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1+0 \\ 1 & 1+0 & 1+0 \\ 1 & 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

となる．$(c_2\ c_1\ c_0)=(0\ 1\ 1)$ であり，商を導くことができた．2 進数である $(0\ 1\ 1)$ を 10 進数に書き換えると 3 であり，同様にして任意の $(a_2\ a_1\ a_0),(b_2\ b_1\ b_0)$ の組み合わせによる商を表 2 に示す．

表 2　位数 2^3 の除算 $(a \div b)$

	÷	a 0	1	2	3	4	5	6	7
	1	0	1	2	3	4	5	6	7
	2	0	5	1	4	2	7	3	6
	3	0	6	7	1	5	3	2	4
b	4	0	7	5	2	1	6	4	3
	5	0	2	4	6	3	1	7	5
	6	0	3	6	5	7	4	1	2
	7	0	4	3	7	6	2	5	1

2.2.2　位数 2^4 の有限体の除算

次に位数 2^4 の有限体を例に説明する．なお，原始多項式は x^4+x+1 を使用する．位数 2^4 の有限体の場合．x の次数は 3 次までしか扱えない．そこで，原始多項式を用いることで 4 次以上の項をそれより小さい次数の項にする．

[手順 1]　位数 2^4 であることから，$a^3+a_2x^2+a_1x+a_0$ と $b^3+b_2x^2+b_1x+b_0$ の 2 つの多項式を用意し，商 $c^3+c_2x^2+c_1x+c_0$ を導くために多項式同士で除算すると，

$$(a^3+a_2x^2+a_1x+a_0) \div (b^3+b_2x^2+b_1x^1+b_0) = c^3+c_2x^2+c_1x+a_0$$

であり，ここで，$b^3+b_2x^2+b_1x+b_0$ を右辺に移項すると，

$$\begin{aligned} &a^3+a_2x^2+a_1x+a_0 = (c^3+c_2x^2+c_1x+a_0) \times (b^3+b_2x^2+b_1x^1+b_0) \\ &= c_3b_3x^6+c_3b_2x^5+c_3b_1x^4+c_3b_0x^3+c_2b_3x^5+c_2b_2x^4+c_2b_1x^3+c_2b_0x^2+c_1b_3x^4+c_1b_2x^3 \\ &\quad +c_1b_1x^2+c_1b_0x+c_0b_3x^3+c_0b_2x^2+c_0b_1x+c_0b_0 \end{aligned}$$

のように未知数と既知数の乗算となる．さらに x の次数で整理すると，

$$\begin{aligned} &a^3+a_2x^2+a_1x+a_0 \\ &= c_3b_3x^6+(c_3b_2+c_2b_3)x^5+(c_3b_1+c_2b_2+c_1b_3)x^4+(c_3b_0+c_2b_1+c_1b_2+c_0b_3)x^3+(c_2b_0 \\ &\quad +c_1b_1+c_0b_2)x^2+(c_1b_0+c_0b_1)x+c_0b_0 \end{aligned} \tag{2.2.5}$$

となる．

[手順 2]　原始多項式を用いて 4 次以上の項をそれより小さい次数の項に置き換えると，

$x^4 \to x+1$ より　$(c_3b_1+c_2b_2+c_1b_3)x^3 \to (c_3b_1+c_2b_2+c_1b_3)x+(c_3b_1+c_2b_2+c_1b_3)$ (2.2.6)

$x^5 \to x(x+1)$

$= x^2+x$ より　$(c_3b_2+c_2b_3)x^5 \to (c_3b_2+c_2b_3)x^2+(c_3b_2+c_2b_3)x$ (2.2.7)

$x^6 \to x(x^2+x)$

$= x^3+x^2$ より　$c_3b_3x^6 \to c_3b_3x^3+c_3b_3x^2$ (2.2.8)

となる．よって，式 (2.2.6)〜式 (2.2.8) を式 (2.2.5) に代入し x の次数で整理すると，

$$(c_3b_0+c_2b_1+c_1b_2+c_0b_3+c_3b_3)x^3+(c_2b_0+c_1b_1+c_0b_2+c_3b_3+c_3b_2+c_2b_3)x^2+(c_1b_0+c_0b_1 +c_3b_2+c_2b_3+c_3b_1+c_2b_2+c_1b_3)x+(c_0b_0+c_3b_1+c_2b_2+c_1b_3)=a_3x^3+a_2x^2+a_1x+a_0 \tag{2.2.9}$$

となる．

[手順３]　式 (2.2.9) の両辺の x の係数同士で等式が成立するので，

$$\begin{cases} a_3 = c_3b_0+c_2b_1+c_1b_2+c_0b_3+c_3b_3 \\ a_2 = c_2b_0+c_1b_1+c_0b_2+c_3b_3+c_3b_2+c_2b_3 \\ a_1 = c_1b_0+c_0b_1+c_3b_2+c_2b_3+c_3b_1+c_2b_2+c_1b_3 \\ a_0 = c_0b_0+c_3b_1+c_2b_2+c_1b_3 \end{cases}$$

となるが，未知数と既知数が混在している．そこで，ベクトル表現を用いて整理すると，

$$\begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} = \begin{bmatrix} b_3c_0 & b_2c_1 & b_1c_2 & (b_0+b_3)c_3 \\ b_2c_0 & b_1c_1 & (b_0+b_3)c_2 & (b_2+b_3)c_3 \\ b_1c_0 & (b_0+b_3)c_1 & (b_2+b_3)c_2 & (b_1+b_2)c_3 \\ b_0c_0 & b_3c_1 & b_2c_2 & b_1c_3 \end{bmatrix} = \begin{bmatrix} b_3 & b_2 & b_1 & b_0+b_3 \\ b_2 & b_1 & b_0+b_3 & b_2+b_3 \\ b_1 & b_0+b_3 & b_2+b_3 & b_1+b_2 \\ b_0 & b_3 & b_2 & b_1 \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}$$

となる．さらに両辺に b を成分とする行列の逆行列をかけると，

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} b_3 & b_2 & b_1 & b_0+b_3 \\ b_2 & b_1 & b_0+b_3 & b_2+b_3 \\ b_1 & b_0+b_3 & b_2+b_3 & b_1+b_2 \\ b_0 & b_3 & b_2 & b_1 \end{bmatrix}^{-1} \begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix}$$

となり，既知数同士の乗算となるため商を導くことができる．

3　有理法とその有効性

この章では，従来法に対する改善点とそこから考えられた有理法のねらいについて説明する．

3.1　従来法とそれに対する改善点

2 章では，有限体の乗算と除算の実現方法を説明した．ここで乗算と除算を実現する手順の内容を比較すると，除算を実現する手順では既知多項式と未知多項式の乗算を行い，さらに連立方程式を解くための 2.2.1 項のベクトル演算を行う必要がある．このベクトル演算によって演算時間の増加が発生している．

3.2　有理法のねらい

本研究では，有理表現を用いて四則演算を行う方法を提案し，有理法と呼ぶこととする．
有理法とは，任意の単一の有限体 a を分母分子のそれぞれに $a = \frac{a_n}{a_d}$ となる任意の単一の有限体 a_n と a_d を用いて表現し，除算の計算を後回しにして分母と分子を別々に計算する方法である．$a = \frac{a_n}{a_d}$ と $b = \frac{b_n}{b_d}$ の四則演算の有理法への変換を表 3 に示す．

有理表現を用いることで分母分子のそれぞれを加算または乗算で行うことができるため，有理表現で計算している間はベクトル演算を必要とする有限体の除算を行う必要がなくなり，それにより演算の高速化を目指す．しかし，加算や乗算においては演算時間は増加する．有理法を実現するために必要な有限体演算の回数を表 4 に示す．

表 4 から，有理法を適用すると加算や乗算では演算時間が増加することがわかり，さらに，有理表現から通常の有限体に戻す際に後回しにしていた分母と分子の除算の演算時間もかかる．その一方で，除算 1 回の演算時間よりも乗算 2 回の演算時間の方が短ければ，除算に有理法を適用することで演算時間は短縮される．したがって，有理法を適用することで増加する演算時間よりも短縮する演算時間の方が大きければ，特に，一連の計算の中で多くの除算が必要であれば，有理法は効果を発揮すると考えられる．

表 3　演算の変換

演算	通常の演算	有理表現への変換	有理法の実現
加算	$a+b$	$\frac{a_n}{a_d} + \frac{b_n}{b_d}$	$\frac{a_n b_d + a_d b_n}{a_d b_d}$
乗算	$a \times b$	$\frac{a_n}{a_d} \times \frac{b_n}{b_d}$	$\frac{a_n b_n}{a_d b_d}$
除算	$a \div b$	$\frac{a_n}{a_d} \div \frac{b_n}{b_d}$	$\frac{a_n b_d}{a_d b_n}$

表 4　有理法による演算回数

有理法の演算	有限体の加算回数	有限体の乗算回数	有限体の除算回数
加算	1	3	0
乗算	0	2	0
除算	0	2	0

4　有効性の検証

この章では，有理法を適用することで四則演算を用いた一連の計算手順における演算時間を短縮できる場合の検証を行う．

4.1　検証方法

有理法の有効性を検証するために，四則演算を用いた一連の計算手順における演算内で除算を多く行う逆行列演算について，有限体の次数や逆行列演算に用いる行列の大きさを変数として，演算時間の計測を行う．逆行列演算は，誤り訂正符号などの線形符号を実現する際に使用されるなど，実用上重要な演算である．また，除算の負荷がより大きい従来研究 [1] で用いられている方法に対しての有理法の有無について比較することで有理法の効果を検証する．ここで，従来研究で用いられている方法は次数と原始多項式を定数として固定することで計算量と分岐命令を減らし有限体演算の高速化を目指すものであり，本稿では固定法と呼ぶこととする．なお，演算には

$$\begin{bmatrix} 1^0 & 1^1 & 1^2 & \cdots & 1^{k-2} & 1^{k-1} \\ 2^0 & 2^1 & 2^2 & \cdots & 2^{k-2} & 2^{k-1} \\ \vdots & \vdots & & \ddots & \vdots & \vdots \\ (k-1)^0 & (k-1)^1 & (k-1)^2 & \cdots & (k-1)^{k-2} & (k-1)^{k-1} \\ k^0 & k^1 & k^2 & \cdots & k^{k-2} & k^{k-1} \end{bmatrix}$$

のような $k \times k$ 行列を使用し，掃出し法によって逆行列演算を行った．

4.2　検証結果と考察

図 1 に逆行列演算に用いる行列のサイズを 2×2〜6×6 で変化させ，固定法に有理法を適用しない場合に対する有理法を適用した場合の実行時間の比を示す．

固定法に有理法を適用しない場合に対する適用した場合の比率は，1 を下回っている場合有理法を適用することで演算時間を短縮できることを示している．図 1 より，ビット長が大きい

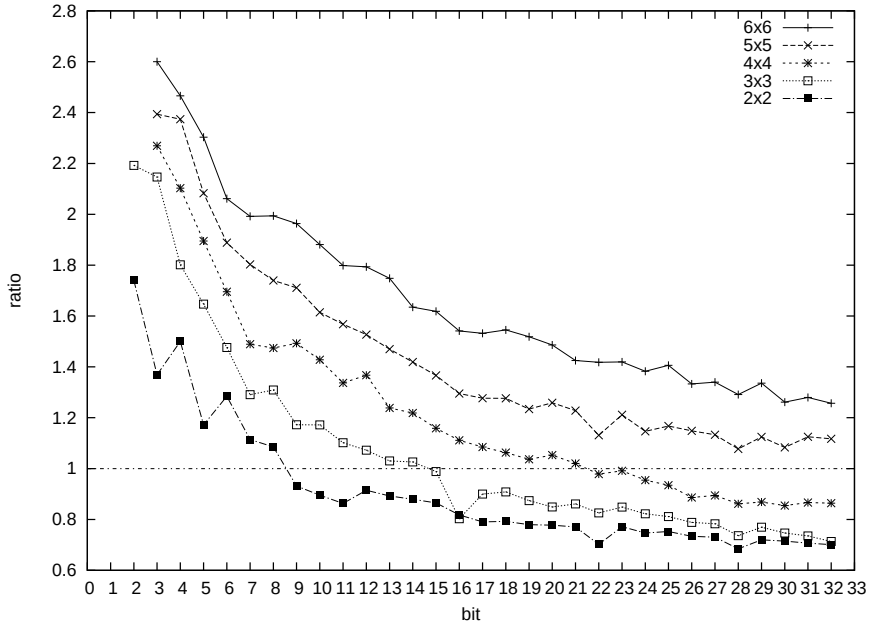

図 1　固定法に対する有理法の効果

ほど有理法の効果が有効であり，逆行列演算に用いた行列のサイズが大きいほど有理法の効果は有効でないことがわかる．以下ではこれらの理由について検討する．

4.2.1　行列のサイズに対する考察

行列のサイズが大きくなると逆行列演算を行う上での四則演算の回数が増える．掃出し法は，行列の値によって条件分岐があるため，処理手順はいつも同じというわけではない．しかし，各四則演算の回数のみに着目すると行列のサイズ n に対して，除算の演算回数は $\sum_{k=0}^{n-1}(2n-k)$ であるが，加算と乗算の演算回数は $(n-1)\sum_{k=0}^{n-1}(2n-k)$ である．そのため，サイズが大きくなることで逆行列演算中の除算の割合が小さくなるため有理法の効果は弱くなると考えられる．したがって，一連の演算の中で多くの除算が必要になる場合に有理法の効果が大きいことが確認された．

4.2.2　ビット長に対する考察

ビット長が大きいほど有理法が有効であることについて，各ビットごとに固定法による有限体の乗算 1 回の計算時間に対する除算 1 回の計算時間の比を実験的に求めた結果を図 2 に示す．

図 2 より計算時間の比はビット長におおむね比例しているとみなすことができる．この現象

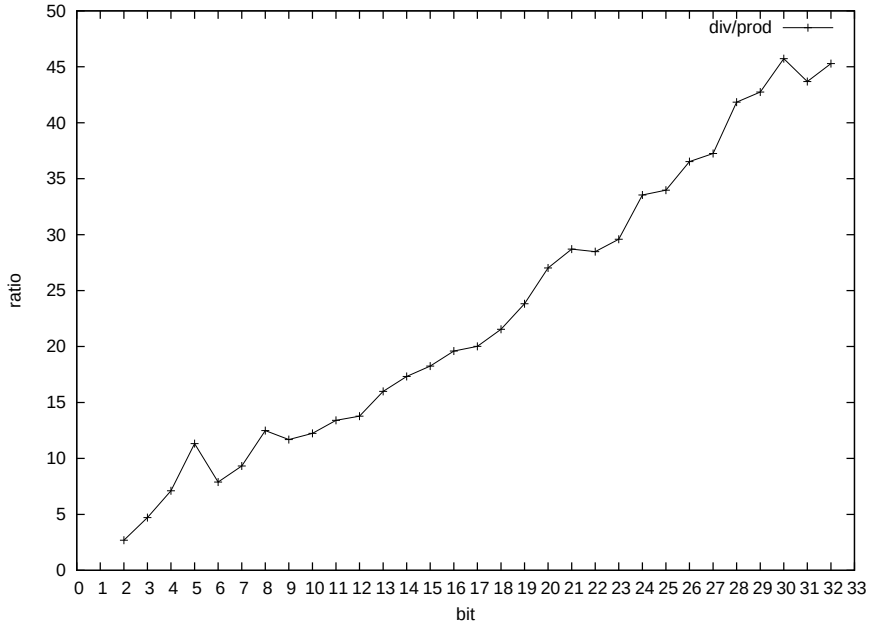

図 2　乗算に対する除算の演算時間の比

の理由は，実際の演算プログラムから説明することができる．図 3 に位数 2^3 の乗算プログラム，図 4 に位数 2^3 の除算プログラムを示す．図 3 の 1 から 3 行目より，乗算がビット長に比例するのに対し，図 4 の 1,13 行目より除算はビット長の二乗に比例しているためであると考えられる．このことから，有理法により高速化されると考えられる除算の演算時間の全体の演算時間に占める割合が大きくなるため，ビット長を大きくするほど有理法の効果は強くなったと考えられる．

また，ビット長が限りなく大きい場合について考える．ここで，有限体の加算 1 回の演算時間を T_a，乗算 1 回の演算時間を T_p，除算 1 回の演算時間を T_d とする．さらに，4.2.1 項より，行列のサイズ n に対して，除算の演算回数は $\sum_{k=0}^{n-1}(2n-k)$，加算と乗算の演算回数は $(n-1)\sum_{k=0}^{n-1}(2n-k)$ である．これらより，有理法を適用しない場合の演算時間は，

$$((2n-1)T_a+(2n-1)T_p+T_d)\sum_{k=0}^{n-1}(2n-k) \tag{4.1}$$

である．一方，有理法を適用した場合の有限体の加算 1 回の演算時間を T'_a，乗算 1 回の演算時間を T'_p，除算 1 回の演算時間を T'_d とすると表 4 より，

```
1         ab[0] = (b & 0x1)? a: 0;
2         ab[1] = (b & 0x2)? a: 0;
3         ab[2] = (b & 0x4)? a: 0;
4         y0 = ((ab[0]) ^ (ab[1] << 1) ^ (ab[2] << 2)) & 0x7;
5         y1 = (ab[1] >> 2) ^ (ab[2] >> 1);
6         r = y0;
7         r ^= (y1 & 0x1)? 0x3: 0;
8         r ^= (y1 & 0x2)? 0x6: 0;
```

図 3　位数 2^3 の乗算プログラム

```
1         for (k = 0; k < 3; k++){
2                 for (j = k; j < 3; j++){
3                         if (c[j] & (0x4 >> k)){
4                                 break;
5                         }
6                 }
7                 if (j >= 3){
8                         return(0);
9                 }
10                tmp = c[k], c[k] = c[j], c[j] = tmp;
11                tmp = e[k], e[k] = e[j], e[j] = tmp;
12
13                for (i = 0; i < 3; i++){
14                        if ((i != k) && (c[i] & (0x4 >> k))){
15                                c[i] = c[k] ^ c[i];
16                                e[i] = e[k] ^ e[i];
17                        }
18                }
19        }
```

図 4　位数 2^3 の除算プログラム

$$T'_a = T_a + 3T_p$$
$$T'_p = 2T_p$$
$$T'_d = 2T_p$$

となるので，有理法を適用した場合の演算時間は，有理表現による演算時間と有理表現から通常の有限体に戻す際に後回しにしていた分母と分子の除算の演算時間を足し合わせたものであり，

$$\begin{aligned}&((2n-1)T'_a + (2n-1)T'_p + T'_d)\sum_{k=0}^{n-1}(2n-k) + T_d \\ &= ((2n-1)T_a + (10n-3)T_p)\sum_{k=0}^{n-1}(2n-k) + T_d \end{aligned} \tag{4.2}$$

となる．ここで，ビット長が限りなく大きい場合は図 2 より，

$$\lim_{bit\to\infty}\frac{T_p}{T_d} = \lim_{bit\to\infty}\frac{1}{\frac{T_d}{T_p}} = \frac{1}{\infty} = 0 \tag{4.3}$$

となる．また，加算は排他的論理和として扱われることから，ビット長が限りなく大きい場合において除算 1 回に比べ加算 1 回の演算時間は非常に小さく，

$$\lim_{bit\to\infty}\frac{T_a}{T_d} = 0 \tag{4.4}$$

となる．式 (4.3)，式 (4.4) と，図 1 の各点が式 (4.2) を式 (4.1) で割ったものであることから，ビット長が限りなく大きい場合，

$$\begin{aligned}&\lim_{bit\to\infty}\frac{((2n-1)T_a + (10n-3)T_p)\sum_{k=0}^{n-1}(2n-k) + T_d}{((2n-1)T_a + (2n-1)T_p + T_d)\sum_{k=0}^{n-1}(2n-k)} \\ &= \lim_{bit\to\infty}\frac{((2n-1)\frac{T_a}{T_d} + (10n-3)\frac{T_p}{T_d})\sum_{k=0}^{n-1}(2n-k) + \frac{T_d}{T_d}}{((2n-1)\frac{T_a}{T_d} + (2n-1)\frac{T_p}{T_d} + \frac{T_d}{T_d})\sum_{k=0}^{n-1}(2n-k)} \\ &= \frac{1}{\sum_{k=0}^{n-1}(2n-k)}\end{aligned}$$

となる．したがって，ビット長が限りなく大きい場合の有理法を適用しない場合に対する有理法を適用しない場合の比は，除算の演算回数の逆数になると考えられる．

4.2.3 随時法に対する有効性と考察

従来研究 [1] で用いられている，次数や原始多項式を固定せず随時変数として宣言し演算を行う従来の方法を本研究では随時法と呼ぶこととする．この随時法に対しても同様に逆行列の演算を有理法の有無でそれぞれ行い比較することで有理法の効果を確認する．

図 5 は随時法に有理法を適用しない場合に対する有理法を適用した場合の効果を示したものである．図 1 と比較すると，有理法の有無の比率が 1 を下回る場合が非常に少なく，効果が弱いことを読み取ることができる．ここで，従来研究 [1] より，随時法に対する固定法の効果は乗算 1 回の計算時間よりも除算 1 回の方が小さいこと，除算のプログラム中における連立方程式を解くプログラムは比重が大きく，次数や原始多項式によっては演算速度の向上の妨げの要因となっていることが知られている．これらから，除算の演算時間全体に占める割合は固定法よりも随時法の方が小さいと考えられる．したがって，随時法は固定法よりも有理法の効果が弱くなったと考えられる．

その一方で，固定法に対する有理法の効果と同様に，ビット長が大きくなることや逆行列演算に用いる行列のサイズが小さいときに有理法の効果が強くなっていることが読み取れる．このことから，有理法は随時法に対しても効果を発揮すると考えられる．

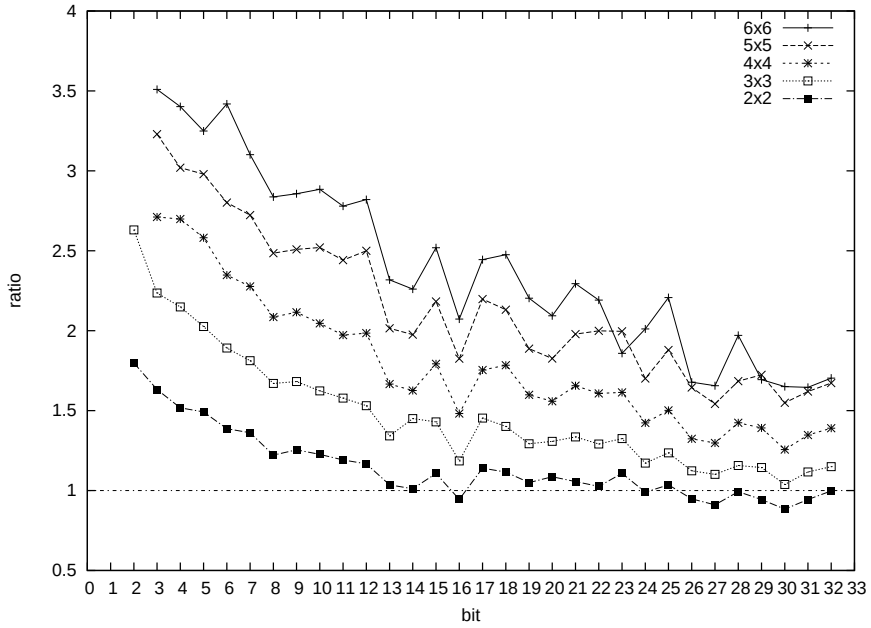

図 5　随時法に対する有理法の効果

5　まとめ

有限体の四則演算を行う際，除算では乗算を行った上で連立方程式を解かなければならず演算時間の増加を招いている．本研究では，基本的に有限体の加算と乗算の 2 種類のみの演算で，有理表現の四則演算が実現できる点に着目した．この有理表現を用いて演算を行う方法を

有理法と呼ぶこととし，過去の研究で行われている次数や原始多項式を定数として固定し演算を行う方法を固定法とし，さらに従来の次数や原始多項式を固定せず随時変数として演算を行う方法を随時法とした．除算の演算時間の短縮の一方で加算と乗算の演算時間が増加する問題があるため，逆行列演算を固定法と随時法のそれぞれに対して有理法の適用した場合と適用しない場合の逆行列演算の演算時間の比較実験を行った．結果として，ビット長が大きいほど有理法の効果が強くなることがわかった．これは乗算 1 回の計算時間に対する除算 1 回の計算時間の比が大きくなるためであると考えられる．また，逆行列演算に用いる行列のサイズが大きいほど有理法の効果は弱くなることがわかった．これは行列のサイズが大きくなるほど除算の回数よりも加算と乗算の回数の方が増加するため逆行列演算中の除算の割合が小さくなるためであると考えられる．さらに，随時法の場合に関して固定法と同様に，ビット長が大きいほど，行列のサイズを小さいほど有理法の効果が強くなった．ただし，固定法の場合と比較すると有理法の効果は弱いことが読み取れた．除算の演算時間全体に占める割合が，固定法よりも随時法の方が小さいためであると考えられる．

今後の課題として，任意の標数に関しても有効であるかという点，有理法を適用した方が演算時間が速くなる場合の除算と加算，乗算の比という点に関して比較検討することが挙げられる．

謝辞

本研究を行うにあたり，細かく指導してくださった指導教員の西新幹彦准教授に感謝の意を表する．

参考文献

[1] 乙井良太,「固定した原始多項式による標数 2 の有限体演算の高速化に関する検討」，信州大学工学部, 学士論文，2012 年.

[2] 土橋遼,「固定した原始多項式による標数 2 の有限体二乗演算の高速化に関する検討」，信州大学工学部, 学士論文，2013 年.

[3] 藤原良，神保雅一，符号と暗号の数理，共立出版株式会社，2008 年.

付録 A　本研究で用いた原始多項式

本研究で用いた次数ごとの原始多項式を表 5 に示す.

表 5　原始多項式一覧

次数	原始多項式
2	x^2+x+1
3	x^3+x+1
4	x^4+x+1
5	x^5+x^2+1
6	x^6+x+1
7	x^7+x+1
8	$x^8+x^7+x^2+x+1$
9	x^9+x^4+1
10	$x^{10}+x^3+1$
11	$x^{11}+x^2+1$
12	$x^{12}+x^8+x^2+x+1$
13	$x^{13}+x^5+x^2+x+1$
14	$x^{14}+x^{12}+x^2+x+1$
15	$x^{15}+x+1$
16	$x^{16}+x^{12}+x^3+x+1$
17	$x^{17}+x^3+1$
18	$x^{18}+x^7+1$
19	$x^{19}+x^5+x^2+x+1$
20	$x^{20}+x^3+1$
21	$x^{21}+x^2+1$
22	$x^{22}+x+1$
23	$x^{23}+x^5+1$
24	$x^{24}+x^7+x^2+x+1$
25	$x^{25}+x^3+1$
26	$x^{26}+x^6+x^2+x+1$
27	$x^{27}+x^5+x^2+x+1$
28	$x^{28}+x^3+1$
29	$x^{29}+x^2+1$
30	$x^{30}+x^{23}+x^2+x+1$
31	$x^{31}+x^3+1$
32	$x^{32}+x^{22}+x^2+x+1$

付録 B　ソースコード

B.1　固定法により逆行列演算を行うプログラム

位数 2^3 の有限体，原始多項式 x^3+x+1 としたときの 3×3 行列における固定法による逆行列演算プログラムを示す．

```
#include<stdio.h>
#include<time.h>

unsigned int
prod(unsigned int a, unsigned int b){
    unsigned int ab[3];
    unsigned int y00, y01;
    unsigned int r;

    ab[0] = (b & 0x1)? a: 0;
    ab[1] = (b & 0x2)? a: 0;
    ab[2] = (b & 0x4)? a: 0;
    y00 = ((ab[0]) ^ (ab[1] << 1) ^ (ab[2] << 2)) & 0x7;
    y01 = (ab[1] >>2) ^ (ab[2] >>1);
    r = y00;
    r ^= (y01 & 0x1)? 0x3: 0;
    r ^= (y01 & 0x2) ? 0x6 : 0;

    return(r);
}

unsigned int
div(unsigned int a, unsigned int b){
    unsigned int i, j, k;
    unsigned int c[3];
    unsigned int e[3];
    unsigned int tmp;

    c[0] = ((b << 0) ^ (b >> 2)) & 0x7;
    c[1] = ((b << 1) ^ (b >> 1) ^ (b >> 2)) & 0x7;
    c[2] = ((b << 2) ^ (b >> 1)) & 0x7;
    e[0] = (a & 0x4)? 0x7: 0x0;
    e[1] = (a & 0x2)? 0x7: 0x0;
    e[2] = (a & 0x1)? 0x7: 0x0;
    for (k = 0; k < 3 ; k++){
            for (j = k ; j < 3 ; j++){
                    if (c[j] & (0x4 >> k)){
                            break;
                    }
            }
            if (j >= 3){
                    return(0);
            }
            tmp = c[k] ,c[k] = c[j], c[j] = tmp;
            tmp = e[k] ,e[k] = e[j], e[j] = tmp;
            for(i = 0 ; i < 3 ; i++){
                    if ((i != k) && (c[i] & (0x4 >> k))){
                            c[i] = c[k] ^ c[i];
                            e[i] = e[k] ^ e[i];
                    }
            }
    }
    return((e[0] & 0x1) ^ (e[1] & 0x2) ^ (e[2] & 0x4));
}

int main(void){
```

```
int n = 3;
int r;
unsigned int a[n][n];
unsigned int b[n][n];
unsigned int x[n][n];
int i, j, k;
unsigned int t, u, tmp;
clock_t begin, end;

x[0][0] = 1;
x[0][1] = 1;
x[0][2] = prod(x[0][1], x[0][1]);
x[1][0] = 1;
x[1][1] = 2;
x[1][2] = prod(x[1][1], x[1][1]);
x[2][0] = 1;
x[2][1] = 3;
x[2][2] = prod(x[2][1], x[2][1]);

begin = clock();
for (r = 0; r < 1000000; r++){
    a[0][0] = x[0][0];
    a[0][1] = x[0][1];
    a[0][2] = x[0][2];
    a[1][0] = x[1][0];
    a[1][1] = x[1][1];
    a[1][2] = x[1][2];
    a[2][0] = x[2][0];
    a[2][1] = x[2][1];
    a[2][2] = x[2][2];

    b[0][0] = 1;
    b[0][1] = 0;
    b[0][2] = 0;
    b[1][0] = 0;
    b[1][1] = 1;
    b[1][2] = 0;
    b[2][0] = 0;
    b[2][1] = 0;
    b[2][2] = 1;

    for (k= 0; k < n; k++){
        printf("%d %d %d\n", a[0][0], a[0][1], a[0][2]);
        printf("%d %d %d\n", a[1][0], a[1][1], a[1][2]);
        printf("%d %d %d\n\n", a[2][0], a[2][1], a[2][2]);
        printf("%d %d %d\n", b[0][0], b[0][1], b[0][2]);
        printf("%d %d %d\n", b[1][0], b[1][1], b[1][2]);
        printf("%d %d %d\n\n", b[2][0], b[2][1], b[2][2]);

        for (i = k; i < n; i++){
            if (a[i][k] != 0){
                for (j = 0; j < n; j++){
                    tmp = a[k][j];
                    a[k][j] = a[i][j];
                    a[i][j] = tmp;
                    tmp = b[k][j];
                    b[k][j] = b[i][j];
                    b[i][j] = tmp;
                }
                break;
            }
        }
        if (i == n){
            printf("逆行列を持たない\n");
            return(0);
        }
        t = a[k][k];
        for (i = 0; i < n; i++) {
                a[k][i] = div(a[k][i], t);
```

```
                b[k][i] = div(b[k][i], t);
            }
            for (j = 0; j < n; j++){
                if (j != k){
                    u = a[j][k];
                    for (i = 0; i < n; i++){
                        a[j][i] = a[j][i] ^ prod(a[k][i], u);
                        b[j][i] = b[j][i] ^ prod(b[k][i], u);
                    }
                }
            }
        }
    }
    end = clock();

    printf("%d %d %d\n", b[0][0], b[0][1], b[0][2]);
    printf("%d %d %d\n", b[1][0], b[1][1], b[1][2]);
    printf("%d %d %d\n\n", b[2][0], b[2][1], b[2][2]);
    printf("3 bit 3x3 inv %f sec\n", ((double)(end - begin) / CLOCKS_PER_SEC / 1000000));
    return(0);
}
```

B.2 固定法に有理法を適用し逆行列演算を行うプログラム

位数 2^3 の有限体，原始多項式 x^3+x+1 としたときの 3×3 行列における固定法に有理法を適用した場合の逆行列演算プログラムを示す．

```
#include<stdio.h>
#include<time.h>
#include<string.h>

struct bunsu{
        unsigned int bunshi;
        unsigned int bunbo;
};

unsigned int
prod(unsigned int a, unsigned int b)
{
        unsigned int ab[3];
        unsigned int y00, y01;
        unsigned int r;

        ab[0] = (b & 0x1)? a: 0;
        ab[1] = (b & 0x2)? a: 0;
        ab[2] = (b & 0x4)? a: 0;
        y00 = ((ab[0]) ^ (ab[1] << 1) ^ (ab[2] << 2)) & 0x7;
        y01 = (ab[1] >>2) ^ (ab[2] >>1);
        r = y00;
        r ^= (y01 & 0x1)? 0x3: 0;
        r ^= (y01 & 0x2) ? 0x6 : 0;

        return(r);
}

unsigned int
div(unsigned int a, unsigned int b)
{
        unsigned int i, j, k;
        unsigned int c[3];
        unsigned int e[3];
        unsigned int tmp;
```

```
        c[0] = ((b << 0) ^ (b >> 2)) & 0x7;
        c[1] = ((b << 1) ^ (b >> 1) ^ (b >> 2)) & 0x7;
        c[2] = ((b << 2) ^ (b >> 1)) & 0x7;
        e[0] = (a & 0x4)? 0x7: 0x0;
        e[1] = (a & 0x2)? 0x7: 0x0;
        e[2] = (a & 0x1)? 0x7: 0x0;

        for (k = 0; k < 3 ; k++){
                for (j = k ; j < 3 ; j++){
                        if (c[j] & (0x4 >> k)){
                                break;
                        }
                }
                if (j >= 3){
                        return(0);
                }
                tmp = c[k] ,c[k] = c[j], c[j] = tmp;
                tmp = e[k] ,e[k] = e[j], e[j] = tmp;
                for(i = 0 ; i < 3 ; i++){
                        if ((i != k) && (c[i] & (0x4 >> k))){
                                c[i] = c[k] ^ c[i];
                                e[i] = e[k] ^ e[i];
                        }
                }
        }
        return((e[0] & 0x1) ^ (e[1] & 0x2) ^ (e[2] & 0x4));
}

struct bunsu
newprod(struct bunsu a, struct bunsu b){
        struct bunsu answer;

        answer.bunshi = prod(a.bunshi, b.bunshi);
        answer.bunbo = prod(a.bunbo, b.bunbo);

        return(answer);
}

struct bunsu
newdiv(struct bunsu a, struct bunsu b){
        struct bunsu answer;

        answer.bunshi = prod(a.bunshi, b.bunbo);
        answer.bunbo = prod(a.bunbo, b.bunshi);

        return(answer);
}

struct bunsu
newadd(struct bunsu a, struct bunsu b){
        struct bunsu answer;

        answer.bunshi = prod(a.bunshi, b.bunbo) ^ prod(a.bunbo, b.bunshi);
        answer.bunbo = prod(a.bunbo, b.bunbo);

        return(answer);
}

int main(void){
    int n;
    n = 3;

    int r;
    struct bunsu a[n][n];
    struct bunsu b[n][n];
    struct bunsu x[n][n];
    unsigned int ans[n][n];
    int i, j, k;
```

```
struct bunsu t, u, tmp, ;
clock_t begin, end;

x[0][0].bunshi = 1;
x[0][1].bunshi = 1;
x[0][2].bunshi = prod(x[0][1].bunshi, x[0][1].bunshi);
x[1][0].bunshi = 1;
x[1][1].bunshi = 2;
x[1][2].bunshi = prod(x[1][1].bunshi, x[1][1].bunshi);
x[2][0].bunshi = 1;
x[2][1].bunshi = 3;
x[2][2].bunshi = prod(x[2][1].bunshi, x[2][1].bunshi);
x[0][0].bunbo = 1;
x[0][1].bunbo = 1;
x[0][2].bunbo = 1;
x[1][0].bunbo = 1;
x[1][1].bunbo = 1;
x[1][2].bunbo = 1;
x[2][0].bunbo = 1;
x[2][1].bunbo = 1;
x[2][2].bunbo = 1;

begin = clock();
for (r = 0; r < 1000000; r++){
    a[0][0] = x[0][0];
    a[0][1] = x[0][1];
    a[0][2] = x[0][2];
    a[1][0] = x[1][0];
    a[1][1] = x[1][1];
    a[1][2] = x[1][2];
    a[2][0] = x[2][0];
    a[2][1] = x[2][1];
    a[2][2] = x[2][2];
    b[0][0].bunshi = 1;
    b[0][1].bunshi = 0;
    b[0][2].bunshi = 0;
    b[1][0].bunshi = 0;
    b[1][1].bunshi = 1;
    b[1][2].bunshi = 0;
    b[2][0].bunshi = 0;
    b[2][1].bunshi = 0;
    b[2][2].bunshi = 1;
    b[0][0].bunbo = 1;
    b[0][1].bunbo = 1;
    b[0][2].bunbo = 1;
    b[1][0].bunbo = 1;
    b[1][1].bunbo = 1;
    b[1][2].bunbo = 1;
    b[2][0].bunbo = 1;
    b[2][1].bunbo = 1;
    b[2][2].bunbo = 1;

    for (k= 0; k < n; k++){
        for (i = k; i < n; i++){
            if (a[i][k].bunshi != 0){
                for (j = 0; j < n; j++){
                    tmp = a[k][j];
                    a[k][j] = a[i][j];
                    a[i][j] = tmp;
                    tmp = b[k][j];
                    b[k][j] = b[i][j];
                    b[i][j] = tmp;
                }
                break;
            }
        }
        if (i == n){
            printf("逆行列を持たない\n");
            return(0);
```

```
            }
            t = a[k][k];
            for (i = 0; i < n; i++) {
                a[k][i] = newdiv(a[k][i], t);
                b[k][i] = newdiv(b[k][i], t);
            }
            for (j = 0; j < n; j++){
                if (j != k){
                    u = a[j][k];
                for (i = 0; i < n; i++){
                        a[j][i] = newadd(a[j][i], newprod(a[k][i], u));
                        b[j][i] = newadd(b[j][i], newprod(b[k][i], u));
                    }
                }
            }
        }
        ans[0][0] = div(b[0][0].bunshi, b[0][0].bunbo);
        ans[0][1] = div(b[0][1].bunshi, b[0][1].bunbo);
        ans[0][2] = div(b[0][2].bunshi, b[0][2].bunbo);
        ans[1][0] = div(b[1][0].bunshi, b[1][0].bunbo);
        ans[1][1] = div(b[1][1].bunshi, b[1][1].bunbo);
        ans[1][2] = div(b[1][2].bunshi, b[1][2].bunbo);
        ans[2][0] = div(b[2][0].bunshi, b[2][0].bunbo);
        ans[2][1] = div(b[2][1].bunshi, b[2][1].bunbo);
        ans[2][2] = div(b[2][2].bunshi, b[2][2].bunbo);
    }
    end = clock();
    printf("%d %d %d\n", ans[0][0], ans[0][1], ans[0][2]);
    printf("%d %d %d\n", ans[1][0], ans[1][1], ans[1][2]);
    printf("%d %d %d\n\n", ans[2][0], ans[2][1], ans[2][2]);
    printf("3 bit 3 newinv %f sec\n", ((double)(end - begin) / CLOCKS_PER_SEC /1000000));
    return(0);
}
```