Vol.2015-HPC-151 No.22
2015/10/1

# ビット数の大きな整数の乗算アルゴリズムの実用性の検証

橋本 翔太[1]　上土井 陽子[1]　若林 真一[1]

**概要**：近年，RSA 暗号，El-Gamel 暗号のような公開鍵暗号技術において，公開鍵を計算するためにビット数の大きな整数の乗算が行われている．ビット数の大きな整数の乗算を，通常の筆算などの桁上げ処理を考慮した逐次的処理を基本とする乗算アルゴリズムでは計算すると時間がかかってしまう．そこで，本稿では既存の乗算アルゴリズムの中から特に高速であるとされるフーリエ変換と畳み込み定理を用いた手法に着目する．フーリエ変換では三角関数や円周率などの浮動小数点数を使用するので，数値計算において誤差を生じてしまう．前述した暗号技術では乗算結果が不正確になることは許されないので，計算が高速であるだけでなく計算精度も高いことが望まれる．本稿では，計算時間と計算精度について，着目した乗算アルゴリズムが実用性をもつかどうか実験的に検証する．

## 1. はじめに

整数の乗算は，あらゆるアルゴリズムの中でよく用いられる基本的な算術命令の一つである．そのため乗算は，アルゴリズムに計算の基本的なステップとして遍在する．一般的なアルゴリズムの中で乗算の対象となる整数は，ハードウェアにより高速に処理できる程度のビット長が小さな整数であるため，乗算の計算量はアルゴリズムの全体効率にほとんど影響しない．しかし，ビット長の大きな整数同士の乗算を必要とする RSA 暗号，El-Gamel 暗号，楕円曲線暗号のような暗号システムの出現によって，乗算の計算量の低減に関する研究は大きな重要性をもつことになった．

数多くある乗算アルゴリズムの中でも，1970 年ごろから 2007 年まで最も高速であると知られていた高速フーリエ変換を用いた Shönhage-Strassen の乗算アルゴリズムがある．2 つの $n$ 桁の整数の乗算を行う計算量は $O(N \log N \log\log N)$ である [5]．2007 年から今現在まで計算量上，最も高速であると知られている Fürer の乗算アルゴリズム [3] がある．計算量は $O(N \log N \cdot 2^{\log^* n})$ である．

本稿では，上記のような暗号技術で用いる桁数の大きな整数に対する乗算の高速アルゴリズムとして，Fürer の乗算アルゴリズムの中でもフーリエ変換と畳み込み演算に注目し，基本となっているアルゴリズムを実装してシミュレーションによる実験的性能評価を行う．また，フーリエ変換の計算では，円周率 $\pi$ や三角関数 $\sin, \cos$ などの無理数の計算を必要とするので，乗算結果の計算精度にどのような影響を与えるかについても考察する．

## 2. 準備

### 2.1 整数の表現

本研究で用いる整数の表現方法について述べる．桁数が $N/2$ の非常に大きな 10 進数の整数 $A$ を式 (1) のように多項式表現する．ここで $N$ は 2 のべき乗で表現可能な十分に大きな整数とする．

$$A = a_0 \times 10^0 + a_1 \times 10^1 + \cdots + a_{N/2-1} \times 10^{N/2-1} \quad (1)$$

次に $a_i = 0 (N/2 \leq i < N,\ i$ は整数) とし，式 (1) の係数系列を式 (2) のように表現する．

$$(\, a_0, a_1, \cdots, a_{N/2-1}, 0, 0, \cdots, 0) \quad (2)$$

さらに，係数系列の各要素を時間の関数 $f(t)$ の関数値とすると，式 (3) のように表現できる．

$$(\, f(0), f(1), \cdots, f(N/2-1), \cdots, f(N-1)\,) \quad (3)$$

式 (2)，式 (3) の間には，次のような関係が成り立っている．

$$f(t) = \begin{cases} a_t & (0 \leq t < N/2) \\ 0 & (N/2 \leq t < N) \end{cases} \quad (4)$$

例えば，4 桁の 10 進数の整数

$$8472 = 2 \times 10^0 + 7 \times 10^1 + 4 \times 10^2 + 8 \times 10^3$$

[1] 広島市立大学大学院 情報科学研究科
Faculty of Information Sciences ,
Hiroshima City University

© 2015 Information Processing Society of Japan

と表せるので，係数系列は

$$(2, 7, 4, 8, 0, 0, 0, 0)$$

である．また，この係数系列を時間成分の系列とみなすと，図 1 のように係数を波としても表現できる．

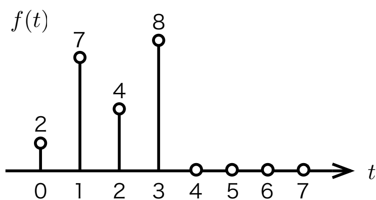



**図 1** 整数を波形で表現

### 2.2 フーリエ変換 [6]

乗算を行う整数を図 1 のように波で表現し，その波に対しフーリエ変換を適用し，実空間からフーリエ空間へ移動する．また，逆にフーリエ空間から実空間に移動する場合は，逆フーリエ変換を適用する (図 2)．

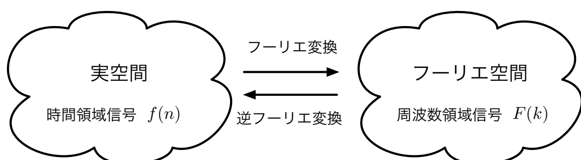



**図 2** 空間の移動

#### 2.2.1 離散フーリエ変換

信号 $f(n)$ に対するつぎの計算を，$N$ 点離散フーリエ変換 (discrete Fourier transform; DFT) という．

$$F(k) = \sum_{n=0}^{N-1} f(n)\, e^{-j2\pi nk/N} \tag{5}$$

#### 2.2.2 逆離散フーリエ変換

$F(k)$ から元の信号 $f(n)$ を求めるつぎの計算を，逆離散フーリエ変換 (inverse discrete Fourier transform; IDFT) という．

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} F(k)\, e^{j2\pi nk/N} \tag{6}$$

#### 2.2.3 高速フーリエ変換

一般に，離散フーリエ変換 (DFT) を用いて時間領域の信号 $f(n)$ を周波数領域の信号 $F(k)$ に変換するためには膨大な計算を必要とする．

しかし，Cooley と Tukey によって開発された高速フーリエ変換 [1](fast Fourier transform; FFT) を利用すれば DFT に比べて少ない計算量で計算可能である．FFT は，$N = 2^i$ ($i$ は整数) 点離散フーリエ変換の計算量を $O(N^2)$ から $O(N \log N)$ に削減するものである．

式 (5) に対して，

$$W_N = e^{-j2\pi/N} = \cos\frac{2\pi}{N} - j\sin\frac{2\pi}{N}$$

を利用すると，

$$F(k) = \sum_{n=0}^{N-1} f(n)\, W_N^{nk} \tag{7}$$

となる．ここで，式 (7) の $W_N$ は回転因子とよばれ，図 3 のような単位円を $N$ 分割したものであり，周期性と対称性をもつ．この性質とバタフライ演算の繰り返し使用により計算量の大幅な低減が可能となるが，詳細については省略する．

図 3 は $N = 8$ のときの回転因子を表したものである．

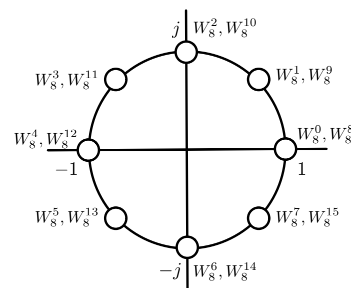



**図 3** 回転因子 $W_8$

#### 2.2.4 逆高速フーリエ変換

次の計算を高速逆フーリエ変換 (inverse fast Fourier transform; IFFT) という．

$$f(n) = \frac{1}{N} \sum_{k=0}^{N-1} f(n)\, W_N^{-nk} \tag{8}$$

### 2.3 畳み込み演算

畳み込み演算とは，関数 $f(t)$ を平行移動しながら関数 $g(t)$ を重ね足し合わせる二項演算 $(f * g)(t)$ である．離散値で定義された関数 $f(t), g(t)$ に対する畳み込みは，式 (9) で定義される．

$$(f * g)(t) = \sum_{u=0}^{t} f(u)\, g(t-u) \tag{9}$$

式 (9) は巡回畳み込みになるので，本研究で扱う整数の係数系列では $N/2$ 桁の整数に対して桁数を 2 倍にし，上位の桁の部分に 0 をつめて巡回しないようにし，畳み込み演算を利用する．その利用方法は後に述べる．

### 2.4 畳み込み定理

関数 $f(t), g(t)$ の畳み込み演算とフーリエ変換には，式 (10) の関係が成り立つ．ただし，$\mathcal{F}[f(t)]$ は関数 $f(t)$ をフーリエ変換したものであるとする．

$$\mathcal{F}[(f * g)(t)] = \mathcal{F}[f(t)] \cdot \mathcal{F}[g(t)] \tag{10}$$

© 2015 Information Processing Society of Japan

式 (10) から，関数 $f(t), g(t)$ に対する畳み込みをフーリエ変換したものは，関数 $f(t), g(t)$ をそれぞれフーリエ変換したもの同士の積と等しいということが分かる．さらに，次の式も導ける．

$$(f * g)(t) = \mathcal{F}^{-1}\Big[\mathcal{F}[f(t)] \cdot \mathcal{F}[g(t)]\Big] \tag{11}$$

式 (11) から，畳み込みをフーリエ変換と逆フーリエ変換を使うことで，高速に計算できる可能性がある．

## 3. 整数の乗算

本節から具体的な乗算アルゴリズムについて，8 桁の 10 進の整数 $A = 24567814$，$B = 82351471$ を例に，積 $A \times B$ を計算する．

簡単に説明するために，この 8 桁の整数 $A, B$ をそれぞれ 2 桁ずつそれぞれ区切る．すなわち，8 桁の 10 進の整数を 4 桁の 100 進数の整数としてみなすと，

$$A = 14 \times 100^0 + 78 \times 100^1 + 56 \times 100^2 + 24 \times 100^3 \tag{12}$$

と表現でき $(N = 8)$，$a_i = 0\,(4 \leq i < 8,\ i$ は整数$)$ とすると，整数 $A$ の係数系列は，

$$(\,14,\,78,\,56,\,24,\,0,\,0,\,0,\,0) \tag{13}$$

と表すことできる．整数 $B$ の係数系列についても同様に．

$$(\,71,\,14,\,35,\,82,\,0,\,0,\,0,\,0) \tag{14}$$

となる．

次に，式 (13) で表した係数系列を時間に関する関数 $f(t)\,(\,0 \leq t < 8)$ の関数値の系列，すなわち時間成分の系列とみなし，図 4 のように横軸を時間とした波で表現する．整数 $B$ についても同様に，式 (14) で表した係数系列を時間に関する関数 $g(t)\,(\,0 \leq t < 8)$ の関数値の系列，すなわち時間成分の系列とみなし，図 5 のように横軸を時間にした波で表現する．

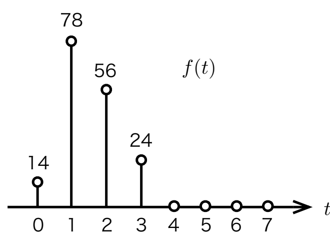



**図 4** 整数 $A$ の波による表現

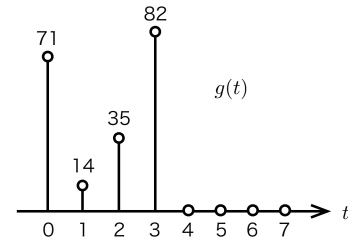
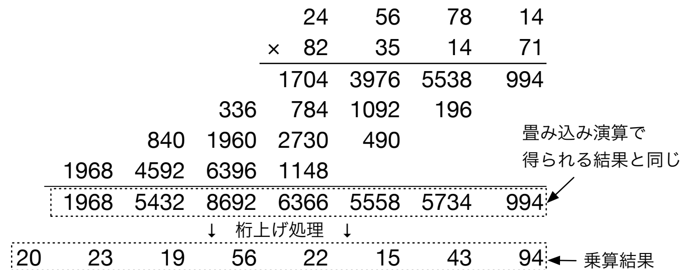



**図 5** 整数 $B$ の波による表現



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | 24 | 56 | 78 | 14 |
| | | | × | 82 | 35 | 14 | 71 |
| | | | | 1704 | 3976 | 5538 | 994 |
| | | | 336 | 784 | 1092 | 196 | |
| | | 840 | 1960 | 2730 | 490 | | |
| | 1968 | 4592 | 6396 | 1148 | | | |
| | 1968 | 5432 | 8692 | 6366 | 5558 | 5734 | 994 |
| | | | ↓ 桁上げ処理 ↓ | | | | |
| 20 | 23 | 19 | 56 | 22 | 15 | 43 | 94 |

畳み込み演算で得られる結果と同じ
乗算結果

**図 6** 筆算による乗算

### 3.1 筆算による乗算 [4]

通常の筆算による乗算を図 6 に示す．乗算の計算量は，$O(N^2)$ である．

### 3.2 畳み込み演算を用いた乗算アルゴリズム

式 (9) で定義した畳み込み演算を整数 $A, B$ の係数系列を時間成分の系列とみなしたときの関数 $f(t), g(t)$ に対して適用する．

$t = 0, 1, 7$ のときを例に，畳み込み演算の計算 (式 (9) と波形を対応させて確認する．

- $t = 0$ のとき

$$(f * g)(0) = \sum_{u=0}^{0} f(u)\, g(0 - u) = f(0) \cdot g(0)$$

これは図 7 における関数 $f, g$ の波の重なった部分の積に対応している．波の重なった部分の積を畳み込み演算 $(f * g)(0)$ の結果として，図 10 の $t = 0$ にプロットしている．$(f * g)(0) = 994$ となる．

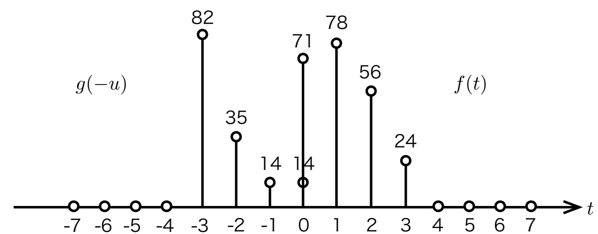



**図 7** $t = 0$ のときの畳み込み演算

- $t = 1$ のとき

$$\begin{aligned}(f * g)(1) &= \sum_{u=0}^{1} f(u)\, g(1 - u) \\ &= f(0) \cdot g(1) + f(1) \cdot g(0)\end{aligned}$$

$t = 0$ のときに比べて，関数 $g$ を右に 1 だけシフトしたものである．波の重なった部分の積の和を畳み込み演算 $(f * g)(1)$ の結果として，図 10 の $t = 1$ にプロッ

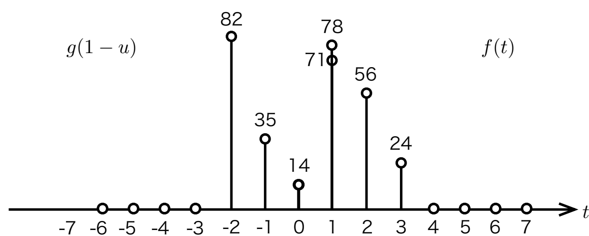



**図 8**　$t = 1$ のときの畳み込み演算

トしている．$(f * g)(1) = 5734$ となる．

- $t = 7$ のとき

$$
\begin{aligned}
(f * g)(7) &= \sum_{u=0}^{7} f(u)\, g(7-u) \\
&= f(0)\cdot g(7) + f(1)\cdot g(6) + f(2)\cdot g(5) \\
&\quad + f(3)\cdot g(4) + f(4)\cdot g(3) + f(5)\cdot g(2) \\
&\quad + f(6)\cdot g(1) + f(7)\cdot g(0)
\end{aligned}
$$

$t = 0$ のときに比べて，関数 $g$ を右に 7 だけシフトしたものである．波の重なった部分の積の和を畳み込み演算 $(f * g)(7)$ の結果として，図 10 の $t = 7$ にプロットしている．$(f * g)(7) = 0$ となる．

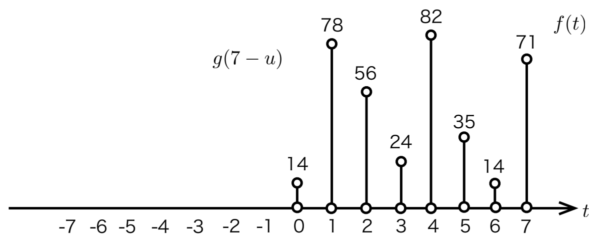



**図 9**　$t = 7$ のときの畳み込み演算

以上の $t = 0, 1, 7$ 以外の時間 $t$ についての畳み込み演算を行い，プロットした図 10 をみる．図 10 と図 6 の筆算に注目すると，畳み込み演算で得られる結果が筆算においても得られるということが分かる．

そこで，畳み込み値を桁上げ処理することで乗算結果を得られるので，以降は畳み込み値を乗算結果として示す．

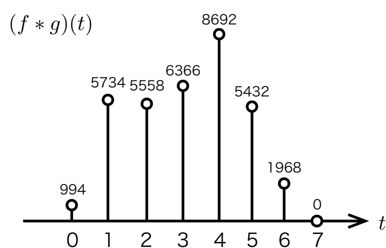



**図 10**　畳み込み演算結果

以上の畳み込み演算を用いた乗算アルゴリズムの流れを図 11 に示す．すべての計算を実空間で行っている．

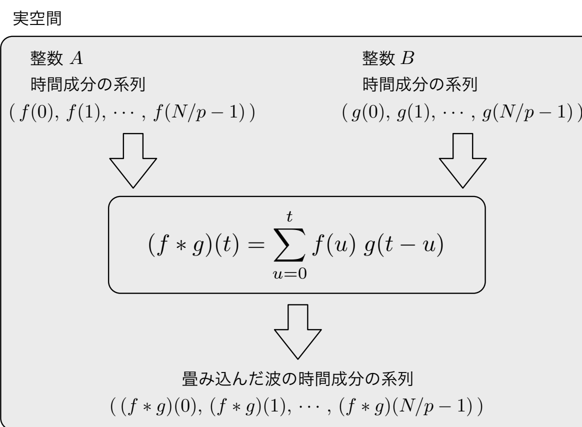



**図 11**　畳み込み演算を用いた乗算アルゴリズム

### 3.3 離散フーリエ変換と畳み込み定理を用いた乗算アルゴリズム

整数 $A \times B$ の計算結果は，整数 $A, B$ の係数系列を時間成分の系列とみなしたときの関数 $f(t), g(t)$ の畳み込みを桁上げ処理することによって得られる．

前述した離散フーリエ変換と畳み込み定理を用いて畳み込みを計算するアルゴリズムの流れを図 12 に示す．

**手順 1**　整数 $A, B$ の各時間成分の系列に離散フーリエ変換をかけ，時間成分の系列を周波数成分のベクトルにする．

**手順 2**　変換された周波数成分のベクトルの成分ごとに積を計算することで，関数 $f, g$ の畳み込んだ波の周波数成分ベクトルが得られる．

**手順 3**　得られた周波数成分のベクトルに逆離散フーリエ変換をかけ，畳み込んだ波の時間成分の系列が得られる（式 (9) の畳み込み定理より）．

**手順 4**　得られた畳み込んだ波の時間成分の系列を係数系列に見方を変え，多項式に変形することで乗算結果が得られる．

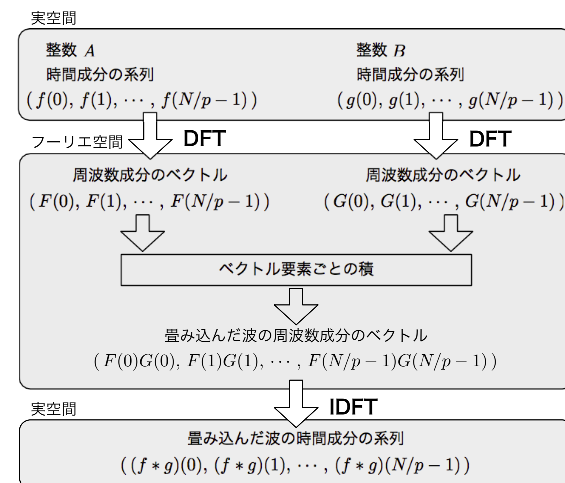



**図 12**　DFT と畳み込み定理を用いた乗算アルゴリズム

畳み込み演算を用いたアルゴリズムと異なり，実空間か

© 2015 Information Processing Society of Japan

**表 1** 各手法の乗算平均計算時間

| 桁数 (10 進) $N$ | 計算時間 [ms] | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 畳み込み演算 | | | DFT と畳み込み定理 | | | FFT と畳み込み定理 | | |
| | $p=1$ | $p=2$ | $p=4$ | $p=1$ | $p=2$ | $p=4$ | $p=1$ | $p=2$ | $p=4$ |
| 16 | 0.029 | 0.021 | 0.017 | 0.095 | 0.039 | 0.024 | 0.035 | 0.024 | 0.017 |
| 32 | 0.035 | 0.025 | 0.019 | 0.291 | 0.087 | 0.036 | 0.042 | 0.028 | 0.022 |
| 64 | 0.062 | 0.038 | 0.026 | 1.108 | 0.290 | 0.094 | 0.072 | 0.043 | 0.033 |
| 128 | 0.131 | 0.064 | 0.040 | 4.395 | 1.104 | 0.310 | 0.133 | 0.074 | 0.054 |
| 256 | 0.327 | 0.135 | 0.072 | 17.673 | 4.472 | 1.172 | 0.262 | 0.138 | 0.960 |
| 512 | 0.978 | 0.339 | 0.152 | 69.195 | 17.578 | 4.542 | 0.543 | 0.276 | 0.190 |
| 1024 | 3.270 | 1.007 | 0.384 | 274.389 | 68.871 | 17.903 | 1.121 | 0.565 | 0.388 |
| 2048 | 11.845 | 3.317 | 1.076 | 1097.118 | 274.616 | 69.723 | 2.298 | 1.160 | 0.797 |
| 4096 | 44.250 | 11.935 | 3.468 | 4380.119 | 1096.042 | 275.499 | 4.777 | 2.409 | 1.630 |

らフーリエ空間に移動し，成分ごとの積を計算してから実空間に戻すという空間移動を行っている．

### 3.4 高速フーリエ変換と畳み込み定理を用いた方法

前節で離散フーリエ変換により周波数成分のベクトルに変換していた部分を高速フーリエ変換に置き換えたアルゴリズムの流れを図 13 に示す．

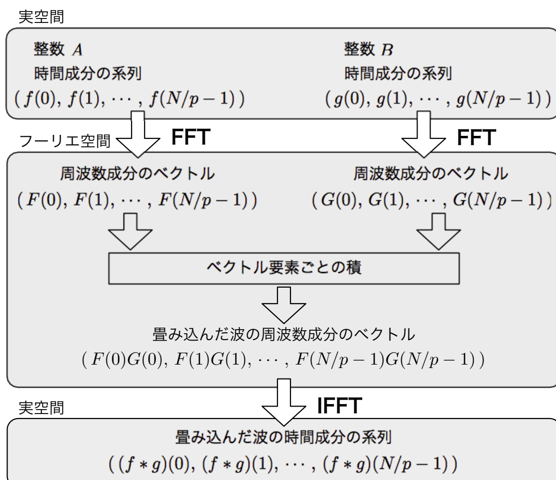



**図 13** FFT と畳み込み定理を用いた乗算アルゴリズム

## 4. 実験による実用性の検証

畳み込み演算を用いた乗算アルゴリズム (図 11)，DFT と畳み込み定理を用いた乗算アルゴリズム (図 12)，FFT と畳み込み定理を用いた乗算アルゴリズム (図 13) の 3 つの乗算アルゴリズムを C 言語を用いて，変数を 64 ビットの倍精度浮動小数点型 `double` の計算精度により実装し，gcc 4.2.1 コンパイラを用いてコンパイルし，CPU:2.93GHz 6-Core Intel Xeon の計算機上に実現しシミュレーション実験を行った．実験で用いた整数は，10 進数で 1 から 9 の数をランダムに $N/2$ 個並べた数値を $N/2$ 桁の 10 進数とみなし生成した整数である．

本稿では，$N = 2^i$ $(4 \leq i \leq 12)$ となる各 $N$ に対して，異なる 5 つの整数の組 Data1〜Data5 に対して乗算結果を計算する実験を行った．

### 4.1 計算時間に関する評価

上記の 3 つのアルゴリズムにおいて，整数 $A, B$ の組合せ Data1〜Data5 に対して乗算結果を得るまでの計算にかかった時間の平均値を表 1 にまとめた．表中の $N$ は計算に用いた 10 進整数の桁数，$p$ は整数の分割桁数を表す．

表 1 から，実験した中で最大の桁数 $N = 4096$ のとき，分割桁数 $p = 4$ の FFT と畳み込み定理を用いた手法が他の手法に比べて，最も高速に乗算結果を得られることがわかった．FFT と畳み込み定理を用いた手法は，畳み込み演算を用いた手法に比べて約 2 倍，DFT と畳み込み定理を用いた手法に比べて約 270 倍の高速化となった．

また，同じ桁数 $N$ に対して分割桁数 $p$ を大きくすると，すべての手法に対して計算時間が小さくなった．これは分割桁数を大きくすることにより，畳み込み演算の回数やフーリエ変換を行う回数が少なくて済むからである．畳み込み演算を用いた手法の場合，分割桁数 $p = 4$ のときは $p = 1$ に比べて約 12 倍の高速化となった．また，畳み込み演算，DFT と畳み込み定理を用いた手法の場合は，桁数増加に対して分割桁数が大きいほど計算時間の増加も急激である．しかし，FFT と畳み込み定理を用いた手法の場合はいずれの分割桁数に対しても桁数に比例した計算時間であることもわかった．

以上から，FFT を用いて分割桁数 $p$ を大きくとると，桁数が大きい場合でも高速に畳み込み値を計算することができるとわかった．

### 4.2 計算精度に関する評価

前節では，分割桁数 $p$ が大きいほど計算時間が小さくなることがわかった．しかし，DFT や FFT のようなフーリエ変換を用いる手法において，分割桁数 $p$ を大きくすると，畳み込み演算による手法で得られた畳み込み値と計算結果に違いが生じた．以降は，畳み込み演算による手法で得ら

れた畳み込み値が正しい畳み込み値とする．

### 4.2.1 畳み込みの計算精度

Data1〜Data5 の整数 $A, B$ の組み合わせに対して，畳み込み演算を用いた手法で得られた畳み込み値と，DFT や FFT と畳み込み定理を用いた手法で得られた畳み込み値が異なる畳み込み値を計算した平均誤り率を，表 2 に示す．表 2 における割合の計算は次の通りである．

$$\text{平均誤り率}\ [\%] = \frac{\text{畳み込み計算誤り個数の平均}}{\text{畳み込み計算を行う回数}\ (= N/p)} \times 100$$

**表 2** 畳み込み値の計算精度

| 桁数 (10 進) $N$ | 畳み込み計算平均誤り率 [%] | | | | | |
|---|---|---|---|---|---|---|
| | DFT と畳み込み定理 | | | FFT と畳み込み定理 | | |
| | $p=1$ | $p=2$ | $p=4$ | $p=1$ | $p=2$ | $p=4$ |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 |
| 32 | 0 | 0 | 0 | 0 | 0 | 0 |
| 64 | 0 | 0 | 10.00 | 0 | 0 | 0 |
| 128 | 0 | 0 | 41.25 | 0 | 0 | 0 |
| 256 | 0 | 0 | 82.19 | 0 | 0 | 0 |
| 512 | 0 | 0 | 93.13 | 0 | 0 | 3.13 |
| 1024 | 0 | 0 | 98.05 | 0 | 0 | 28.75 |
| 2048 | 0 | 0 | 99.57 | 0 | 0 | 70.98 |
| 4096 | 0 | 10.21 | 99.79 | 0 | 0 | 80.86 |

表 2 より，DFT と畳み込み定理を用いた手法では桁数 $N = 64$ 以上のときに正確でない畳み込み値の計算をする場合があることがわかる．分割桁数 $p = 4$，桁数 $N = 512$ 以上のとき，平均誤り率が 90%を超えており，桁数 $N = 4096$ のときほぼすべての畳み込み値の計算で不正確な計算が行われていることがわかった．また，FFT と畳み込み定理を用いた手法では分割桁数 $p = 1, 2$ ときはいずれの桁数においても正確な畳み込み計算をできた．しかし，分割桁数 $p = 4$ のとき，桁数 $N = 512$ 以上になると DFT と畳み込み定理を用いた手法よりも平均誤り率が低いが，最悪で約 80%の正確でない畳み込み値の計算が行われていた．これは，浮動小数点数の計算の増加，フーリエ変換を行う際の三角関数 $\sin, \cos$，円周率 $\pi$ を含む無理数の計算，コンパイラやプログラムの書き方の違いによって生じるものであると考えられる．

本稿では Data1〜Data5 の 5 つのデータに対してのみ実験を行っているため，表 2 の平均誤り率が 0 となっている桁数や分割桁数のときに必ず 0 であるかどうかは，より多くの種類のデータに対して実験を行う，または数値計算誤差解析のような数学的な証明が必要である．

### 4.2.2 乗算結果の計算精度

次に畳み込み値の計算精度の乗算結果への影響を考察する．得られた畳み込み値に対して桁上げ処理を行い，乗算結果を 10 進整数にしたとき，誤っている最上位桁の最大値と最小値を表 3 に示す．表の数値が大きければ大きいほど，上位の桁が誤っているので，正しい乗算結果との誤差が広がっていることを，× のところは誤りがなかったことを表す．

例えば，桁数 $N = 512$，分割桁数 $p = 4$ のときの乗算結果の誤りが生じた最高桁は，最悪の場合で DFT と畳み込み定理を用いた手法の場合が 509 桁目に対して，FFT と畳み込み定理を用いた手法の場合は 145 桁目であり，FFT と畳み込み定理を用いた手法の方が正しい乗算結果に比べると誤差は小さくなっていることがわかる．しかし，分割桁数 $p = 4$ のとき，FFT と畳み込み定理を用いた手法では桁数 $N = 1204, 2048, 4096$ と大きくしていくと，誤りが生じた最高桁数の最大値と最小値の差が大きくなっている．最悪の場合は，DFT と畳み込み定理を用いた手法とあまり変わらない精度となった．つまり，畳み込み値の計算精度の高低は，乗算結果の精度の高低と対応していないことがわかる．

さらに，誤りが生じた桁数は畳み込み値の 1 桁目あるいは 2 桁目の下位の桁に対応していることもわかった．例えば，得られた畳み込み値に対して桁上げ処理を行うのとき，分割桁数 $p = 4$ ならば，$10^4$ で割った余りを乗算結果の各桁に対応させ，商は次の位の桁にあげる．$p = 4$ のとき乗算結果に誤りが生じた最高桁の桁数を 4 で割るとすべて 1 余る．これは畳み込み値の 1 の位が間違っていることを表している．このことから，表 2 から，桁数 $N = 4096$，分割桁数 $p = 4$ のとき，FFT と畳み込み定理を用いた手法では約 80%の畳み込み値の計算が誤っているが，この誤り率の大きさが乗算結果の最高桁が誤る確率と等しくならず，低くなることが予想できる．

よって，畳み込み値の平均誤り率が低いとしても誤りが生じた畳み込み値の 1 の位に対応する乗算結果の桁数が大きいときは，正しい乗算結果との誤差が大きくなる．反対に，畳み込み値の平均誤り率が高いとしても下位の桁に集中していれば，誤差は小さくなる．

**表 3** 乗算結果の計算精度

| 桁数 (10 進) $N$ | 乗算結果に誤りが生じた最高桁の最大値 (最小値)[桁目] | | | | | |
|---|---|---|---|---|---|---|
| | DFT と畳み込み定理 | | | FFT と畳み込み定理 | | |
| | $p=1$ | $p=2$ | $p=4$ | $p=1$ | $p=2$ | $p=4$ |
| 16 | × | × | × | × | × | × |
| 32 | × | × | × | × | × | × |
| 64 | × | × | 25(25) | × | × | × |
| 128 | × | × | 98(89) | × | × | × |
| 256 | × | × | 253(249) | × | × | × |
| 512 | × | × | 509(501) | × | × | 145(105) |
| 1024 | × | × | 1001(997) | × | × | 957(465) |
| 2048 | × | × | 1957(1957) | × | × | 1965(1029) |
| 4096 | × | 3665(3551) | 4093(4085) | × | × | 4077(2910) |

前節に述べた計算時間に関する考察を考慮に入れると，暗号技術で用いるようなビット数の大きな整数の乗算結果を計算するときには，分割桁数 $p = 2$ とし FFT と畳み込み定理を用いた手法が最も適していると考えられる．しか

© 2015 Information Processing Society of Japan

し，分割桁数 $p$ を大きくとると高速に計算ができるので，分割桁数を大きくとっても，畳み込み値の下位の桁が誤ることのない計算精度向上を目指した改良が必要と考えられる．

## 5. 数論変換と畳み込み定理を用いた乗算アルゴリズムの考察

DFT や FFT は複素数体上での変換であり，三角関数や円周率などの浮動小数点の計算が存在するため計算精度が低下した．そこで，複素数体上で変換していた数を有限個の元からなる体，つまり有限体上で行うフーリエ変換とみなされる数論変換 [2] を用いることを試みる．

数論変換の概要について述べる．$P$ を素数としたとき

$$\alpha^{P-1} = 1 \ (\mathrm{mod} P) \tag{15}$$

において，$N = P - 1$ を最小の指数として式 (15) を満足する整数 $\alpha$ を位数 $N$ の原始根という．

ここで, 原始根 $\alpha$ を用いた次のような変換対を考える．これを数論変換 (Number Theorem Transform;NTT) という．

$$F(k) = \sum_{n=0}^{N-1} f(n)\,\alpha^{kn} \ (\mathrm{mod} P) \tag{16}$$

$$f(n) = \frac{1}{N}\sum_{k=0}^{N-1} F(k)\,\alpha^{-kn} \ (\mathrm{mod} P) \tag{17}$$

離散フーリエ変換は $\alpha = e^{-j2\pi/N}$ のときであり，$\alpha$ を整数にしたものを数論変換では考える．$\alpha$ を 2 のべきに選ぶことができれば，$\alpha$ のべきをかける演算は 2 進数におけるビットシフトでよく，乗算なしに計算できる．

素数 $P$ や $\alpha$ の値によって，畳み込み定理が使えるときと使えないときがある．畳み込み定理が使える素数 $P$ と原始根 $\alpha$ の組み合わせの例として, 素数 $P$ を $2^b+1\,(b = 2^t)$(フェルマー数)，原始根 $\alpha = 2$ とした数論変換があり，これをフェルマー数変換 (Fermat Number Transform;FNT) と呼ぶ．

以上の変換と畳み込み定理を用いて，四則演算が定義され閉じている有限集合の中で整数を扱い，誤差が生じることもなく高速に乗算を計算できる可能性がある．

## 6. まとめと今後の課題

FFT と畳み込み定理を用いることにより，高速に計算可能であることを確認した．分割桁数 $p$ を大きくとった場合には，DFT や FFT と畳み込み定理を用いると畳み込み値の計算精度が低下することがわかった．畳み込み値の計算精度と乗算結果の計算精度は等価ではないこともわかった．

数論変換を用いることで，フーリエ変換で生じた計算精度の低下が抑えられると考えられるので，アルゴリズムを計算機に実装し，性能について検証することが今後の課題である．

## 参考文献

[1] J. W. Cooley and J. W. Tukey , “An algorithm for the machine calculation of complex Fourier series,” Math. of Comput., 19, pp. 297-301, 1965.
[2] 電子情報通信学会 編, “ディジタル信号処理ハンドブック,” オーム社, 1993.
[3] M. Fürer, “Faster Integer Multiplication,” in Proceedings of the 39th Annual ACM Symposium on Theory of Computing, pp. 57-66, 2007.
[4] パターソン, ヘネシー, “コンピュータの構成と設計 上 第 3 版,” 日経 BP 社, 2011.
[5] A. Schönhage and V. Strassen,“Schnelle Multiplikation großer Zahlen, Computing 7, pp. 281-292, 1971.
[6] 谷萩 隆嗣, “高速アルゴリズムと並列信号処理,” コロナ社, 2000.
[7] 辻井 重男, “暗号　情報セキュリティの技術と歴史,” 講談社,2012.
[8] 内田 智史, “C 言語によるプログラミング 基礎編 第 2 版,” オーム社, 2008.

© 2015 Information Processing Society of Japan