A-010

ビット数の大きな整数の乗算アルゴリズムの実験的性能評価

Experimental Performance Evaluation of a Multiplication Algorithm for Big Integers

橋本 翔太 † Shota Hashimoto　**上土井 陽子** † Yoko Kamidoi　**若林 真一** † Shin'ichi Wakabayashi

1　はじめに

近年，ビット数の大きな整数の乗算は RSA 暗号，ElGamel 暗号，楕円曲線暗号のような暗号技術に用いられている．例えば RSA 暗号は，$71 \times 97 = 6,887$ のように素数同士の乗算は簡単であるが，その逆の $6,887 = p \times q$ を満たす素数 p, q を求める素因数分解の困難性を利用した技術である．実際には，元の素数（例では $71, 97$）は十進数で約 300 桁から約 600 桁以上，二進数で数千ビット長にもなる大きな数が用いられるため，乗算した値（例では $6,887$）が公開されていても，2 つの素数に分解することはスーパーコンピュータを用いても不可能とされている [1].

上記のような暗号の公開鍵を計算す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加算を基本とするなどの乗算アルゴリズムで計算すると時間がかかってしまう．高速乗算アルゴリズムは多数存在するが，本稿ではその中でもフーリエ変換と畳み込み定理を用いた手法に注目する [2]. 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が生じる可能性がある．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，計算時間の高速化のみならず計算精度も高いことが望まれる．

本稿では畳み込み演算，フーリエ変換と畳み込み定理を利用した乗算手法に注目し，計算時間と計算精度について実験的性能評価を行う．

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 $a_i = 0$ $(N/2 \le i < N, i$ 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．

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

例えば，4 桁の 10 進数の整数 $8472 = 2 \times 10^0 + 7 \times 10^1 + 4 \times 10^2 + 8 \times 10^3$ 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．

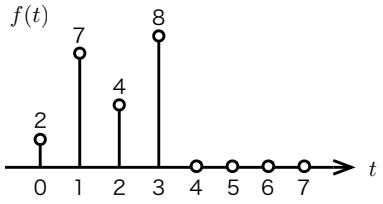

図 1: 整数を波形で表現

2.2　フーリエ変換 [4]

乗算を行う整数を図 1 のように波で表現し，その波に対し離散フーリエ変換 (DFT) 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，逆離散フーリエ変換 (IDFT) を適用する (図 2)．高速乗算アルゴリズムにおいて，フーリエ変換を適用する理由については後述する．

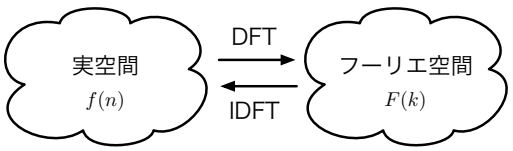

図 2: 空間の移動

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．

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

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 trans-

†広島市立大学大学院 情報科学研究科 情報工学専攻

form; 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

N 点の離散フーリエ変換を計算するには，N^2 回の積和計算が必要となる．そこで，DFT を少ない計算量で計算可能な高速フーリエ変換 (fast Fourier transform; FFT) を用いる．FFT はビット反転操作とバタフライ演算から構成されている．高速フーリエ変換を用いると $N \log N$ に比例する計算量で済む．

式 (5) に対して，$W_N = e^{-j2\pi/N} = \cos\dfrac{2\pi}{N} - j\sin\dfrac{2\pi}{N}$ 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．

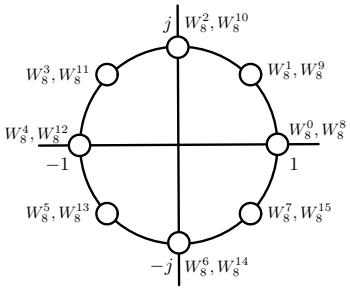

図 3: 回転因子 W_8

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．

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

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，

$$A = 14\times100^0 + 78\times100^1 + 56\times100^2 + 24\times100^3 \tag{12}$$

と表現でき ($N = 8$)，$a_i = 0\,(4 \le i < 8, i$ 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 $f(t)\,(0 \le t < 8)$ 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

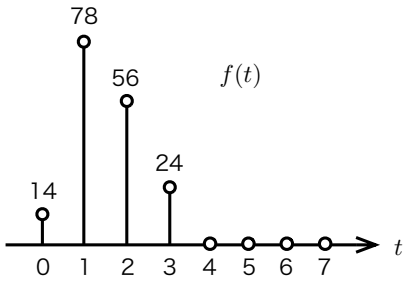

図 4: 整数 A の波による表現

数系列を時間に関する関数 $g(t)\,(0 \le t < 8)$ 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．

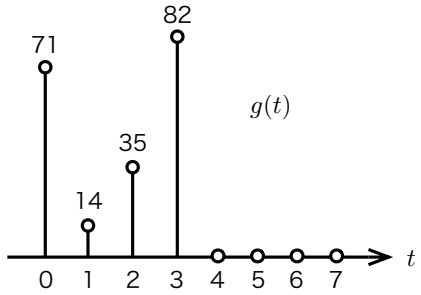

図 5: 整数 B の波による表現

3.1 筆算による乗算 [5]

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．

			24	56	78	14	
		×	82	35	14	71	
			1704	3976	5538	994	
		336	784	1092	196		
	840	1960	2730	490			畳み込み演算で得られる結果と同じ
1968	4592	6396	1148				
1968	5432	8692	6366	5558	5734	994	
		↓ 桁上げ処理 ↓					
20	23	19	56	22	15	43	94 ← 乗算結果

図 6: 筆算による乗算

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．

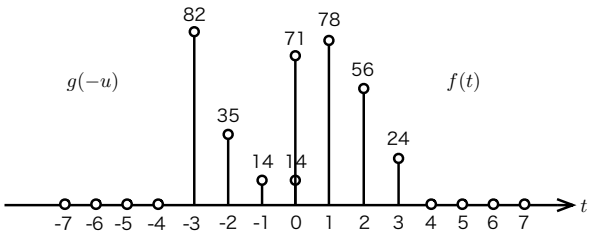

図 7: $t = 0$ のときの畳み込み演算

- $t = 1$ 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 $t = 1$ にプロットしている．$(f * g)(1) = 5734$ となる．

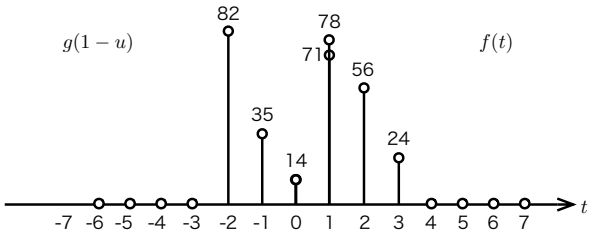

図 8: $t = 1$ のときの畳み込み演算

- $t = 7$ 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．

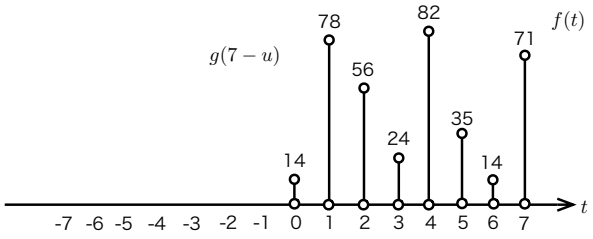

図 9: $t = 7$ のときの畳み込み演算

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．

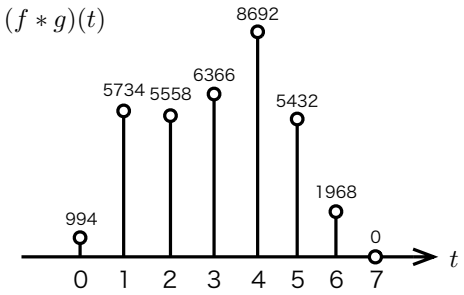

図 10: 畳み込み演算結果

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．

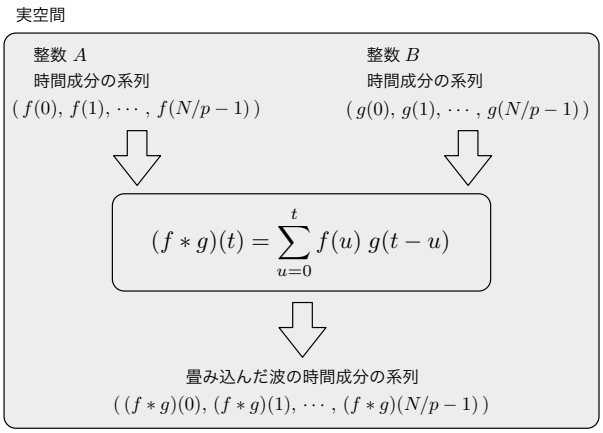

図 11: 畳み込み演算を用いた乗算アルゴリズム

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．

畳み込み演算を用いたアルゴリズムと異なり，実空間か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．

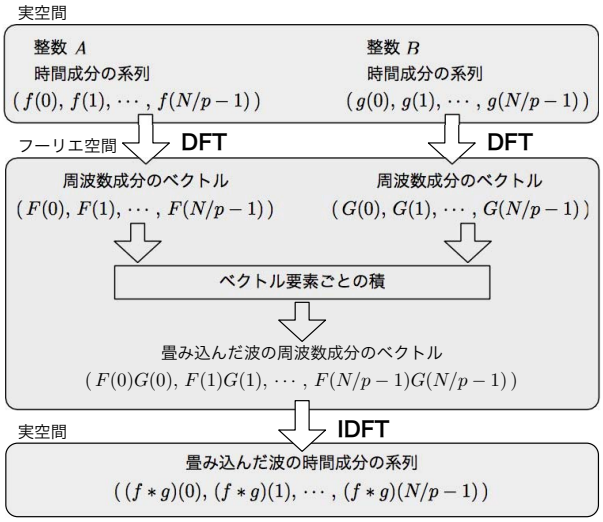

図 12: DFT と畳み込み定理を用いた乗算アルゴリズム

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．

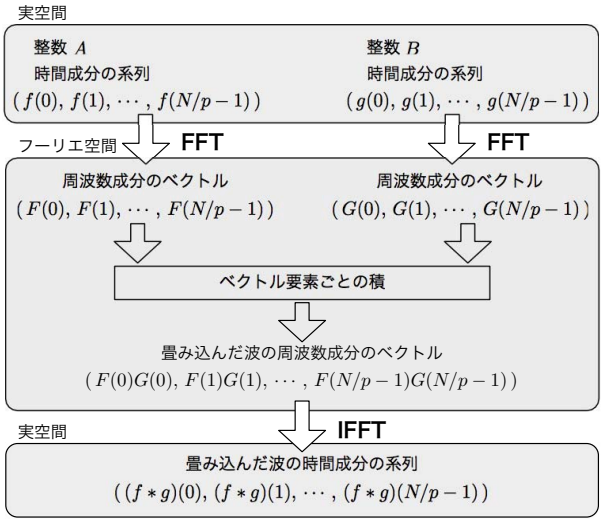

図 13: FFT と畳み込み定理を用いた乗算アルゴリズム

4 実験的性能評価

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，それぞれ単精度浮動小数点型 `float`(以降，`float` 型と表記)，倍精度浮動小数点型 `double`(以降，`double` 型と表記) の 2 つ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．

実験で用いたコンパイラのヘッダファイル `float.h` の浮動小数点型の属性を定義したマクロを確認すると，`float` 型変数が 10 進数で表すことのできる精度の桁数が 6(`FLT_DIG`) であった．分割桁数 $p = 4$ のとき畳み込み値が $4+4 = 8$ 桁以上になるため，`float` 型変数の計算精度により実装した場合は計算不可能である．一方で，`double` 型変数は 10 進数で表すことのできる精度の桁数が 15(`DBL_DIG`) であるため，分割桁数 $p = 1, 2, 4$ のいずれにおいても計算可能である．よって，`float` 型変数による実装は分割桁数 $p = 1, 2$，`double` 型変数による実装は分割桁数 $p = 1, 2, 4$ の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，整数 A, B の組み合わせ 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，`float` 型または `double` 型によって実装した場合をそれぞれ表 1，表 2，表 3 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: 畳み込み演算を用いた手法の平均計算時間

桁数 (10 進) N	計算時間 [ms]				
	$p=1$		$p=2$		$p=4$
	float	double	float	double	double
16	0.030	0.029	0.023	0.021	0.017
32	0.034	0.035	0.024	0.025	0.019
64	0.061	0.062	0.036	0.038	0.026
128	0.128	0.131	0.063	0.064	0.040
256	0.323	0.327	0.135	0.135	0.072
512	0.971	0.978	0.337	0.339	0.152
1024	3.250	3.270	0.999	1.007	0.384
2048	11.763	11.845	3.312	3.317	1.076
4096	44.636	44.250	11.924	11.935	3.468

表 2: DFT と畳み込み定理を用いた手法の平均計算時間

桁数 (10 進) N	計算時間 [ms]				
	$p=1$		$p=2$		$p=4$
	float	double	float	double	double
16	0.102	0.095	0.040	0.039	0.024
32	0.323	0.291	0.099	0.087	0.036
64	1.217	1.108	0.330	0.290	0.094
128	4.823	4.395	1.238	1.104	0.310
256	19.110	17.673	4.849	4.472	1.172
512	76.394	69.195	19.082	17.578	4.542
1024	304.624	274.389	75.624	68.871	17.903
2048	1220.510	1097.118	302.979	274.616	69.723
4096	4874.814	4380.119	1206.247	1096.042	275.499

表 3: FFT と畳み込み定理を用いた手法の平均計算時間

桁数 (10 進) N	計算時間 [ms]				
	$p=1$		$p=2$		$p=4$
	float	double	float	double	double
16	0.035	0.035	0.025	0.024	0.017
32	0.042	0.042	0.029	0.028	0.022
64	0.072	0.072	0.045	0.043	0.033
128	0.132	0.133	0.085	0.074	0.054
256	0.257	0.262	0.152	0.138	0.960
512	0.537	0.543	0.279	0.276	0.190
1024	1.118	1.121	0.573	0.565	0.388
2048	2.272	2.298	1.183	1.160	0.797
4096	5.042	4.777	2.467	2.409	1.630

最初に，`float` 型と `double` 型による変数の精度の違いが乗算の計算時間にどのように影響するかを調べる．表 1 と表 3 から，浮動小数点数の計算がない畳み込み演算を用いた手法や，DFT に比べて浮動小数点数の計算量が少ない FFT と畳み込み定理を用いた手法は，`float` 型と `double` 型の変数の精度で計算時間に差がほとんどないと言える．一方，表 2 より DFT と畳み込み定理を用いた手法の場合は，桁数 N が大きくなればなるほど `float` 型と `double` 型による変数の精度の違いで計算時間に差が生じる．$p = 1$ のとき，`float` 型に比べて `double` 型が $N = 2048$ で約 130[ms]，$N = 4096$ で約 500[ms] 高速に計算が可能であった．これは現在のコンピュータの内部処理が `double` 型を基本としていることなどの理由により，`float` 型の変数は一度 `double` 型に変換する処理があるため，一般的にビット数が `double` 型の半分であるにも関わらず，`float` 型の方が浮動小数点数の計算に時間がかかっている．そのため，精度の高さや計算時間の速さの観点においても，浮動小数点数を表現する場合には `double` 型を使用するべきであると考えられる [6]．

次に `double` 型により実装したときの桁数や分割桁数に対して，実行時間がどのような関係があるかどうか考察する．表 1，表 2，表 3 から桁数 $N = 4096$ のとき分割桁数 p に関係なく FFT と畳み込み定理を用いた手法が他の手法に比べて，最も高速に乗算の計算が可能であることがわかった．FFT と畳み込み定理を用いた手法は分割桁数 $p = 1$ のとき，畳み込み演算を用いた手法に比べて約 9 倍，DFT と畳み込み定理を用いた手法に比べて約 917 倍の高速化となった．

表 1，表 2，表 3 の各手法の `double` 型による変数の精度で実行したときの平均計算時間をグラフにしたものを図 14，図 15，図 16 に示す．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．分割桁数 $p = 4$ のときは $p = 1$ に比べて，畳み込み演算を用いた手法の場合は約 13 倍，DFT と畳み込み定理を用いた手法の場合は約 16 倍の高速化となった．分割桁数 p を 2 倍，4 倍にすると，約 4 倍，約 16 倍の高速化となっていることが分かる．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，図の縦軸の縮尺を考慮にいれたとき，桁数

増加に対して分割桁数が小さいほど計算時間の増加も急激である．桁数 N を 2 倍，4 倍にすると，計算時間は約 4 倍，約 16 倍となっているので，畳み込み演算を用いた手法，DFT と畳み込み定理を用いた手法においては，桁数の増加割合の 2 乗に比例した計算時間である．一方，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．

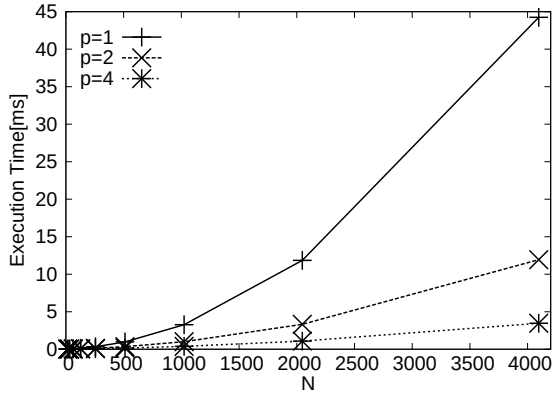

図 14: 畳み込み演算を用いた手法の平均計算時間

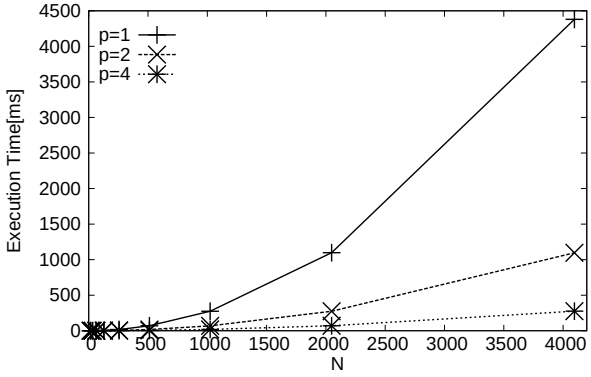

図 15: DFT と畳み込み定理を用いた手法の平均計算時間

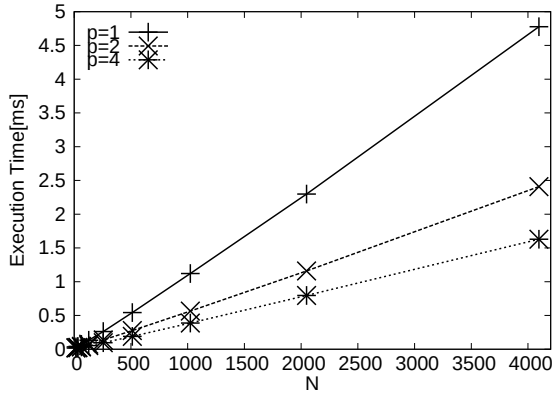

図 16: FFT と畳み込み定理を用いた手法の平均計算時間

4.2 計算精度に関する考察

4.2.1 畳み込み値の計算精度

前節では，`double` 型により実装したとき分割桁数 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．

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，`float` 型で実装した場合を表 4，`double` 型で実装した場合を表 5 に示す．表 4，5 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 4: 畳み込み値の計算精度 (float 型による実装)

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	
	$p=1$	$p=2$	$p=1$	$p=2$
16	92.50	92.50	48.75	67.50
32	93.13	96.25	54.38	65.00
64	98.44	93.13	76.88	57.50
128	98.28	95.94	67.66	77.50
256	98.75	97.97	68.20	68.91
512	99.41	98.67	70.94	68.67
1024	99.77	99.41	76.52	74.34
2048	99.84	99.71	78.73	76.66
4096	99.92	99.79	81.74	78.70

表 5: 畳み込み値の計算精度 (double 型による実装)

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 4 より `float` 型で実装した場合は，DFT と畳み込み定理を用いたとき桁数増加につれて平均誤り率も高くなり，桁数 $N = 1024$ 以上では分割桁数に関係なく，ほぼすべての畳み込み値の計算が正確でないこと

がわかった．また FFT と畳み込み定理を用いたときは，DFT と畳み込み定理を用いるよりも誤り率は抑えられているものの，桁数 $N = 1024$ 以上では約 75%以上の正確でない畳み込み計算が行われていた．これは，浮動小数点数の演算回数が多くなるほど，計算誤差が広がるからであると考えられる．このことから，フーリエ変換のように浮動小数点数の計算を必要とし，その計算結果を暗号技術に使用するなどの精度が高い計算を求められるときは，`float` 型による実装は用いるべきでないと考えられる．

次に `double` 型で実装した場合について考える．表 5 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 π を含む無理数の計算，コンパイラやプログラムの書き方によって生じるものであると考えられる．しかし，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 5 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

`double` 型で実装した場合の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平均値を表 6 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 $N = 1024$，分割桁数 $p = 4$ のとき，DFT と畳み込み定理を用いた手法と正しい乗算結果との差は少なくとも 10^{999} もあるのに対して，FFT と畳み込み定理を用いた手法の場合は正しい乗算結果との差が少なくとも 10^{653} で誤差が小さくなっている．他の桁数，分割桁数においても同様のことが言える．

しかし，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かった．例えば，畳み込み値の平均誤り率が低いとしても誤りが生じた最高桁が大きいときは，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．これらの関係については，より詳細な考察を行う必要がある．

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，`double` 型による実装を行い，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．

表 6: 乗算結果の計算精度 (double 型による実装)

桁数 (10 進) N	乗算結果の誤りが生じた最高桁 [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	×	×	56	×	×	×
128	×	×	96	×	×	×
256	×	×	252	×	×	×
512	×	×	503	×	×	129
1024	×	×	999	×	×	653
2048	×	×	1957	×	×	1245
4096	×	3603	4090	×	×	3727

5 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．

今後の課題として，本稿で得られた実験結果の正当性を示す数学的証明や，フーリエ変換を用いた場合に生じる畳み込み値の計算精度低下の原因の調査とその低減方法の開発が挙げられる．

参考文献

[1] 辻井 重男, “暗号　情報セキュリティの技術と歴史,” 講談社, 2012.

[2] M. Fürer, “Faster Integer Multiplication,” in Proceedings of the 39th Annual ACM Symposium on Theory of Computing, 2007, pp. 57-66.

[3] A. De, P. P. Kurur, C. Saha, and R. Saptharishi “Fast integer multiplication using modular arithmetic, ” SIAM J. Comput., 42, 2013, pp. 685-699.

[4] 金城 繁徳, 尾知 博, “例題で学ぶ　ディジタル信号処理,” コロナ社, 1997.

[5] 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.

[6] 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.