# 自然対数の底 e の超多数桁計算

# 自然対数の底eの超多数桁計算

濱口雄人，有末宏明

Ultra High Precision Computation of Napier's Constant

Yuto HAMAGUCHI*, Hiroaki ARISUE**

**ABSTRACT**

Napier's constant is computed in ultra high precision. Using the Divide and Rationalize method and the Fast Fourier Transform, the CPU time to calculate the constant in digits $N$ has been reduced drastically to be proportional to $N(\log N)^3$. We obtained the constant in 500,000,000 digits.

**Key Words:** Fast Fourier Transform, Newton method, Divide and Rationalize method.

## 1．はじめに

本研究は超越数の一つである自然対数の底 $e$ を高位桁まで算出することを目的とする．計算機の存在しなかった時代を考えると自然対数の高位桁が現在のような短時間かつ効率的に算出されること自体が魅力的である．同時に，自然対数の底 $e$ の小数点以下の莫大な桁数の数値そのものは円周率と同様に乱数度の極めて高い乱数列としてモンテカルロシュミレーション等に利用するといった実用的な役割が期待される．

本研究のような数値計算では，同じ結果を得るのにも最も効率的な手法を採用することが重要である．本研究では，自然対数の底を算出する過程での計算量・計算時間に着目し，それらの短縮を図ろうとするものでもある．高速フーリエ変換 (FFT) やニュートン法など，計算量・計算時間を短縮化するような技法は数値計算・その他の工学分野でも利用され，科学技術を発展させてきた．

本研究では分割有理数化法 (DRM)[1]，離散高速フーリエ変換を用いた乗算，ニュートン法を応用した除算などを，$e$ の展開式の値の計算に適用することで $e$ の高位桁計算を行う．今回の研究では，単体の PC 上での計算に限り，算出桁数は小数点以下約 5 億桁を目標とする．計算量・計算時間が最も最少となるような計算アルゴリズム，計算結果の正しさの検証方法等についての研究を行う．

2008 年 4 月 9 日 受理

* 明石高専専攻科　機械・電子システム工学専攻
(Advanced Course of Akashi National College of technology, Mechanical and Electronic System Engineering)
** 総合工学システム学科　機械システムコース
(Dept. of Industrial Systems Engineering : Mechanical Systems Course)

## 2．理論

### 2.1 自然対数の底

自然対数の底を計算するには，次の一般的な $e^x$ のマクローリン展開

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!} + \cdots$$

において $x = 1$ とした次式を用いた．

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \cdots + \frac{1}{n!} + O\left(\frac{1}{(n+1)!}\right) \quad (1)$$

さらにこれを通分して有理数型に変形する．

$$e - 1 = \frac{1 + \sum_{i=2}^{n} n(n-1)\cdots(n-i+2)}{n!} \quad (2)$$

式 (1) を見ても分かるように，$n$ が大きくなればなるほど $e$ の精度は高くなる．例えば 1 億桁まで計算しようとする場合，必要な $n$ の大きさは約 15,000,000 である．

### 2.2 多倍長演算の理論

**2.2.1 多倍長化** 任意の桁数の整数 $k$ は整数配列 $a(i), (i = 0, n_{max})$ を用いて，

$$k = \sum_{i=0}^{n_{max}} a(i) M^i \quad (3)$$

と表現できる．ここで $M$ は任意の整数で，進数表記で，(3) 式はいわば $M$ 進数表記と呼ぶべきものである．また，任意の実数 $x$ は同じく整数配列 $a(i)$ を用いて，

$$x = M^{-N_x} \sum_{i=0}^{n_{max}} a(i) M^i \quad (4)$$

と表現できる．例えば$x = 2.71828182845904523536028 7$については，例えば$M = 1000$ととれば$a(7) = 718$, $a(6) = 281$, $a(5) = 828$, $a(4) = 459$, $a(3) = 45$, $a(2) = 235$, $a(1) = 360$,$a(0) = 287$, および$N_x = -7$で表現できる．$e$の値を5億桁の精度で求めるためには，後述する理由により，$M = 10000$とし，$a(i)$の各要素は2バイト整数とした．

### 2.2.2 多倍長演算と繰上げ

図1は，整数どうしの加算の例である．ここで$M = 10000$としている．配列$a$+配列$b$を行っているが，これでは結果$c$の各配列要素が$M$を超えてしまう．図2のように下位桁から順に繰上げ操作を行う．$j$は繰上げ分で，一つ上位の配列へ繰上げ分を加算する．ここで繰上げ操作で除算を用いているが，2バイト整数を対象としたものであるので計算量としてはさほど大きくはならず，$N$桁の整数どうしの加算であれば計算時間はほぼ$N$に比例したものとなる．また最後の繰上げの際は，繰上げ先が配列の定義外の領域であることが多く，配列要素数の宣言を1つ多く取るなどの注意が必要である．

| | | | |
|---|---|---|---|
| a | 1111 | 1111 | 1111 |
| + | + | + | + |
| b | 1919 | 9191 | 9999 |
| ‖ | ‖ | ‖ | ‖ |
| c | 3030 | 10302 | 11110 |
| ⇒ | 3031 | 0303 | 1110 |

**図1 多倍長加算**

$$\mathbf{c' = c[i] /10000}$$

$$\mathbf{c[i] = c[i] - c' * 10000}$$

$$\mathbf{c[i+1] = c[i+1] + c'}$$

**図2 多倍長加算**

| | | | |
|---|---|---|---|
| a | 1111 | 0101 | 1010 |
| \| | \| | \| | \| |
| b | 9999 | 1010 | 0101 |
| ‖ | ‖ | ‖ | ‖ |
| c | −8888 | −0909 | 0909 |
| c | −8888 | −0908 | −9091 |

**図3 多倍長減算**

減算を行う場合は図3のように，先ずは配列要素間で減算を行う．その場合，結果の$c$の各配列要素の符号は一般には正号と負号が混在する．

$$\mathbf{c' = c[i+1] * 10000 + c[i]}$$

$$\mathbf{c'' = c'/10000}$$

$$\mathbf{c[i+1] = c''}$$

$$\mathbf{c[i] = c' - c'' * 10000}$$

**図4 符号の統一**

これらの符号を統一するために，図4のように上位桁から順に符号の統一演算を実施する．この場合も加算と同様に計算時間は桁数$N$に比例する．

| | | | |
|---|---|---|---|
| | a[2] | a[1] | a[0] |
| × | b[2] | b[1] | b[0] |
| − | | | |
| | | | + a[0] × b[0] |
| | | + a[1] × b[0] | |
| | + a[2] × b[0] | | |
| | | + a[0] × b[1] | |
| | + a[1] × b[1] | | |
| + | a[2] × b[1] | | |
| | | + a[0] × b[2] | |
| | + a[1] × b[2] | | |
| + | a[2] × b[2] | | |
| − | | | |
| | c[5] | c[4] | c[3] | c[2] | c[1] | c[0] |

**図5 多倍長乗算**

次に，乗算の方法について述べる．乗算は図5のようにたすき掛けを行う．途中結果である3行目から11行目は配列 $c$ に加算される．$a$,$b$ の各配列要素の大きさは $M$ 未満であり，それらの積は $M^2$ 未満である．$c$ の対応する配列への加算時には毎回繰り上げる．これらの計算は $a$,$b$ の各配列要素が2バイト整数なら4バイト整数演算で対応できる．この乗算の計算時間は $a$,$b$ の桁数をそれぞれ $N$ として $N^2$ に比例する．

## 3 計算アルゴリズム

### 3.1 離散高速フーリエ変換

#### 3.1.1 離散フーリエ変換 (DFT)

$z = x \times y$ を行うとき，入力値 $x, y$ は連続なものではなく離散的な数値であるため，それぞれ $x, y$ に対してまず離散フーリエ変換 (Discrete Fourier Transform) を行う．

$n$ 個の入力値 $x(n)$ に対する離散フーリエ変換を $X(m)$

$$X(m) = \sum_{n=0}^{N-1} x(n) \cdot \omega^{-m*n} \qquad (5)$$

$$(m = 0, 1, 2, \cdots, N-1)$$

$n$ 個の入力値 $y(n)$ に対する離散フーリエ変換を $Y(m)$

$$Y(m) = \sum_{n=0}^{N-1} y(n) \cdot \omega^{-m*n} \qquad (6)$$

$$(m = 0, 1, 2, \cdots, N-1)$$

で定義する．ここで $\omega$ は，

$$\omega = e^{\frac{2\pi i}{N}} \quad (i \text{ は虚数単位}) \qquad (7)$$

とする．オイラーの公式より，

$$\omega = \cos(\frac{2\pi}{N}) + i \sin(\frac{2\pi}{N})$$

のようにも表記できる．

$\omega$ は回転子とも呼ばれ，次の性質を持つ．

$$\omega^N = \omega^0 = 1 \qquad (8)$$

次に，

$$Z(m) = X(m) \times Y(m) \quad (m = 0, 1, 2, \cdots, N-1) \qquad (9)$$

と定義すると，

$$Z(m) = \left\{ \sum_{n=0}^{N-1} x(n) \cdot \omega^{-m*n} \right\} \times \left\{ \sum_{n=0}^{N-1} y(n) \cdot \omega^{-m*n} \right\}$$

これを展開すると，

$$Z(m) =$$

$$\{x(0)\cdot\omega^{-m*0} + x(1)\cdot\omega^{-m*1} + x(2)\cdot\omega^{-m*2} \cdots x(N-1)\cdot\omega^{-m*(N-1)}\}$$

$$\times\{y(0)\cdot\omega^{-m*0} + y(1)\cdot\omega^{-m*1} + y(2)\cdot\omega^{-m*2} \cdots y(N-1)\cdot\omega^{-m*(N-1)}\}$$

$$=$$

$$\omega^{-m*0}\{x(0)\cdot y(0) + x(1)\cdot y(N-1) + x(2)\cdot y(N-2) + \cdots + x(N-1)\cdot y(1)$$

$$+\omega^{-m*1}\{x(0)\cdot y(1) + x(1)\cdot y(0) + x(2)\cdot y(N-1) + \cdots + x(N-1)\cdot y(2)$$

$$+\omega^{-m*2}\{x(0)\cdot y(2) + x(1)\cdot y(1) + x(2)\cdot y(0) + x(3)\cdot y(N-1) +$$
$$\cdots + x(N-1)\cdot y(3)$$
$$\vdots$$
$$+\omega^{-m*(N-1)}\{x(0)\cdot y(N-1) + x(1)\cdot y(N-2) + x(2)\cdot y(N-3) +$$
$$\cdots + x(N-1)\cdot y(0)$$

まとめると，

$$Z(m) = \sum_{n=0}^{N-1} \omega^{-m*n} \times z(n)$$

ただし，

$$z(n) = \sum_{i=0}^{N-1} x(i) \cdot y((n-i)\ mod\ N)$$

ここで，

$$x(n) = y(n) = 0 \quad (n = N/2, \cdots, N-1) \qquad (10)$$

と置くと，

$$z(n) = \sum_{i=0}^{N/2-1} x(i) \cdot y(n-i) \qquad (11)$$

すなわち，$x(n)$ $(n = 0, \cdots, N/2)$ と $y(n)$ $(n = 0, \cdots, N/2)$ で表される多倍長整数について，$x, y$ を (10) のように拡張した上で，それぞれ離散フーリエ変換させ，その同位置の積をとった後に，逆変換することにより，$x \times y$ の乗算結果を得ることができる．従って，計算の手順としては

(i) 2入力値の離散フーリエ変換
(ii) (i) の同位置について乗算
(iii) (ii) で得た結果の逆離散フーリエ変換

となる．(ii) では「乗算」としているが，これは同位置 (同じ要素) についての乗算のみであるので計算時間は $N$ に比例する．

### 3.1.2 高速フーリエ変換 (FFT)

しかし，この手法はこのままでは，離散フーリエ変換の計算時間は $N^2$ に比例するため，$N$ 桁どうしの乗算全体としても計算時間は $N^2$ に比例する．そこで離散フーリエ変換に高速フーリエ変換 (Fast Fourier transform) を取り入れる．

式 (3) の離散フーリエ変換の定義式において，$\omega = e^{-\frac{2\pi i}{N}}$ としたの上で，行列化したものを以下に示す．

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ \vdots \\ X(N-1) \\ X(N) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \cdots & \omega^0 \\ \omega^0 & \omega^1 & \cdots & \omega^N \\ \omega^0 & \omega^2 & \cdots & \omega^{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \omega^0 & \omega^{N-1} & \cdots & \omega^{(N-1)N} \\ \omega^0 & \omega^N & \cdots & \omega^{N^2} \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ \vdots \\ x(N-1) \\ x(N) \end{bmatrix}$$

ここでは簡単のため，$x$ についてのみの離散フーリエ変換を示し，要素 (周期)$N = 4$ とする．

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

周期 $N = 4$ であるから，$\omega^4 = \omega^0$ と書き換えると，

$$\begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^0 & \omega^2 \\ \omega^0 & \omega^3 & \omega^2 & \omega^5 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}$$

となり，さらに $\omega$ の行列を中心で区切ると，奇数行では左右が等しく，偶数行では左右は半周期ずれた関係となっている．括り出すと

$$\begin{bmatrix} x'(0) \\ x'(2) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^0 \end{bmatrix} \begin{bmatrix} x(0) \\ x(2) \end{bmatrix}$$

$$\begin{bmatrix} x'(1) \\ x'(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^2 \end{bmatrix} \begin{bmatrix} x(1) \\ x(3) \end{bmatrix}$$

$$\begin{bmatrix} X(0) \\ X(2) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^2 \end{bmatrix} \begin{bmatrix} x'(0) \\ x'(2) \end{bmatrix}$$

$$\begin{bmatrix} X(1) \\ X(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^1 \\ \omega^0 & \omega^3 \end{bmatrix} \begin{bmatrix} x'(1) \\ x'(3) \end{bmatrix}$$

が得られる．すなわち，高速フーリエ変換を用いると離散フーリエ変換が一般の $N$ について $N/2$ 個の 2 行 2 列の行列計算を $\log_2 N$ 回繰り返すことに帰着し，計算時間は $N \log_2 N$ に比例する．なお，高速フーリエ変換にも制約がある．配列要素数 $N$ が 2 の冪乗でないといけないことである．離散高速フーリエ変換を用いて多倍長数の乗算を実行する場合には，多倍長数の配列要素数はその $N$ に対して $N/2$ 以下なければならないが，それがちょうど 2 の冪乗で与えられているときよりも効率は低くなる．

### 3.1.3 FFT サブルーチン

(11) 式において，$z(n)$ $(n = 0, \cdots, N)$ は $x(i)$ $(i = 0, \cdots, N/2-1)$ と $y(n-i)$ $(i = 0, \cdots, N/2-1)$ の積が $i = 0, N/2-1$ の範囲で加算されている．また配列要素 $x(i), y(i)$ はそれぞれ最大値が $M$ の量である．したがって $z(n)$ は値として最大 $M^2N/2$ をとり得る．FFT サブルーチンにおいて $z(i)$ を 8 バイト実数 (倍精度実数) にすると，それが表現できる最大桁数は 16 桁である．自然対数の底を 5 億の桁まで求めるには $N$ は最大 $10^8$ までとる必要がある．よって，$M$ は 10000 とするのが適切である．これ以上の値にとると，$z(i)$ の計算結果が溢れてしまう．

### 3.1.4 乗算における計算時間の比較

積計算で高速フーリエ変換を使う場合と使わない場合の計算時間を比較する目的で．入力値 N としては $N = 2^m - 1$ を与え，それを 2 乗させた．CPU 時間の計測には runtimc.cxc を用いた．

**表1 CPU time for N*N**

| N | FFT[s] | 比率 | Normal[s] | 比率 |
|---|---|---|---|---|
| $2^{10}-1$ | 0.062 | - | 0.031 | - |
| $2^{12}-1$ | 0.062 | 1 | 0.188 | 6 |
| $2^{14}-1$ | 0.109 | 2 | 2.344 | 12 |
| $2^{16}-1$ | 0.562 | 5 | 36.843 | 16 |
| $2^{18}-1$ | 2.718 | 5 | 589.203 | 16 |
| $2^{20}-1$ | 12.375 | 5 | × | - |
| $2^{22}-1$ | 52.609 | 4 | × | - |
| $2^{24}-1$ | 798.375 | 15 | × | - |

表中の「比率」は，$N$ を増やしたときの計算時間の増加割合を示す．単純な乗算方法である「Normal」の $2^{20}$ 以降での計算は，時間過多のために中断した．この結果から計算時間は高速フーリエ変換でほぼ $N \log N$ に比例し，単純な乗算では $N^2$ に比例することを確認するこ

とができた．また高速フーリエ変換を用いた場合では，複素数による配列を多数宣言するためメモリ消費は $2^{22}$ では問題なかったが $2^{24}$ 付近からメモリ消費が 1GB に達しており，そのために CPU 時間が 15 倍も急激に増加したものと考えられる．これを回避するためにメモリの動的宣言を取り入れたり，使用する配列の共有や削減，乗算を分割して高速フーリエ変換を小さい規模で行う，などによる総メモリ使用量の軽減が考えられる．

### 3.2 分割有理数化法

次に分割有理数化法 (Divide and Rationalize Method: DRM) による計算方法について述べる．これは，式 (1) のような数列をトーナメント式に通分を行い，かつそこに含まれる乗算全てに高速フーリエ変換を適用することで計算時間の短縮を図ろうとするものである．計算時間の短縮のメカニズムはトーナメント式に通分を行うことと，高速フーリエ変換を乗算に用いることによる相乗効果によるものである．式 (1) の分母と分子を従来は別々に計算していたが，この方法を用いることにより，重複する計算を行わずに済むため，従来に比べ計算量・計算時間においても非常に効率の良い方法であると言える．

#### 3.2.1 DRM のアルゴリズム

図 6 に $n!$ のトーナメント計算を，図 7 に分子のトーナメント計算を示した．式 (1) おいて $n = 2^3 = 8$ とした

$$e - 1 = \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \frac{1}{6!} + \frac{1}{7!} + \frac{1}{8!}$$

の各項隣り合う組どうしで通分を行う．

1st round

$$p[1] = 1,\ p[2] = 2,\ p[3] = 3,\ p[4] = 4,$$
$$p[5] = 5,\ p[6] = 6,\ p[7] = 7,\ p[8] = 8$$

2nd round

$$p'[1] = p[1] \times p[2] = 2,\quad p'[2] = p[3] \times p[4] = 12,$$
$$p'[3] = p[5] \times p[6] = 30,\quad p'[4] = p[7] \times p[8] = 56$$

3rd round

$$p''[1] = p'[1] \times p'[2] = 24,$$
$$p''[2] = p'[3] \times p'[4] = 1680$$

4th round

$$p'''[1] = p''[1] \times p''[2] = 40320$$

**図 6 分母 ($n!$) のトーナメント計算 [$n = 8$]**

1st round

$$q[1] = 1,\ q[2] = 1,\ q[3] = 1,\ q[4] = 1,$$
$$q[5] = 1,\ q[6] = 1,\ q[7] = 1,\ q[8] = 1$$

2nd round

$$\frac{2\times1+1}{2!} + \frac{4\times1+1}{4!} + \frac{6\times1+1}{6!} + \frac{8\times1+1}{8!}$$

$$= \frac{p[2]\times q[1]+q[2]}{2!} + \frac{p[4]\times q[3]+q[4]}{4!}$$
$$+ \frac{p[6]\times q[5]+q[6]}{6!} + \frac{p[8]\times q[7]+q[8]}{8!}$$

$$= \frac{q'[0]}{2!} + \frac{q'[1]}{4!} + \frac{q'[2]}{6!} + \frac{q'[3]}{8!}$$

3rd round

$$= \frac{4\times3\times q'[0]+q'[1]}{4!} + \frac{8\times7\times q'[2]+q'[3]}{8!}$$

$$= \frac{p'[1]\times q'[0]+q'[1]}{4!} + \frac{p'[3]\times q'[2]+q'[3]}{8!}$$

$$= \frac{q''[1]}{4!} + \frac{q''[2]}{8!}$$

4th round

$$= \frac{8\times7\times6\times5\times q''[1]+q''[2]}{8!}$$

$$= \frac{p''[1]\times q''[1]+q''[2]}{8!}$$

$$= \frac{q'''[1]}{8!}$$

**図 7 分子のトーナメント計算 [$n = 8$]**

図を見ても分かるように図 6 の 1st round 終了後，図 7 の 2nd round を行い，再び図 6 に戻り 2nd round を開始する．それが終了すれば図 7 の 3rd round を行い，…と，全ての組が 1 つになるまでループさせる．図 6 と図 7 を見ると，分母の n!計算で使用した配列 $p$ の値を分子の計算で再利用していることが分かる．この再利用により重複する乗算を省くことが可能となり，計算効率が高まるのである．もちろん全ての乗算の計算には高速フーリエ変換を用いる．

### 3.2.2 DRMの計算量

次にDRMの計算量について述べる．計算のトーナメント「1st round」において，最後の数列2項の乗算の大きさは，最大で$n^2$程であり，同「2nd round」においては，最大$n^4$程度の大きさとなる．一般に「$i$-th round」では

$$n^{2^i}$$

の大きさとなる．その桁数$N$は，

$$N = 2^i \log_{10} n.$$

さらに，1組の乗算における計算時間$t$は，FFTを用いると一般に$N \log N$に比例し，

$$t \propto (2^i \log n) \cdot \log(2^i \log n)$$

と表記できる．$n$が十分大きい時，

$$t \propto 2^i i \log n$$

となる．最終ラウンドを$m$-th round ($m = \log_2 n$)とすると，$i$-th roundでの全ての組数($2^{m-i}$)の計算時間$T_i$は，

$$T_i \propto 2^{m-i} \cdot 2^i i \log n \propto 2^m \cdot i \log n.$$

よって1st roundからfinal roundまでの全計算時間$T$は，

$$\begin{aligned} T &\propto \sum_{i=1}^{m} 2^m i \log n \\ &\sim 2^m \frac{m^2}{2} \log n \propto n(\log n)^3 \end{aligned} \quad (12)$$

このようにDRMとFFTを併用した場合の通分に要する計算時間は，$n(\log n)^3$に比例する．

**表2 CPU time for DRM with FFT**

| $n$ | CPU time[s] | 比率 | $n(logn)^3$ の比率 |
|---|---|---|---|
| $2^{13}$ | 0.406 | - | - |
| $2^{14}$ | 1.107 | 2.73 | 2.50 |
| $2^{15}$ | 2.668 | 2.41 | 2.46 |
| $2^{16}$ | 6.099 | 2.29 | 2.43 |
| $2^{17}$ | 13.588 | 2.23 | 2.40 |
| $2^{18}$ | 30.233 | 2.22 | 2.37 |
| $2^{19}$ | 67.142 | 2.22 | 2.35 |

表2は，DRMとFFTを併用した場合の通分に要する計算時間の比較を検証したものである．$n$の増え方に対して計算時間はほぼ理論の予測通りの振舞いを示している．

### 3.3 ニュートン法

自然対数の底$e$を与える式(2)の右辺は多倍長整数どうしの除算である．これを実行する最も効率の良い方法は，分母を$a$，分子を$b$として，$b/a$=$b$×{1/a}すなわちまず$1/a$をニュートン法により多倍長小数として求め，これと分母$b$との積をとることである．

ニュートン法(Newton Method)とは，一般に方程式を数値計算によって解くための反復解法の一つである．収束の速さは2次収束と速く，古くから数値計算で用いられているアルゴリズムである．反復法としては二分法なども存在するが，本研究の目的には収束の速さから，ニュートン法が効率的である．

#### 3.3.1 漸化式の導出

$1/n!$を求めたければ，$a = n!$と置き，

$$f(x) = (1/x) - a = 0 \quad (13)$$

を満たす$x$を求めればよい．

$$f'(x) = -1/x^2$$

傾きは，

$$f'(x) = \frac{f(x_n)}{x_n - x_{n+1}} \quad (14)$$

で与えられ，$f(x), f'(x)$を代入し，

$$1/x_n^2 = -\frac{(1/x_n) - a}{x_n - x_{n+1}}$$

これを整理し，

$$x_{n+1} = x_n - x_n(ax_n - 1) \quad (15)$$

を得る．

式(15)より分かるように，ニュートン法を用いることで除算を，加減算と乗算で置き換えることが可能になる．

#### 3.3.2 ループの制御方法

ニュートン法において，分母の逆数を小数化するが，ここで必要な小数点以下の精度は$n!$の桁数，

$$\log_{10} n! \sim \frac{n \log(n) - n}{\log(10)} \quad (16)$$

に等しい．従って，ニュートン法のループにおいて，$x$ の値の収束した桁数が (16) の値を超えた段階で，ループは終了させる．

### 3.3.3 計算量と収束性

式 (13) により，まず初期値 $x_0$ として $1/a$ の適切な近似値を代入し，漸化式の反復を繰り返すことにより，$1/n!$ を $n!$ の桁数の精度まで求めることが可能である．しかし初期値の精度が良く，適当なものでないと収束が別の方向に進み，時間もかかってしまうことがある．初期値の精度を 12 桁程度にしてニュートン法の反復を開始した．ニュートン法の収束桁数はループを 1 度回すたびに 2 倍になる．

式 (15) より，多倍長除算を加算・減算・乗算のみで表現することが可能となった．計算時間としては，加算・減算は桁数 $n$ に比例し，乗算は高速フーリエ変換を用いることで $n\log n$ に比例する．従って，ニュートン法における計算時間としては最大でも $n\log n$ に比例する．表 3 はニュートン法の収束性についての表である．$n = 10^4$ で実行したとき $n!$ の逆数の出力桁数は小数点以下約 35656 となり，その値までの収束桁数はループごとに正確に 2 倍で伸びている．

**表 3 Convergence behavior in Newton loop**

| ループ回数 | 収束桁数 | 収束桁数比率 |
|---|---|---|
| 1 | 12 | - |
| 2 | 20 | 1.7 |
| 3 | 36 | 1.8 |
| 4 | 68 | 1.9 |
| 5 | 128 | 1.9 |
| 6 | 492 | 3.8 |
| 7 | 980 | 2.0 |
| 8 | 1952 | 2.0 |
| 9 | 3900 | 2.0 |
| 10 | 7796 | 2.0 |
| 11 | 15588 | 2.0 |
| 12 | 31168 | 2.0 |
| 13 | 35656 | - |

## 4．値の検証

算出した値の正確さについては，(2) 式から求めた値と，これとは独立に以下の式から求めた値とを比較して，両者が所与の桁数まで完全に一致することで確かめる．すなわち，

$$1/e = 1 - \frac{1}{1!} + \frac{1}{2!} - \cdots - \frac{1}{(n-1)!} + \frac{1}{n!} + O\left(\frac{1}{(n+1)!}\right) \quad (n : \text{even}) \tag{17}$$

を利用する．整理して

$$1/e = \frac{\sum_{i=1}^{n-1}(-1)^i n(n-1)\cdots(n-i+1)}{n!} \tag{18}$$

したがって $e$ の値は，

$$e = \frac{n!}{\sum_{i=1}^{n-1}(-1)^i n(n-1)\cdots(n-i+1)} \tag{19}$$

によっても独立に与えられる．この方法を用いて計算した結果と (2) 式から求めた結果とを全桁にわたって比較して値の正確さを検証した．

## 5．結果と考察

### 5.1 自然対数の底

(2) 式および (19) 式において $n = 2^{26}$ ととることで，自然対数の底 $e$ の値を小数点以下 496,101,200 桁まで求めた．(2) 式の結果と (19) 式の結果はこの桁数まで完全に一致している．

$e =$

小数点以下 100 桁まで

2.71828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642 74

小数点以下 999,901 桁 ～ 1,000,000 桁

6339906983460881918177801985474703355659596098930543011992358063149337865286220013798176447694228188

小数点以下 9,999,901 桁 ～ 10,000,000 桁

4572409290546935887832818194917834507601852037034894317635589923616102851120105544429298561396705376

小数点以下 19,999,901 桁 ～ 20,000,000 桁

0502153630112559627022242938445315856686545075208422779052672712563230048612758064085962801109783389

小数点以下 29,999,901 桁 ～ 30,000,000 桁

9573958454871637455369990016911555798514891835624499928626898494083887062043072252526069601927093065

小数点以下 39,999,901 桁 ～ 40,000,000 桁
06713836293235227138898470130169056179923612729148990712344214070011115854081948855760811772598789 57

小数点以下 49,999,901 桁 ～ 50,000,000 桁
94063685479684197768717869572994275233401490096926302800588506917390518731136989537663909176782564 60

小数点以下 99,999,901 桁 ～ 100,000,000 桁
72940376929296168793765656664305283347101616837403289058548401913656302051190590829606283144921182 02

小数点以下 199,999,901 桁 ～ 200,000,000 桁
71679417783467765044563877501706499044178124450480957110439780715384393574948779245263089573455088 33

小数点以下 299,999,901 桁 ～ 300,000,000 桁
28640708549262682265147446712761568601398081237155550336407165272893241054208970932530185017782475 20

小数点以下 399,999,901 桁 ～ 400,000,000 桁
35126376287858301575175152886067032472826054506397262533204216016965461962426514438486119911744771 81

小数点以下 496,101,101 桁 ～ 496,101,200 桁
81032007440746601939795683516440442665610122323282240996008440935606027054188884035522410707836686 44

この計算では，必要な最大桁数の乗算を実行するのに，乗数，被乗数のそれぞれを 32 分割し各部分の乗算に FFT サブルーチンを使って実行した．これにより，主プログラムで内で (2) 式，(19) 式の分子・分母や分母の逆数を保持しまたそれらを求めるための DRM とニュートン法の作業領域として使う主メモリが約 1.0GB，FFT サブルーチンで使う主メモリが約 0.5GB，必要な全主メモリが約 1.5GB で実行することができた．このような乗数，被乗数の分割を行わなければ，計算時間は 32 分の 1 にできるが，FFT サブルーチンで使う主メモリは 32×.5GB=16GB と大きくなってしまい，通常の PC では実行不可能である．上記の結果を得るのに要した計算時間は Pentium4(3GHz) の PC で (2) 式，(19) 式についてそれぞれ約 45 時間ずつである．

### 5.2 各数値の発生率

表 4 数値発生率

| 数値 | 個数 | 割合 |
|---|---|---|
| 0 | 276472 | 0.0997 |
| 1 | 277381 | 0.1001 |
| 2 | 276952 | 0.0999 |
| 3 | 277153 | 0.1000 |
| 4 | 277173 | 0.1000 |
| 5 | 277141 | 0.1000 |
| 6 | 277549 | 0.1001 |
| 7 | 276589 | 0.0998 |
| 8 | 277508 | 0.1001 |
| 9 | 277096 | 0.0999 |

表 4 は小数点以下算出桁数 277 万 1014 桁の各数値の発生率を示したものである．この結果から各数値がほぼ均等に誤差 ±0.0003 の範囲で個数割合がほぼ 10 分割されて発生していることが分かる．これらの値のばらつき方を見ても自然対数の底を用いて乱数列を生成させることができそうである．乱数としての検定は今後の課題である．

## 6．結論

本研究「自然対数の底の超多数桁計算」として，以下の成果を得ることができた．

- DRM に高速フーリエ変換を用いることで計算時間が $N^2$ からほぼ $N(\log N)^3$ へ大幅に軽減することが確認できた．
- 計算アルゴリズム全てに高速フーリエ変換を組み込むことで，単体の PC を用いて自然対数の底を約 5 億桁まで計算することに成功した．
- それらの値が正しい数値であることを 4.1 節の方法により確認した．

### 参考文献

1) 後保範，金田康正，高橋大介，「級数に基づく多数桁計算の演算量削減を実現する分割有理数化法」，情報処理学会論文誌 Vol.41 No.6 (2000)