Gauss-Legendre法による超高速円周率計算

2024年1月3日
Tokieda Yukinobu

この20年の間に, コンピュータによる円周率計算は大きな進化を遂げた。それまで主流だった級数展開にかわり, 算術幾何平均を用いた計算によって計算桁を飛躍的にのばすことができたからである。それを可能にした新アルゴリズムは, 1976年にSalaminとBrentがそれぞれ独立に提出した論文の中で提案された。このアルゴリズムはGauss-Legendre法とよばれ, 1980年以降, 円周率計算の記録を更新し続けている。本書では、そのアルゴリズムの導出方法と, アルゴリズムの収束性について示す。

1 Gauss-Legendre法

Gauss-Legendre法は, 算術幾何平均を用いた円周率の高速計算アルゴリズムである。この節では, Gauss-Legendreアルゴリズムの基本形を紹介する。

初期値として, $a_0 = 1$, $b = 1/\sqrt{2}$ を与え, 次の漸化式によって系列 a_n, b_n を決定する。

$$a_{n+1} = \frac{a_n + b_n}{2}, \qquad b_{n+1} = \sqrt{a_n b_n}.$$

この系列から派生する系列:

$$\pi_n = \frac{2a_n^2}{1 - \sum_{i=0}^{n} 2^i (a_i^2 - b_i^2)},$$

は急速に円周率 π に収束する。この系列の収束速度は2次の次数をもっているため, n が1つ増えるごとに有効桁数が2倍になる。Machinの公式などの級数展開を利用した公式では, 計算桁の2乗に比例して計算時間が増大していたが, Gauss-Legendre法は計算桁が2倍になっても計算時間はたかだか2.3倍程度にしかならない。

級数展開に比べ複雑な処理を要するため, 少ない桁数では効率的な方法とはいえないが, 膨大な桁数の計算では非常に高速な計算を実現する。最近では, この基本公式をさらに変形して, さらなる高速化がなされている。このアルゴリズムは, 楕円積分とよばれる初等数学の域を超えた分野の副産物として導き出された公式である。

2　楕円積分

楕円積分とよばれる数学の分野が Gauss-Legendre 法の発見のきっかけとなった。特に公式の発見に直接関係のあるものは, 第一種楕円積分 $F(k,\varphi)$ と第二種楕円積分 $E(k,\varphi)$ である。これらは,

$$F(k,\varphi) = \int_0^{\varphi} \frac{\mathrm{d}\theta}{\sqrt{1-k^2\sin^2\theta}}, \tag{1}$$

$$E(k,\varphi) = \int_0^{\varphi} \sqrt{1-k^2\sin^2\theta}\,\mathrm{d}\theta, \tag{2}$$

で定義されており, k は楕円積分の母数とよばれる。この k が 0 または 1 であれば, これらの積分を実行することができるが, $0 < k < 1$ なる任意の値については, 置換などの手段で積分を解析的に実行することはできない。

3　Landen 変換

第一種と第二種の楕円積分は, 古くから物理学にも現れており, しかも, 比較的単純な形をしているため, 級数展開によって計算されることが多い。一方, Landen 変換というまったく異なったアプローチで楕円積分を計算することもできる。この節では, Landen 変換による完全楕円積分の計算手法を導出する。

楕円積分の母数 k と, 積分変数 φ について

$$k_1 = \frac{1-k'}{1+k'}, \tag{3}$$

$$\tan(\varphi_1 - \varphi) = k'\tan\varphi, \tag{4}$$

を用いて, 新たな母数 k_1 と積分変数 φ_1 を定義する。ここで, $k' \equiv \sqrt{1-k^2}$ は楕円関数の補母数である。この変換は Landen 変換とよばれる。後の計算でも使うため, (3) を逆変換して,

$$k' = \frac{1-k_1}{1+k_1}, \qquad k = \frac{2\sqrt{k_1}}{1+k_1},$$

となることを注意しておく。

この変換式から楕円積分計算に必要な性質を抽出していこう。変換式 (4) から即座に,

$$\sin(\varphi_1-\varphi) = \frac{k'\sin\varphi}{\sqrt{1-k^2\sin^2\varphi}}, \qquad \cos(\varphi_1-\varphi) = \frac{\cos\varphi}{\sqrt{1-k^2\sin^2\varphi}},$$

が導かれるが, これに加法定理を適用すると,

$$\sin(2\varphi-\varphi_1)=\frac{(1-k')\sin\varphi\cos\varphi}{\sqrt{1-k^2\sin^2\varphi}},\qquad \sin\varphi_1=\frac{(1+k')\sin\varphi\cos\varphi}{\sqrt{1-k^2\sin^2\varphi}},$$

$$\cos(2\varphi-\varphi_1)=\frac{\cos^2\varphi+k'\sin^2\varphi}{\sqrt{1-k^2\sin^2\varphi}},\qquad \cos\varphi_1=\frac{\cos^2\varphi-k'\sin^2\varphi}{\sqrt{1-k^2\sin^2\varphi}},$$

が得られる。これらの間の関係を調べると,

$$\sin(2\varphi-\varphi_1)=k_1\sin\varphi_1, \tag{5}$$

$$\cos(2\varphi-\varphi_1)+k_1\cos\varphi_1=(1+k_1)\sqrt{1-k^2\sin^2\varphi}, \tag{6}$$

を得ることができる。

補足として, Landen 変換で得られた母数 k_1 が $k_1<k^2$ であることを述べておく。それを示すには, (3) を計算すればよい。実際に計算してみると,

$$k_1=\frac{1-k'}{1+k'}=\frac{1-\sqrt{1-k^2}}{1+\sqrt{1-k^2}}=\frac{k^2}{2-k^2+2\sqrt{1-k^2}},$$

が得られる。母数の範囲 $0\le k<1$ において, 分母が k について単調減少関数であり,

$$1<2-k^2+2\sqrt{1-k^2}\le 4,$$

を満たすので, $k_1<k^2$ が成立するのだ。しかも, $k\ll 1$ ならば $k_1\simeq k^2/4$ となる。したがって, Landen 変換 (3) によって楕円積分の母数は小さい値に変換される。

3.1 第一種楕円積分

Landen 変換には, 楕円関数の母数 k を減少させ, 楕円積分の被積分関数を一定の規則で変化させる性質がある。本節で説明するように, その性質は楕円積分を実行するにあたり, 大きな効果を発揮する。言い換えると, Landen 変換は第一種楕円積分を計算するための有効な手段となる。

第一種楕円積分の計算における Landen 変換の効果を示すには, 関係式 (5) の微分:

$$\cos(2\varphi-\varphi_1)(2\,\mathrm{d}\varphi-\mathrm{d}\varphi_1)=k_1\cos\varphi_1\,\mathrm{d}\varphi_1,$$

を整理し, $\mathrm{d}\varphi$ と $\mathrm{d}\varphi_1$ の関係:

$$2\cos(2\varphi-\varphi_1)\,\mathrm{d}\varphi=[\cos(2\varphi-\varphi_1)+k_1\cos\varphi_1]\,\mathrm{d}\varphi_1,$$

なる関係を導出しておく。左辺については, (5) より $\cos(2\varphi-\varphi_1)=\sqrt{1-k_1^2\sin^2\varphi_1}$ であり, 右辺については (6) が成り立つので,

$$\frac{\mathrm{d}\varphi}{\sqrt{1-k^2\sin^2\varphi}}=\frac{1+k_1}{2}\frac{\mathrm{d}\varphi_1}{\sqrt{1-k_1^2\sin^2\varphi_1}}, \tag{7}$$

なる関係が得られる。この式の両辺を積分すると,

$$F(k,\varphi)=\frac{1+k_1}{2}F(k_1,\varphi_1), \tag{8}$$

を得る。これが第一種楕円積分の Landen 変換である。Landen 変換は, 第一種楕円積分を, さらに母数が小さな第一種楕円積分に変換できる効果がある。これを繰り返していくと, 母数がゼロの第一種楕円積分に帰着できるというのが, 次の段落での考察である。

Landen 変換の反復　実際の数値計算では (8) を繰り返し適用すれば効率的に第一種楕円積分を計算できる。補母数について, $a\geq b$ なる正の実数を用いて $k'=b/a$ とおき, $a_1=(a+b)/2$, $b_1=\sqrt{ab}$ とすれば, $k_1=b_1/a_1$, $1+k_1=a/a_1$ が成立することは容易にわかる。

さらに Landen 変換を繰り返し, 母数 k_1 から $k_2,k_3,k_4,\ldots$ をつくっていくことを考えよう。すなわち,

$$k_{n+1}=\frac{1-k'_n}{1+k'_n},\qquad \tan(\varphi_{n+1}-\varphi_n)=k'_n\tan\varphi_n,$$

を適用した場合を想定する。すでに述べたように, この変換によって楕円関数の母数は $k_{n+1}<k_n^2$ のように急激にゼロに収束する。ここで, 先ほどの a_1,b_1 から $a_{n+1}=(a_n+b_n)/2$, $b_{n+1}=\sqrt{a_nb_n}$ を定義すると, $k'_n=b_n/a_n$, $1+k_n=a_{n-1}/a_n$ が成り立つことは帰納的に証明できる。

第一種楕円積分に関する Landen 変換 (8) を繰り返し適用してみると,

$$F(k,\varphi)=\frac{a}{a_1}\frac{F(k_1,\varphi_1)}{2}=\frac{a}{a_2}\frac{F(k_2,\varphi_2)}{2^2}=\cdots=\frac{a}{a_n}\frac{F(k_n,\varphi_n)}{2^n},$$

が得られる。Landen 変換を無限に繰り返し, $n\to\infty$ とすれば, $k_n\to 0$ となるので, $F(k_n,\varphi_n)\to\varphi_n$ となる。さらに, $b_n\leq b_{n+1}\leq a_{n+1}\leq a_n$ という事実から, 系列 a_n と b_n は共通の極限値 $M(a,b)$ に収束することも明らかである。この極限値 $M(a,b)$ は, a と b の算術幾何平均とよばれるが, これを用いると,

$$F(k,\varphi)=\lim_{n\to\infty}\frac{a\,\varphi_n}{2^nM(a,b)}, \tag{9}$$

と書くことができる。

特に $\varphi = \pi/2$ の場合, すなわち, 第一種完全楕円積分について考えてみる。積分変数 φ_n が $\pi/2$ の整数倍であれば, (6) より $\varphi_{n+1} = 2\varphi$ となるので, $\varphi_n = 2^{n-1}\pi$ と書くことができる。よって, 第一種完全楕円積分は

$$F(k) = \frac{a\pi}{2M(a,b)}, \tag{10}$$

で計算することができる。

3.2 収束速度

Landen 変換による第一種楕円積分の精度は, 算術幾何平均の収束速度に決定される。まず, a_{n+1} と b_{n+1} の差を評価してみると,

$$\begin{aligned} a_{n+1} - b_{n+1} &= \frac{a_n + b_n - 2\sqrt{a_n b_n}}{2} \\ &= \frac{(\sqrt{a_n} - \sqrt{b_n})^2}{2} = \frac{(a_n - b_n)^2}{2(\sqrt{a_n} + \sqrt{b_n})^2} \\ &= \frac{(a_n - b_n)^2}{4(a_{n+1} + b_{n+1})} \leq \frac{(a_n - b_n)^2}{8M(a,b)}, \end{aligned} \tag{11}$$

が得られる。 すなわち,

$$\left| \frac{a_{n+1} - b_{n+1}}{M(a,b)} \right| \leq \frac{1}{8} \left| \frac{a_n - b_n}{M(a,b)} \right|^2, \tag{12}$$

が成り立つ。 さらに, $b_{n+1} \leq M(a,b)$ より $a_{n+1} - b_{n+1} \geq a_{n+1} - M(a,b)$ が成り立ち, $a_n + b_n \geq 2M(a,b)$ より $a_n - b_n \leq 2(a_n - M(a,b))$ が導かれるので,

$$\left| \frac{a_{n+1} - M(a,b)}{M(a,b)} \right| \leq \frac{1}{2} \left| \frac{a_n - M(a,b)}{M(a,b)} \right|^2, \tag{13}$$

を得る。すなわち, a_n は $M(a,b)$ に向かって二次の収束をすることが示された。

4 第二種楕円積分の変換

第二種楕円積分についても, Landen 変換を適用して効率よく数値計算をすることができる。まず, (6) を自乗すると,

$$2(1 - k_1^2 \sin^2\varphi_1) - k_1'^2 + 2k_1 \cos\varphi_1 \sqrt{1 - k_1^2 \sin^2\varphi_1} = (1 + k_1)^2 (1 - k^2 \sin^2\varphi),$$

が得られる。この式に (7) を乗じて積分すれば, 第二種楕円積分の Landen 変換:

$$E(k,\varphi) = \frac{1}{1+k_1}\left[E(k_1,\varphi_1) - \frac{k_1'^2}{2}F(k_1,\varphi_1) + k_1\sin\varphi_1\right], \tag{14}$$

を得ることができる。

それでは, $\varphi = \pi/2$ の場合, すなわち, 完全楕円積分について Landen 変換を繰り返してみよう。しかしながら, $\varphi = \pi/2$ の条件で (14) を書き直すと,

$$E(k) = \frac{1}{1+k_1}\left[2E(k_1) - k_1'^2 F(k_1)\right],$$

となるので, これを繰り返すと, 右辺第一項が発散してしまうことが予想できる。発散項が発生することを考えると, この変換は数値計算には不向きである。そこで, (14) を (8) で除すると,

$$\frac{E(k)}{F(k)} = \frac{2}{(1+k_1)^2}\left[\frac{E(k)}{F(k)} - \frac{k_1'^2}{2}\right], \tag{15}$$

を得る。さらに, この式の両辺から $1-k^2/2$ を引くと,

$$\frac{E(k)}{F(k)} - \left(1-\frac{k^2}{2}\right) = \frac{2}{(1+k_1)^2}\left[\frac{E(k)}{F(k)} - \left(1-\frac{k_1^2}{2}\right) - \frac{k_1^2}{2}\right], \tag{16}$$

となる。この式を見やすくするため,

$$G(k) \equiv \frac{E(k)}{F(k)} - \left(1-\frac{k^2}{2}\right),$$

とおいてみると, (16) は

$$G(k) = \frac{2}{(1+k_1)^2}\left[G(k_1) - \frac{k_1^2}{2}\right], \tag{17}$$

と書くことができる。この変換式は発散項を含まないので, 前節の方法であらかじめ $F(k)$ を必要な精度で計算していれば, $E(k)$ を必要な精度で求めることができる。

Landen 変換の反復　それでは, (17) の変換を繰り返し, 数値計算に適した方法を見つけてみよう。ここでも, $a \geq b$ なる正の実数を用いて, $k' \equiv b/a$ とし, 算術幾何平均を用いて a_n と b_n を更新していく方法をとる。すると,

$$\begin{aligned}
G(k) &= 2\frac{a_1^2}{a^2}G(k_1) - \frac{a_1^2-b_1^2}{a^2} \\
&= 4\frac{a_2^2}{a^2}G(k_2) - 2\frac{a_2^2-b_2^2}{a^2} - \frac{a_1^2-b_1^2}{a^2} \\
&= 2^n\frac{a_n^2}{a^2}G(k_n) - \sum_{i=1}^{n} 2^{i-1}\frac{a_i^2-b_i^2}{a^2} \\
&= -\sum_{i=1}^{\infty} 2^{i-1}\frac{a_i^2-b_i^2}{a^2},
\end{aligned} \tag{18}$$

となることがわかる。この式の変形に関して, $n \to \infty$ のとき $k_n \to 0$ となることを利用している。さらに, この式の両辺に $1 - k^2/2$ を加えると,

$$\frac{E(k)}{F(k)} = 1 - \sum_{i=0}^{\infty} 2^{i-1} \frac{a_i^2 - b_i^2}{a^2}, \tag{19}$$

が得られる。ここで便宜上, $a_0 \equiv a$, $b_0 \equiv b$ とおいた。

収束速度　この節で公式化した $E(k)/F(k)$ の収束について調べるために, 数列 $a_n^2 - b_n^2$ の収束速度について考察する。

$$\begin{aligned} a_{n+1}^2 - b_{n+1}^2 &= \left(\frac{a_n - b_n}{2} \right)^2 \\ &= \frac{1}{4} \left(\frac{a_n^2 - b_n^2}{a_n + b_n} \right)^2 \leq \frac{1}{16} \left(\frac{a_n^2 - b_n^2}{M(a,b)} \right)^2. \end{aligned} \tag{20}$$

右辺の不等式は $a_{n+1} \geq M(a,b)$ であるので成立する。この関係式をもう少し整理すると、

$$\left| \frac{a_{n+1}^2 - b_{n+a}^2}{M(a,b)^2} \right| \leq \frac{1}{16} \left| \frac{a_n^2 - b_n^2}{M(a,b)^2} \right|^2, \tag{21}$$

が得られるので、$a_n^2 - b_n^2$ が 0 に向かって二次の収束をすることがわかる。

5　円周率の公式

楕円関数の分野において, 楕円積分 $F(k)$ と $E(k)$ の間には

$$F(k)\,E(k') + F(k')\,E(k) - F(k)\,F(k') = \frac{\pi}{2}, \tag{22}$$

という関係が成り立つ。これは Legendre の関係式とよばれている。本書ではこの関係式の証明は省略するが, 楕円関数の教科書には必ず載っているので, 興味があれば参照すればよい。

特殊な場合として, $k = \sqrt{1/2}$ であれば,

$$\left[2E(\sqrt{1/2}) - F(\sqrt{1/2}) \right] F(\sqrt{1/2}) = \frac{\pi}{2}, \tag{23}$$

が成り立つことがわかる。ここでこの関係式を

$$\left[\frac{2E(\sqrt{1/2})}{F(\sqrt{1/2})} - 1 \right] \left[F(\sqrt{1/2}) \right]^2 = \frac{\pi}{2},$$

と書き直し, $a = 1$, $b = 1/\sqrt{2}$ の条件下で, (10) と (19) を代入すると,

$$\pi = \frac{2[M(a,b)]^2}{1 - \sum_{i=0}^{\infty} 2^i (a_i^2 - b_i^2)}, \tag{24}$$

を得ることができる。さらに, $n \to \infty$ のとき $a_n \to M(a,b)$ であることを考えると,

$$\pi = \lim_{n\to\infty} \frac{2a_n^2}{1 - \sum_{i=0}^{n} 2^i (a_i^2 - b_i^2)}, \tag{25}$$

となる。これが, Salamin と Brent が発見した Gauss-Legendre 法である。既に見たように, この公式の分子と分母は 2 次収束をするので, この公式自体も π に向かって 2 次収束する。この収束の速さゆえに, 最近ではこのアルゴリズムが円周率計算の主流であるが, 手計算の時代, 多倍桁どうしの乗算, 除算, さらに, 平方根演算を含むこのアルゴリズムの検証はほぼ不可能であった。この公式自体は 200 年前から存在していたかもしれないと言われるが, 1980 年代になるまで使われることはなかった。

5.1 単純な実装

C 言語の簡単なプログラムで Gauss-Legendre 法を検証すると下のようになる。ただし, このプログラムはアルゴリズムを解説するための仮想的なプログラムである。下の検証結果のように, 小数点以下 100 桁の演算を検証するには, 計算機の倍精度 (double 型) 演算では不足である。その精度で計算するには多倍長演算用のプログラムが必要である。

```
main()
{
    int    i;
    double dfA = 1.0;
    double dfB = sqrt(0.5);
    double dfX = 0.5;
    double dfC = 0.5;

    for (i=0; i< 7; i++) {
        double dfY;
        double dfP;

        dfC -= dfX * (dfA + dfB) * (dfA - dfB);
        dfP  = dfA * dfA / dfC;

        dfY  = (dfA + dfB) * 0.5;
        dfB  = sqrt(dfA * dfB);
        dfA  = dfY;
        dfX *= 2.0;
```

```
        printf("Loop %d: pi=%2.100f\n", i+1, dfP);
    }
}
```

Salamin と Brent による基本的なアルゴリズムでは, 第 1 回目のループでは整数部さえも誤っているが, その以降, 1 桁, 3 桁, 9 桁,. . . のように急激に有効桁数が増大している。既に説明したように, このアルゴリズムが 2 次収束をするため, ループのたびに有効桁が 2 倍以上増大し, 第 8 回目のループで小数点以下 100 桁以上の精度を達成している。

```
Loop 1:  pi = 4.00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
                00000 00000 00000 00000 00000 00000 00000 00000 00000 00000
Loop 2:  pi = 3.18767 26427 12108 62720 19299 70525 36923 26510 53571 85936
                92264 87633 98627 51228 32528 12233 01147 28610 66016 17974
Loop 3:  pi = 3.14168 02932 97653 29391 80704 24560 00938 27957 19438 81540
                28326 44189 46319 56630 01010 25531 93888 89427 51526 46103
Loop 4:  pi = 3.14159 26538 95446 49600 29147 58818 04348 61088 79237 26131
                15896 51101 35768 46530 79503 08650 17740 97586 28986 31570
Loop 5:  pi = 3.14159 26535 89793 23846 63606 02706 63132 17577 02411 34242
                93564 86846 01523 84109 48606 92775 82680 62200 73327 62131
Loop 6:  pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69949 16472
                66058 34696 12594 87480 06095 32900 58518 51575 93171 01939
Loop 7:  pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
                58209 74944 59230 78164 06286 20899 86280 46852 22865 41150
Loop 8:  pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
                58209 74944 59230 78164 06286 20899 86280 34825 34211 70679
```

5.2 公式の改良 1

Gauss-Legendre 法による収束の速さをさらに改善することはできないにしても, 公式を変形することによって, 第 1 回目のループでの精度を向上するなどの工夫は可能である。ここでは, 頻繁に用いられる公式を導出する。まず, 公式 (25) の分母を X_n とおくと,

$$
\begin{aligned}
X_n &= 1 - \sum_{i=0}^{n} 2^i (a_i^2 - b_1^2) = \frac{1}{2} - \sum_{i=1}^{n} 2^i (a_i^2 - b_1^2) \\
&= \frac{1}{2} - \sum_{i=1}^{n} 2^i \left[\left(\frac{a_{i-1}^2 + b_{i-1}^2}{2} \right)^2 - a_{i-1} b_{i-1} \right] = \frac{1}{2} - \sum_{i=1}^{n} 2^i \left(\frac{a_{n-1} - b_{n-1}}{2} \right)^2 \\
&= \frac{1}{2} - \sum_{i=1}^{n} 2^i (a_{i-1} - a_i)^2 = 2 \left[\frac{1}{4} - \sum_{i=1}^{n} 2^{i-1} (a_{i-1} - a_i)^2 \right],
\end{aligned}
$$

のように計算できる。分子については, a_n のかわりに, もう 1 段精度の良い a_{n+1} を使うことにし,

$$\pi_n = \frac{(a_n + b_n)^2}{4\left[\dfrac{1}{4} - \displaystyle\sum_{i=1}^{n} \quad 2^{i-1}(a_{i-1} - a_i)^2\right]}, \tag{26}$$

を定義する。この数列も π に向かって 2 次収束する。前節の基本アルゴリズムから直接変形した式であるので, 収束の速さは変わらないが, π_0 の精度が改善されているため, 同じ精度を得る場合, ループの回数を減らすことができる。もう一つの改善点は, 総和記号の中の $a_i^2 - b_i^2$ が $(a_{i-1} - a_i)^2$ に変形されていることである。この改善は数値計算の上で重要である。それを示すため, $a_i^2 - b_i^2$ と $a_{i-1} - a_i$ を個別に数式変形すると,

$$a_i^2 - b_i^2 = (a_i + b_i)(a_i - b_i) \simeq 2M(a, b)\,(a_1 - b_i),$$

$$a_{i-1} - a_i = a_{i-1} - \frac{a_{i-1} + b_{i-1}}{2} = -\frac{a_{i-1} - b_{i-1}}{2},$$

のように計算できる。この数式で注目するのは, 第 1 式では $a_i - b_i$, 第 2 式では $a_{i-1} - b_{i-1}$ である。既に示したように, $a_i - b_i$ はゼロに向かって 2 次収束するので, 十分に大きな i に対して $|a_{i-1} - a_i| \gg |a_i - b_i|$ が成立する。数値計算において, 同程度の数値を減算すると, 桁落ちと呼ばれる好ましくない状態に陥る。桁落ちを考えると, 改善したアルゴリズムでは, 減算による桁落ちの危険性が緩和されているのだ。

```
main()
{
    int    i;
    double dfA = 1.0;
    double dfB = sqrt(0.5);
    double dfX = 4.0;
    double dfC = 1.0;

    for (i=0; i< 5; i++) {
        double dfY;
        double dfP;

        dfY  = (dfA + dfB) * 0.5;
        dfC -= dfX * (dfA - dfY) * (dfA - dfY);
        dfB  = sqrt(dfA * dfB);
        dfA  = dfY;
        dfX *= 2.0;
        dfP  = (dfA + dfB) * (dfA + dfB) / dfC;

        printf("Loop %d: pi=%2.100f\n", i+1, dfP);
    }
}
```

このプログラムを実行すると, 次のような演算結果が得られる。第 1 回目のループで小数点以下 2 桁目まで正確に計算されている。このアルゴリズムでは 6 回の繰り返しで小数点以下 100 桁以上の精度が達成できる。

Loop 1: pi = 3.14057 92505 22168 24831 13312 68975 82331 17734 40237 51294
83356 43486 69334 55827 58034 90290 78272 87621 55276 69005

Loop 2: pi = 3.14159 26462 13542 28214 93444 31982 69577 43144 37223 34560
27945 59539 48482 14347 67220 79526 46946 43448 91799 13058

Loop 3: pi = 3.14159 26535 89793 23827 95127 74801 86397 43812 25504 83544
69357 87330 70202 63821 37838 92739 90314 16942 04346 90584

Loop 4: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 14678 28364
89215 56617 10697 60267 64500 64306 17110 06577 72659 80684

Loop 5: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86256 28703 21167 20359

Loop 6: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679

5.3 公式の改良 2

さらなる改良として, (26) において, $a_0 \leftarrow a_1$, $b_0 \leftarrow b_1$ と置き換える方法がある。そうすれば, ループの回数を 1 回だけ減らすことができる。これを具体的に書くと

$$
\begin{aligned}
a_1 &= \frac{2+\sqrt{2}}{4}, \qquad b_1 = \frac{1}{\sqrt[4]{2}}, \\
\pi_n &= \frac{(a_n+b_n)^2}{4\left[\dfrac{2\sqrt{2}-1}{8} - \displaystyle\sum_{i=2}^{n} 2^{i-1}(a_{i-1}-a_i)^2\right]},
\end{aligned}
\tag{27}
$$

のように書くことができる。プログラムは, 初期値が変わっただけである。

```
main()
{
    int    i;
    double dfA = (2.0 + sqrt(2.0)) * 0.25;
    double dfB = sqrt(sqrt(0.5));
    double dfX = (sqrt(8.0) - 1.0) * 0.125;
    double dfC = 2.0;

    for (i=0; i < 5; i++) {
        double dfY;
        double dfP;

        dfA  = (dfA + dfB) * 0.5;
        dfY  = (dfA - dfB) * (dfA - dfB);
        dfX -= (dfC * dfY);
        dfB  = sqrt(dfA * dfA - dfY);
        dfC *= 2.0;

        dfP  = (dfA + dfB) * (dfA + dfB) * 0.25 / dfX;
        printf("Loop %d: pi=%17.17f\n", i+1, dfP);
    }
}
```

このアルゴリズムの効果は, 上で説明したように, ループ回数が 1 回減っただけである。前回の結果と比較すると, 単純にループ番号が 1 だけずれていることが確認できるはずだ。

Loop 1: pi = 3.14159 26462 13542 28214 93444 31982 69577 43144 37223 34560
27945 59539 48482 14347 67220 79526 46946 43448 91799 13058

Loop 2: pi = 3.14159 26535 89793 23827 95127 74801 86397 43812 25504 83544
69357 87330 70202 63821 37838 92739 90314 16942 04346 90584

Loop 3: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 14678 28364
89215 56617 10697 60267 64500 64306 17110 06577 72659 80684

Loop 4: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86256 28703 21167 20359

Loop 5: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679

5.4 大浦のアルゴリズム

大浦先生 (京都大学) は, 公式 (26) において, $A_n \equiv a_n^2$, $B_n \equiv b_n^2$ のように置き換えた。その置き換えによって, これらの数列は,

$$A_{n+1} = \frac{1}{2}\left(\frac{A_n + B_n}{2} + \sqrt{A_n B_n}\right), \qquad B_{n+1} = \sqrt{A_n B_n}, \tag{28}$$

なる漸化式を満足する。これらの数列は言うまでもなく $M^2(a,b)$ に収束する。しかも,

$$\begin{aligned} A_{n+1} - B_{n+1} &= \frac{(\sqrt{A_n} - \sqrt{B_n})^2}{2} = \frac{(A_n - B_n)^2}{2(A_n + B_n + 2\sqrt{A_n B_n})} \\ &= \frac{(A_n - B_n)^2}{8A_{n+1}} \leq \frac{1}{8}\frac{(A_n - B_n)^2}{M^2(a,b)}, \end{aligned}$$

であることから, $A_n - B_n$ が 2 次収束することがわかる。これらの数列を用いて (26) を書き換えると,

$$\pi_n = \frac{2A_n}{1 - \sum_{i=0}^{n} 2^i (A_i - B_i)}, \tag{29}$$

を得る。 この公式による計算結果を下に示す。 このアルゴリズムは、第 5.1 節のアルゴリズムに対応する基本的な実装例であるが、第 5.1 節よりも 1 回少ないループで同一の結果が得られている。

```
main()
{
    int i;
    double dfA = 1.0;
    double dfB = 0.5;
```

```
    double dfC = dfA - dfB;
    double dfX = 1.0 - dfC;
    double dfP = 1.0;

    for (i=0; i < 5; i++) {
        double dfY;

        dfP *= 2.0;
        dfY = 0.5 * (dfA + dfB);
        dfB = sqrt(dfA * dfB);
        dfA = 0.5 * (dfY + dfB);
        dfC = dfA - dfB;
        dfX = dfX - dfP * dfC;

        dfY = 2 * dfA / dfX;
        printf("Loop %d: pi=%17.17f\n", i+1, dfY);
    } /* <----- end for */
}
```

Loop 1: pi = 3.18767 26427 12108 62720 19299 70525 36923 26510 53571 85936
92264 87633 98627 51228 32528 12233 01147 28610 66016 17974

Loop 2: pi = 3.14168 02932 97653 29391 80704 24560 00938 27957 19438 81540
28326 44189 46319 56630 01010 25531 93888 89427 51526 46103

Loop 3: pi = 3.14159 26538 95446 49600 29147 58818 04348 61088 79237 26131
15896 51101 35768 46530 79503 08650 17740 97586 28986 31570

Loop 4: pi = 3.14159 26535 89793 23846 63606 02706 63132 17577 02411 34242
93564 86846 01523 84109 48606 92775 82680 62200 73327 62130

Loop 5: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69949 16472
66058 34696 12594 87480 06095 32900 58518 51575 93171 01938

Loop 6: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 46852 22865 411502

Loop 7: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679

5.5 大浦の方法 2

分子 A_n の代わりにもう一段精度のよい A_{n+1} を用いたほうが少ない繰り返しで良い精度を得ることができる。 その場合,

$$\pi_n = \frac{A_n + B_n - 2(A_{n+1} - B_{n+1})}{1 - \sum_{i=0}^{n} 2^i (A_i - B_i)},$$

となるが, 上で示したように, $A_n - B_n$ は 0 に向かって 2 次収束するため, 分母を $A_n + B_n$ で代用してもよい。そうすると,

$$\pi_n = \frac{A_n + B_n}{1 - \sum_{i=0}^{n} 2^i (A_i - B_i)}, \tag{30}$$

なる公式を得る。この公式によって、ループの回数が 1 回減らせるわけではないが, 精度がよい計算ができていることがわかる。

```
main()
{
    int i;
    double dfA = 1.0;
    double dfB = 0.5;
    double dfC = dfA - dfB;
    double dfX = 1.0 - dfC;
    double dfP = 1.0;

    for (i=0; i < 5; i++) {
        double dfY;

        dfP *= 2.0;
        dfY = 0.5 * (dfA + dfB);
        dfB = sqrt(dfA * dfB);
        dfA = 0.5 * (dfY + dfB);
        dfC = dfA - dfB;
        dfX = dfX - dfP * dfC;

        dfY = (dfA + dfB) / dfX;
        printf("Loop %d: pi=%17.17f\n", i+1, dfY);
    } /* <----- end for */
}
```

Loop 1: pi = 3.14075 44820 34081 47040 14474 77894 02692 44882 90178 89452
69198 65725 48970 63421 24396 09174 75860 46457 99512 13480

] Loop 2: pi = 3.14159 26468 24848 79608 64969 02723 48368 66275 24652 98936
31526 96067 50754 12915 81953 34818 95298 00314 53410 93865

Loop 3: pi = 3.14159 26535 89793 23828 69472 13656 12084 95019 56604 74590
31919 84176 53031 67449 22672 15611 99188 64229 42342 96303

Loop 4: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 15777 86289
04912 76120 17425 78899 64118 88307 61440 59023 48466 95632

Loop 5: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86256 52756 98474 61300

Loop 6: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679

5.6 大浦の方法 3

公式 (29) において、第 $n+1$ 項の式:

$$\pi_{n+1} = \frac{2A_{n+1}}{1 - \sum_{i=0}^{n+1} 2^i (A_i - B_i)},$$

について考えてみる。 まず、分子を次のように変形してみる。

$$
\begin{aligned}
2A_{n+1} &= A_n + B_n - \left(\frac{A_n + B_n}{2} - \sqrt{A_n B_n}\right) \\
&= A_n + B_n - \frac{(A_n - B_n)^2}{8A_{n+1}} = \frac{1}{A_{n+1}}\left[(A_n + B_n)A_{n+1} - \frac{(A_n - B_n)^2}{8}\right] \\
&= \frac{1}{A_{n+1}}\left[(A_n + B_n)^2 \frac{A_{n+1}}{A_n + B_n} - \frac{(A_n - B_n)^2}{8}\right] \\
&\approx \frac{1}{2A_{n+1}}\left[(A_n + B_n)^2 - \frac{3}{8}(A_n - B_n)^2\right],
\end{aligned}
$$

ここで,

$$
\frac{A_{n+1}}{A_n + B_n} \approx \frac{1}{2}\left[1 - \frac{(A_n - B_n)^2}{8(A_n + B_n)^2}\right],
$$

なる近似が成り立つことを利用した。

円周率の第 n 近似式の分母を X_n をおき, 同様にして X_{n+1} を評価する。

$$
\begin{aligned}
X_{n+1} &= X_n - 2^{n+1}(A_{n+1} - B_{n+1}) \\
&= X_n - \frac{2^{n+1}}{16A_{n+1}}(A_n - B_n)^2 \\
&= \frac{1}{A_{n+1}}\left[X_n A_{n+1} - \frac{2^n}{8}(A_n - B_n)^2\right] \\
&\approx \frac{1}{2A_{n+1}}\left[X_n(A_n + B_n) - \frac{2^n}{4}(A_n - B_n)^2\right].
\end{aligned}
$$

これらの結果より, 次の近似式を得ることができる。

$$
\pi_n = \frac{(A_n + B_n)^2 - \dfrac{3}{2}\dfrac{(A_n - B_n)^2}{4}}{(A_n + B_n)\left[1 - \displaystyle\sum_{k=0}^{n} 2^k (A_k - B_k)\right] - 2^n \dfrac{(A_n - B_n)^2}{4}}. \tag{31}
$$

この近似式は複雑そうに見えるが, これまでの実装と比べてもそれほど大きな変化はない。ループが終了したときの後処理として, 多少の乗算が必要となるが, ループ (平方根計算が含まれる) をもう 1 回実行するのに比べると計算量はかなり小さくなる。 実装例を以下に示すが、1 回目の精度が先ほどよりも改善されている。

```
main()
{
    int i;
    double dfA = 1.0;
    double dfB = 0.5;
```

```
    double dfC = dfA - dfB;
    double dfX = 1.0 - dfC;
    double dfP = 1.0;

    for (i=0; i < 5; i++) {
        double dfY;

        dfP *= 2.0;
        dfY = 0.5 * (dfA + dfB);
        dfB = sqrt(dfA * dfB);
        dfA = 0.5 * (dfY + dfB);
        dfC = dfA - dfB;
        dfX = dfX - dfP * dfC;

        {   double dfAB = dfA + dfB;
            double dfDD = dfC * dfC * 0.25;

            dfY = (dfAB * dfAB - 1.5 * dfDD) / (dfAB * dfX - dfP * dfDD);
            printf("Loop %d: pi=%17.17f\n", i+1, dfY);
        }
        printf("Loop %d: pi=%17.17f\n", i+1, dfY);
    } /* <----- end for */
}
```

Loop 1: pi = 3.14159 26000 44887 78801 30778 86509 51254 78884 37738 36481
39000 40563 56280 27382 27014 57300 48226 63431 28191 02019

Loop 2: pi = 3.14159 26535 89793 23697 05911 23304 36161 59246 37918 21129
89326 44688 98958 64370 34698 75824 83390 50022 27013 88487

Loop 3: pi = 3.14159 26535 89793 23846 26433 83279 50288 41967 00000 71606
66903 19563 11311 79392 49944 95214 68889 91233 91087 81963

Loop 4: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86068 84855 40266 77552

Loop 5: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679

5.7 大浦の方法 4

大浦の方法では初項が $A_0 = 1,\ B_0 = 1/2$ というように簡単すぎる形をしているので、最初の計算を省略することもできそうである。 第 2 項を計算すると、

$$A_2 = \frac{1}{4}\left[\sqrt{1+3\sqrt{1/8}} + \left(1+3\sqrt{1/8}\right) - \frac{5}{8}\right],$$

$$B_2 = \frac{1}{2}\sqrt{1+3\sqrt{1/8}},$$

$$X_2 = \sqrt{1+3\sqrt{1/8}} - \sqrt{1/8} - \frac{5}{8},$$

が得られる。 これを初項とすることでループの回数を減らすことができる。形式的にアルゴリズムを書くと次のようになる。

$$\pi_n = \frac{(A_n + B_n)^2 - \dfrac{3}{2}\dfrac{(A_n - B_n)^2}{4}}{(A_n + B_n)\left[X_2 - \displaystyle\sum_{k=3}^{n} 2^k (A_k - B_k)\right] - 2^n \dfrac{(A_n - B_n)^2}{4}}, \tag{32}$$

数式を見ると, 多少複雑に見えるかもしれないが, 共通部分が多いので, プログラムは簡単である。初項には, $\sqrt{1/8}$ や $1 + 3\sqrt{1/8}$ が繰り返し現れている。最終的に円周率を決定する (32) にも, $A_n + B_n$ と $(A_n - B_n)^2$ が繰り返し現れているので, プログラムの簡略化ができる。プログラム例を示すと次のようになる。

```
main()
{
    int i;
    double dfC = sqrt(0.125);
    double dfA = 1.0 + 3.0 * dfC;
    double dfB = sqrt(dfA);
    double dfX = 0.625;
    double dfP = 4.0;

    dfA = dfA - dfX + dfB;
    dfX = dfB - dfX - dfC;
    dfB *= 0.5;
    dfA *= 0.25;

    for (i=0; i < 5; i++) {
        double dfY;

        dfP *= 2.0;
        dfY = 0.5 * (dfA + dfB);
        dfB = sqrt(dfA * dfB);
        dfA = 0.5 * (dfY + dfB);
        dfC = dfA - dfB;
        dfX = dfX - dfP * dfC;

        {   double dfAB = dfA + dfB;
            double dfDD = dfC * dfC * 0.25;

            dfY = (dfAB * dfAB - 1.5 * dfDD) / (dfAB * dfX - dfP * dfDD);
            printf("Loop %d: pi=%17.17f\n", i+1, dfY);
        }
    } /* <----- end for */
}
```

このアルゴリズムを実行した結果は下のようになる。驚くことに, 第 1 回目のループですでに小数点以下 38 桁まで正確に計算できているのだ。第 2 回目のループで精度は小数点以下 82 桁まで, 第 3 回目には 100 桁を超えている。

Loop 1: pi = 3.14159 26535 89793 23846 26433 83279 50288 41967 00000 71606
66903 19563 11311 79392 49944 95214 68889 91233 91087 81963

Loop 2: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86068 84855 40266 77552

Loop 3: pi = 3.14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679