多倍長計算手法

平成24年度第4四半期

目次

１．はじめに

数学と計算機で実行した結果は必ずしも一致しません。これは，

１．計算機で使用される変数は連続でなく離散的であること。
２．浮動小数点演算では有限のビット数しかもっていないので演算毎に誤差が発生。
３．また表現可能な数値範囲が有限であること。
４．極限操作が有限回の操作で行うこと。

などが挙げられます。

このため、必要とされる精度を持った結果を得るには、それに必要なビット数が問題毎にあります。一般に使用されている演算精度は倍精度ですが、これではビット数が不足すると多倍長計算が必要となります。

もう一つの問題に極限操作があり,加速法の使用により,収束範囲が数学より広くなる場合があり,結果の妥当性のチェックが要求される場合が発生します。これは必要なビット数との相互作用がある場合があるため、ここでも多倍長計算が必要になってきます。

2.反復解法
2.1　対称行列

3次元ポアソン方程式を差分法で解く際に現れる係数行列をA,
$Ax = b$で$x^T = (1,1,....,1,1)$になる様にbを設定.x^Tの初期値を
$x_0{}^T = (0,0,...,0,0)$としてサイズ$129 \times 129 \times 129$
収束判定値は残差10^{-12}とする。

3次元反復法		SR16000/M1			
	129*129*129 収束判定値　1.0e-12				倍精度
			実行時間(秒)		
解法	反復回数	single	32core	64smp	
bcg	618	24.274	3.571	2.86	
cgs	414	16.649	2.721	2.232	
scg	618	16.255	1.975	1.799	
bicgs		16.339	2.393	2.382	
	反復回数	402	395	420	
cgsmil		26.125	18.985	19.739	
	反復回数	277	292	308	
gpbicg		24.897	3.094	2.707	
	反復回数	434	381	414	
cgsilu	精度	single	32core	64smp	
	倍精度	944.418	644.799	635.057	
	反復回数	10000	10000	10000	
cgsiluq	4倍精度	307.614	159.012	158.055	
	反復回数	275	275	275	
(注) 10000 は収束しないのでこの回数で打ち切ったもの					

この様に比較的収束し易い問題でも4倍精度が必要になる事があります。

2.2 非対称行列

3 次元ポアソン方程式

$$\Delta u + R \times \frac{\partial u}{\partial x} = -f$$

領域 $[0,1] \times [0,1] \times [0,1]$

解析解 $u(x,y,z) = e^{xyz} \times \sin(\pi x) \times \sin(\pi y) \times \sin(\pi z)$

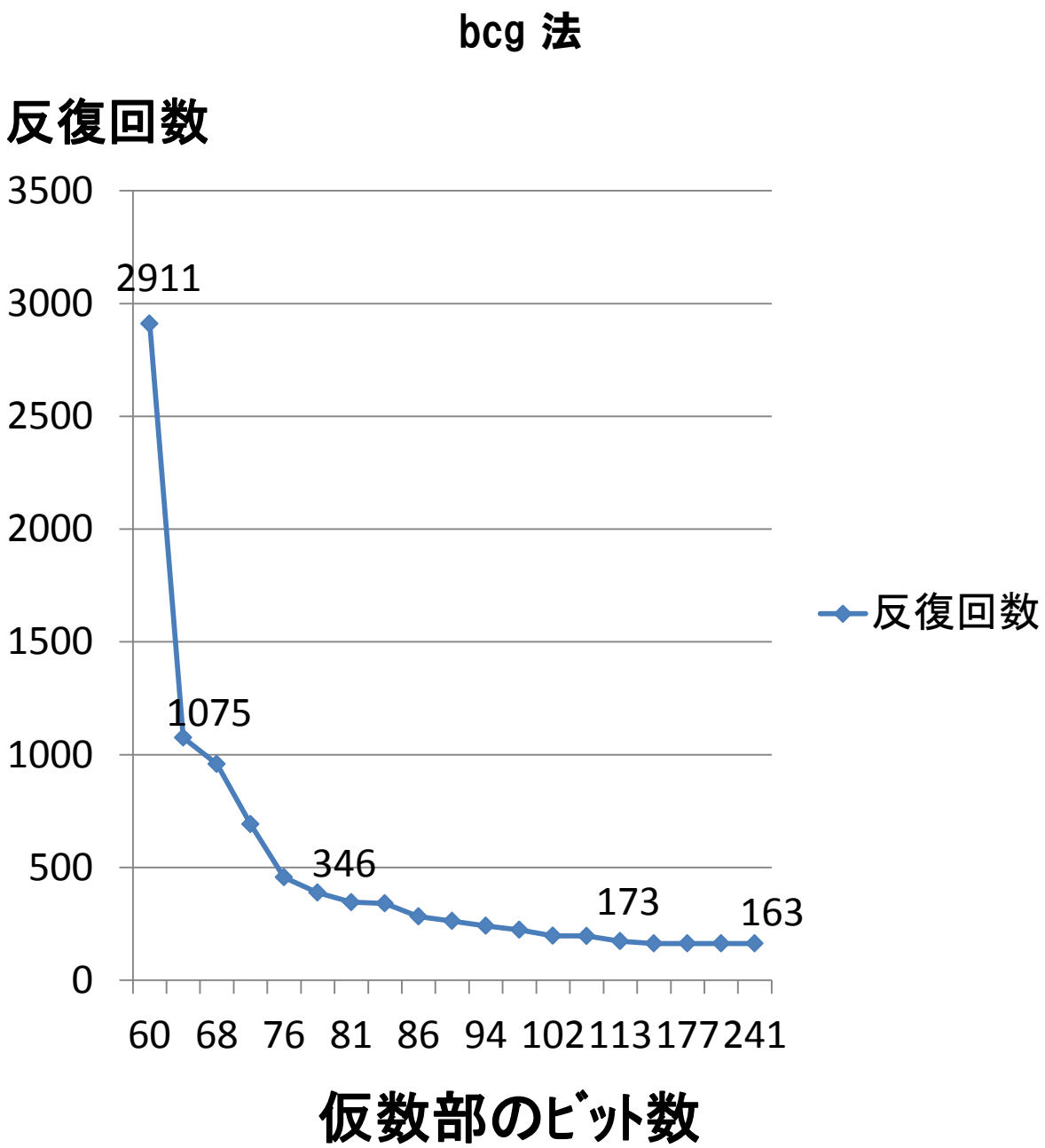
$$R = 100, nx = ny = nz = 65$$

収束判定値は共役残差 10^{-12}

反復回数の調査は,SR16000/M1とx5570を使用して行っています。

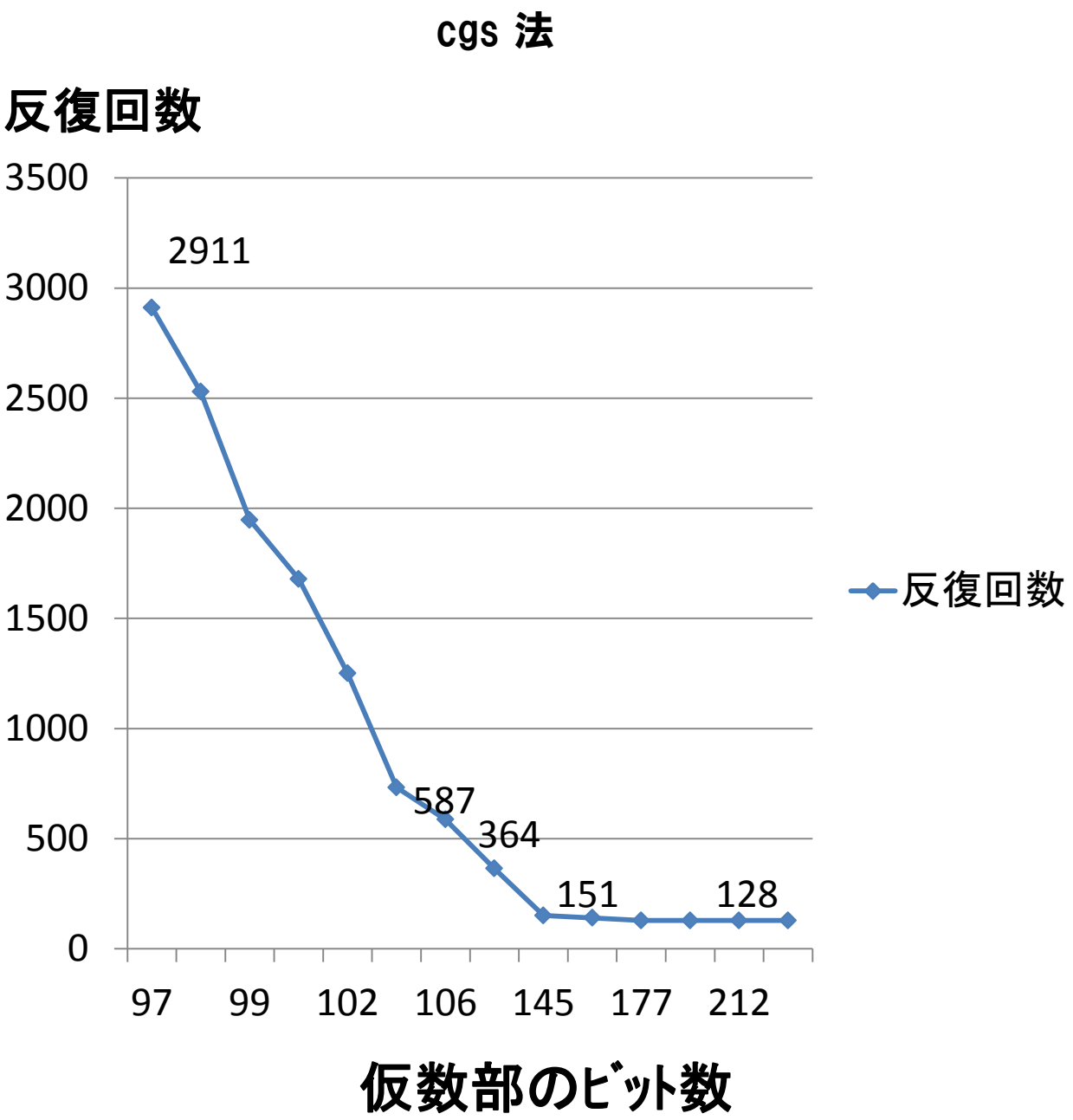
2.2.1 bcg 法

演算精度と反復回数は以下の様になっています。これより,4倍精度演算が良い事がわかります。



2.2.2　cgs　法

演算精度と反復回数は以下の様に
なっています。
これより,6倍精度演算が良い事がわかります。



3.乱数ルーチン

3.1 手法概略

疑似乱数に関する概略説明

(1)混合型合同式法(*mixed congruence method*)

$$x_{n+1} = \lambda x_n + \mu \pmod{M}$$

（ア）μとMは互いに素

（イ）PがMの素因数ならば$\lambda \equiv 1 \pmod{M}$

（ウ）4がMの素因数ならば$\lambda \equiv 1 \pmod{4}$

ならば周期M.

これは,$M = 2^s$ならば,μが奇数で$\lambda \equiv 1 \pmod{4}$

すなわち,$\lambda = 2^a + 1, a \geq 2$はすべて周期$2^s$.

系列相関係数x_n, x_{n+1}の間の相関係数$\rho \fallingdotseq \frac{1}{\lambda} - \frac{6\mu}{\lambda M}(1 - \frac{\mu}{M}) + K$

$$(-\frac{\lambda}{M} \leq K \leq \frac{\lambda}{M})$$

(2)乗算型合同式法

$$x_{n+1} = \lambda x_n \pmod{M}$$

$M = 2^{35}, \lambda = 2^7 + 1$

$M = 2^{47}, \lambda = 2^9 + 1$

$M = 2^s$; x_0が奇数,$\lambda \equiv 5 \pmod 8$,または$\lambda \equiv 3 \pmod 8$ならば最大周期2^{s-2}.

$\lambda = 5^{2k+1}, \lambda = 7^{4k+1}, \lambda = 13^{13}$

3．2　並列化コーディング例

3.2.1　乗算型合同式法

$x_{n+1} = \lambda x_n \pmod{M}$

$x_0 = iseed$(乱数ルーチンの種)

一般に$M = 2^s$; x_0が奇数, $\lambda \equiv 5 \pmod 8$, または$\lambda \equiv 3 \pmod 8$

ならば最大周期2^{s-2}となる.事と混合型（線形)合同式法

にみられる系列相関係数x_n, x_{n+1}が大きくなる事のない

$\lambda = 13^{13}, M = 2^{53}$($2^{59}$の方が良いと考えられますが,拡張倍精度

が必要となるために$M = 2^{53}$にしています.)

オリジナルソースをinline 展開したソース

```
fave=(double) (ib)*r ;
for (i=1;i<68719476736;i++) {
ic=ia*ib ;
ib=ic&mask ;
g05faf=(double) ( ib) *r ;
fave=fave+g05faf ;
}
    fave=fave*t ;
```

での結果 fave=　0.50000054390657433512

に対し　修正した結果

1cpu　fave=　0.50000054390657433512

8smp　fave=　0.50000054390660353398

となり,1cpuでは結果は最後まで一致しています.

プログラムでは, $x_{n+1} \equiv \lambda x_n \pmod{M}$を変形し
$x_n \equiv \lambda^n x_0 \pmod{M}$として先に$\lambda^n \pmod{M}$を計算してテーブルにします.

```
long int i, j, k ;
    ilist[0] = 1;
    ib = 1;
    for(i = 1; i < n; i++) {
    ic = ia * ib ;
    ib = ic & mask ;
    ilist[i] = ib ;
    }
    ic = ia * ib ;
    ia = ic & mask ;
    jlist[0] = 1;
    ib = 1;
    for(i = 1; i < n; i++) {
    ic = ia * ib ;
    ib = ic & mask ;
    jlist[i] = ib ;
    }
    ic = ia * ib ;
    ia = ic & mask ;
    klist[0] = 1;
    ib = 1;
    for(i = 1; i < n; i++) {
    ic = ia * ib ;
    ib = ic & mask ;
    klist[i] = ib ;
    }
```

実行部分のソースは以下のようになります。

```
#pragma omp parallel private(i,j,k,ia,ib,ic,g05faf)
    reduction(+:fave) ¥     num_threads(8)
#pragma omp for
    for (l=0;l<68719476736;l++) {
    k=l>>24 ;
    j=(l-(long int)(k<<24))>>12 ;
    i=l-(long int)(k<<24)-(long int)(j<<12) ;
    ib=klist[k] ;
    ia=jlist[j] ;
    ic=(ia*ib)&mask ;
    ia=ilist[i];
    ib=(ia*ic)&mask ;
    ia=(ib*iseed)&mask ;
    g05faf=(double) ( ia) *r ;
    fave=fave+g05faf ;
    }
       fave=fave*t ;
```

テーブル作成には、iseed(種)は全く影響しません。並列実行時は参照のみとなります。

N=2^36での実行時間(秒)は,

	e5430	t2k
オリジナル(inline化ソース)	235	248
修正ソース	415	589
修正ソース(smp)	72 (8smp)	44 (16smp)

コンパイルオプション icc -Os -lm

同じ問題をMPI（フラットMPI）で実行した結果は以下の様になっています.

乗算合同法			
	T2K		BG/Q 32ノード
mpi数	実行時間(秒)	mpi数	実行時間(秒)
16	40.302161	1	2576.929578
32	20.273622	512	5.090237
64	10.1492	1024	3.278346
128	5.073721	2048	2.882218
256	2.550183	7.297119	

並列化効果がはっきりでています。

拡張倍精度

乱数列 $r_n \equiv a r_{n-1} \pmod{2^{59}}$　　$a = 13^{13}, iseed(種) = 5^{11}$

$N = 2^{36}$ での平均値

倍精度	0.50000207337461433088
拡張倍精度	0.50000207337165584846327970325319 3
4 倍精度	0.50000207337165584846327970325319 3

実行時間(秒)　($e5430$　$1cpu$)

倍精度	234.624821
拡張倍精度	217.760483
4 倍精度	3240.429380

条件

倍精度　icc -Os -lm
拡張倍精度　icc -Os –fp-model extended -lm
4倍精度　ifort -Os

精度、性能面では拡張倍精度演算が倍精度演算より良い事を示している。

3.2.2　混合(線形)型合同式法

$$r_n \equiv a r_{n-1} + c \pmod{M}$$

$$r_n \equiv a^n r_0 + (\sum_{m=0}^{n} a^{m-1})c \pmod{M}$$

乗算合同法のテーブルに加えて,$\sum_{m=0}^{n} a^{m-1}$のテーブルを作成する.

$$l = i + j \times 4096 + k \times 4096^2$$

$$(0 \le i \le 4095, 0 \le j \le 4095, 0 \le k \le 4095, l \ne 0)$$

乗算合同法のテーブルを$ilist$, $jlist$, $klist$として,追加するテーブルを,$ilist1$, $jlist1$, $klist1$とする.

$$ilist1(i) = \sum_{m=0}^{i} a^m \quad (0 \le i \le 4095)$$

$$jlist1(j) = \sum_{m=0}^{j} (a^{4096})^m \quad (0 \le j \le 4095)$$

$$klist1(k) = \sum_{m=0}^{k} (a^{4096\times4096})^m \quad (0 \le k \le 4095)$$

$$A = \sum_{m=0}^{4095} a^m = ilist1(4095), B = \sum_{m=0}^{4096\times4096-1} a^m = A \times jlist1(4096)$$

$$\sum_{m=1}^{l} a^{m-1} = B \times klist1(k-1) + klist(k) \times [A \times jlist1(j-1) + jlist(j) \times ilist1(i-1)] \pmod{M}$$

$k = 0$のときは,$klist1(k-1) = 0$, $j = 0$のときは$jlist1(j-1) = 0$,
$i = 0$のときは,$ilist1(i-1) = 0$とする.

オリジナルソースをinline化したものは以下の様になります。

```
for (i=1;i<68719476736;i++) {
    ie=ia*ib+ic ;
    ib=ie&mask ;
    g05faf=(double) ( ib) *r ;
    fave=fave+g05faf ;
    }
        fave=fave*t ;
```

N=2^36での実行時間(秒)は,

	e5430	t2k
オリジナル(inline化ソース)	262	256
修正ソース	708	1145
修正ソース(smp)	120 (8smp)	82 (16smp)

となっています.

コンパイルオプションはicc -Os –lm です.
具体的なテーブル作成部分と実行部分のソースを
以降に記述しました。

追加するテーブル部分のコーディング

```
 ilist1[0]=1 ;
for(i=1;i<n;i++) {
ie=ia*ilist1[i-1]+1 ;
ib=ie&mask ;
ilist1[i]=ib ;
}
ih=jlist[1] ;
jlist1[0]=1 ;
for(i=1;i<n;i++) {
ie=ih*jlist1[i-1]+1 ;
ib=ie&mask ;
jlist1[i]=ib ;
}
ih=klist[1] ;
klist1[0]=1 ;
for(i=1;i<n;i++) {
ie=ih*klist1[i-1]+1 ;
ib=ie&mask ;
klist1[i]=ib ;
}
```

結果にかんしては、オリジナルの
fave=　0.50000176689742203973
に対し,シングルジョブでは
fave=　0.50000176689742203973
8smpでは以下の様になっています.
fave=　0.50000176689738828895

実行部分のソースは以下の様になります。

```
#pragma omp parallel private(i,j,k,ib,ih,ie,ik1,ik2,ij1,ij2,ii,g05faf)
reduction(+:fave) num_threads(8)
#pragma omp for
    for (l=1;l<68719476736;l++) {
    k=l>>24 ;
    j=(l-(long int)(k<<24))>>12 ;
    i=l-(long int)(k<<24)-(long int)(j<<12) ;
    ib=klist[k] ;
    ih=jlist[j] ;
    ie=(ih*ib)&mask ;
    ih=ilist[i];
    ib=(ih*ie)&mask ;
```

次ぎに続きます。

```
if(k==0) {
  ik1=0 ;
  ik2=1 ;
  }
else {
  ik1=klist1[k-1] ;
  ik2=klist[k] ;
}
if(j==0) {
  ij1=0 ;
  ij2=1 ;
  }
else {
  ij1=jlist1[j-1] ;
  ij2=jlist[j] ;
}
if(i==0) {
  ii=0  ;
  }
else {
  ii=ilist1[i-1] ;
}
ie=b*ik1+ik2*(a*ij1+ij2*ii) ;
ih=(ib*iseed+ie*ic)&mask ;
g05faf=(double) ( ih) *r ;
fave=fave+g05faf ;
}
   fave=fave*t ;
```

MPIでの値は以下の様になっています。

混合法			
	T2K		BG/Q 32ノード
mpi数	実行時間(秒	mpi数	実行時間(秒)
16	75.631308	1	6335.624106
32	38.844884	512	12.06219
64	21.414003	1024	6.877529
128	10.84745	2048	5.518025
256	7.297119		

乗算合同法と同じく
並列化効果がはっきりでています。

4.ループ積分（数値積分）における特異点処理

ループ積分では複数の数値積分を行い,その和を最終値とする処理を行っています。

$S=\sum_{i=1}^{n} A_i$　　A_i：個々の数値積分の値,S：最終値

ここで,Sは有限値となりますが,個々のA_iでは発散する場合があります。
一般に発散する被積分関数Dに対しては、微小量$i\varepsilon$を加え,$\varepsilon->0$としたときの値で発散する積分の有限部分を取り出す操作を行います.ただし計算機では数学的な$\varepsilon->0$の操作は出来ないので,$\varepsilon_n->0$となる点列を取り,$D-i\varepsilon_n$の値の極限値を有限回の操作で求める事になります。

数学的に求まる値と計算機で求まる値

$0<a<1,\varepsilon>0$

$\int_0^1 \frac{1}{x-a}dx$：　　計算機で求まる値＝主値積分の値

$\int_0^1 \frac{1}{(x-a)^2}dx$：　　計算機で求まる値＝*Hadamard*積分の値

説明は次ページ以降に記述します。

$$\int_0^1 \frac{1}{x-a}dx$$

$$\lim_{\varepsilon->0}[\int_0^{a-\varepsilon} \frac{1}{x-a}dx+\int_{a+\varepsilon}^1 \frac{1}{x-a}dx]=\lim_{\varepsilon->0}(\ln(1-a)-\ln a)=\ln(\frac{1-a}{a})$$

$$\lim_{\varepsilon->0}[\int_0^1 \frac{x-a}{(x-a)^2+\varepsilon^2}dx]=\lim_{\varepsilon->0}[\frac{1}{2}\ln(\frac{(1-a)^2+\varepsilon^2}{a^2+\varepsilon^2})]=\ln(\frac{1-a}{a})$$

$$\int_0^1 \frac{1}{(x-a)^2}dx$$

$$\lim_{\varepsilon->0}[\int_0^{a-\varepsilon} \frac{1}{(x-a)^2}dx+\int_{a+\varepsilon}^1 \frac{1}{(x-a)^2}dx]=\lim_{\varepsilon->0}[-(\frac{1}{a}+\frac{1}{1-a})+\frac{2}{\varepsilon}]$$

有限部分$-(\frac{1}{a}+\frac{1}{1-a})$

$$\lim_{\varepsilon->0}[\int_0^1 \frac{(x-a)^2-\varepsilon^2}{((x-a)^2+\varepsilon^2)^2}dx]=\lim_{\varepsilon->0}[\frac{-(1-a)}{(1-a)^2+\varepsilon^2}-\frac{a}{a^2+\varepsilon^2}]=-(\frac{1}{a}+\frac{1}{1-a})$$

$$S_n=-(\frac{1}{a}+\frac{1}{1-a})+\frac{2}{\varepsilon_n} \quad (\varepsilon_0>\varepsilon_1>.....>\varepsilon_n->0)$$

エイトケン加速法、及びその一般形のε－算法で

は$-(\frac{1}{a}+\frac{1}{1-a})$に収束する.

エイトケン加速法

$$S_n=a+b\lambda^n \quad (|\lambda|\neq 1)$$

$$\Delta S_n=S_n-S_{n-1}=b\lambda^{n-1}(\lambda-1)$$

$$\Delta^2 S_n=\Delta S_n-\Delta S_{n-1}=b\lambda^{n-2}(\lambda-1)^2$$

$$T_n=S_n-\frac{(\Delta S_n)^2}{\Delta^2 S_n}=a+b\lambda^n-\frac{b^2\lambda^{2n-2}(\lambda-1)^2}{b\lambda^{n-2}(\lambda-1)^2}=a$$

特異点が端点の場合の例

H(*Hadamard*　　にちなむ)

$H\int_a^b f(x)dx$　　区間$[a,b]$での発散積分の有限部分.

積分の*Cauchy*の主値は発散積分の有限部分の一例

$\varepsilon-$算法は端点特異点の場合$H\int_a^b f(x)dx$　　を求めている事になる.

$f(x):[a+\varepsilon,b]$で有界,積分可能で,aの近傍で*Laurent*展開が可能。

$$f(x)=\frac{c_0}{x-a}+\sum_{r=1}^{m}\frac{c_r}{(x-a)^{\lambda_r}}+h(x)\quad[\lambda_r>1]$$

($h(x)$は$[a,b]$で積分可能)

$$H\int_a^b f(x)dx=\lim_{\varepsilon->0}[\int_{a+\varepsilon}^b f(x)dx-c_0\log\frac{1}{\varepsilon}-\sum_{r=1}^{m}\frac{c_r}{\lambda_r-1}(\frac{1}{\varepsilon})^{\lambda_r-1}]$$

$$=\int_a^b h(x)dx+c_0\log(b-a)-\sum_{r=1}^{m}\frac{c_r}{\lambda_r-1}(\frac{1}{b-a})^{\lambda_r-1}$$

$$H\int_0^1\frac{1-x}{x^3}dx=-(\frac{-1}{1}+\frac{1}{2})=\frac{1}{2}$$

$h(x)=0,c_0=0,c_1=-1,c_2=1,\lambda_1=2,\lambda_2=3$

$$a_n = \int_{\varepsilon_n}^{1} \frac{1-x}{x^3}\,dx = \frac{1}{2} + \frac{1}{2}\frac{1}{{\varepsilon_n}^2} - \frac{1}{\varepsilon_n} \qquad (\varepsilon_n = \frac{1}{2^n})$$ のε－算法の結果は

ex - 0.364048765700024415D + 04　0.104D + 10　- 0.36405D + 04
ex　0.50000000000000000D + 00　0.637D + 04　0.50000D + 00
ex　0.50000000000000000D + 00　0.147D - 25　0.50000D + 00
ex　0.50000000000000000D + 00　0.235D - 26　0.50000D + 00
ex　0.50000000000000000D + 00　0.861D - 28　0.50000D + 00
ex　0.50000000000000000D + 00　0.319D - 29　0.50000D + 00
ex　0.50000000000000000D + 00　0.281D - 29　0.50000D + 00
ie = 15　0.50000000000000000D + 00　0.47666766123584087 4D - 31

ε－算法の結果はHadamard積分の値に一致しています。

より一般的な場合の例

$\inf ra \quad box \quad s=0, \lambda=0, t=-150^2,$

$m_f = 150, m_e = 0.0005$の計算結果

$H\int_0^1 \frac{1-x}{x^3}dx = \frac{1}{2}$ とした場合の

解析解 = 44.444444507677456465204143093617

εの初期値は$1.2^{-90}, \varepsilon_n = \varepsilon_{n-1}/1.2$ で15回反復計算.

DE

eps = 0.58230485Q - 08

result = - 0.60247206492294017 3Q + 03

ex 0.44385662140120909 6Q + 02 0.755Q + 03 0.44386Q + 02

ex 0.44445514441291720Q + 02 0.122Q + 00 0.44445Q + 02

ex 0.44444440470920105Q + 02 0.461Q - 03 0.44444Q + 02

ex 0.44444444236695617Q + 02 0.348Q - 05 0.44444Q + 02

ex 0.44444446858620597Q + 02 0.352Q - 06 0.44444Q + 02

ex 0.44444444677995797Q + 02 0.451Q - 06 0.44444Q + 02

ex 0.44444444079332656Q + 02 0.901Q - 06 0.44444Q + 02

ex : 15 0.44444446858620 6Q + 02 0.225Q - 06

ループ積分においては被積分関数Dに対し，微小量i ε を加え，ε －算法により積分値を求める方法は非常に有用な手法となっています。