2020 年度卒業研究レポート
多次元の数値積分
フィボナッチ数列を利用した三次元の優良格子点法

明治大学　総合数理学部　現象数理学科　4 年
小林賢弥

2021 年 2 月 25 日

目次

1　はじめに

私は、もともとモンテカルロ法について興味があり調べていた。そこで準モンテカルロ法の優良格子点法は 2 次元の場合、モンテカルロ法と比べかなり精度の良い数値積分であると知りこの分野に興味を持った。優良格子点は、3 次元以上で (2 次元のときのフィボナッチ数列を利用する方法のような) 分かりやすい求め方は知られていなくて、ほぼしらみつぶしに近いやり方で探索して求められている (例えば鳥居・杉浦)。この研究では、2 次元の時の真似をして、フィボナッチ数列やトリボナッチ数列を使って優良格子点法と同じ式 (レポートの (2), 杉原・室田では (11.26)) で積分の近似値を求めると、どのような精度が得られるかを調べてみる。

2　数値積分

数値積分とはある領域上の関数 $f(x)$ の積分値を数値的に計算する方法のことをいう。

1 次元の場合、積分で求められる $I(=\int_a^b f(x)dx)$ を数値積分で求められる $I_n(=\sum_{k=1}^n w_k f(x_k))$ で近似していく。(w_k と x_k をそれぞれ重みと分点と呼ぶ。) 重みや分点を工夫することで精度のいい数値計算を可能にしていくことができる。しかし、多次元の数値積分になると、積分領域や被積分関数が複雑になるため 1 次元では優秀な計算方法も使えないことが多い。

そこで、多次元の数値積分でも使えるモンテカルロ法と準モンテカルロ法を使用していく。

3　モンテカルロ法

モンテカルロ法とは積分領域 (Ω) 上に一様に分布する N 個の乱数 $x_1, \cdots, x_N$ を用いて数値積分する方法ことをいう。

$$I_n = \frac{1}{N}\sum_{k=1}^{N} f(x_k) \times (\Omega \text{ の体積}) \tag{1}$$

乱数の生成方法については色々ありますが、今回は C 言語 (コンパイラー:Apple LLVM version 9.0.0 (clang-900.0.39.2)) の rand 関数を使う。rand 関数の乱数列は線形合同法を使っており、周期性がある乱数列になっている。

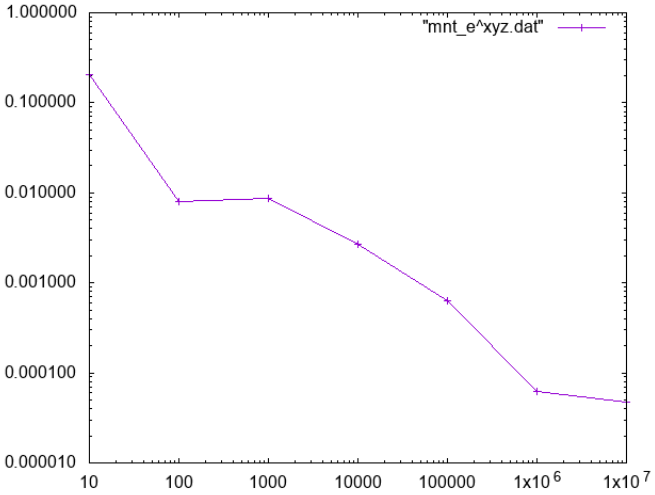

図 1　$\int\int\int_{[0,1]^3} e^{xyz}dxdydz$ の数値積分の誤差

$N = 10^7$ で誤差が 10^{-4} から 10^{-5} までの間になっている。

3.1　モンテカルロ法のソースコード

```
//モンテカルロ法による e^xyz の 0<x,y,z<1 の
//数値積分した時の N=10^k(k=1,2,...,7) の時の誤差を表示する。
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#define max (7)

double f(double x,double y,double z);
int main(){
  long long int i,j,N;
  double x,y,z;
  double T,S,F,e,I;
  //真値
  T=1.146499072528642;
  N=1;
  srand(11);
  //N=10~10^max まで
  for(j=1;j<max;j++){
    S=0;
    N=N*10;
    for(i=1;i<N;i++){
```

```
      x=(double)rand()/RAND_MAX;
      y=(double)rand()/RAND_MAX;
      z=(double)rand()/RAND_MAX;
      F=f(x,y,z);
      S=S+F;
    }
    I=S/N;
    e=fabs(T-I);
    //N と誤差を表示する
    printf("%lld %e\n",N,e);
  }
  return 0;
}

//被積分関数
double f(double x,double y,double z){
  double F=0;
  F=exp(x*y*z);
  return F;
}
```

4　優良格子点法

優良格子点法とは準モンテカルロ法の一つの方法である。準モンテカルロ法はモンテカルロ法とほとんど同様に数値計算を行っていくが、乱数の代わりに超一様乱数などを利用して数値計算を行う方法である。
優良格子点法は次のように定義されている。

$$I_N = \frac{1}{N}\sum_{k=0}^{N-1} f\left(\left\{\frac{g_1^{(N)}}{N}k\right\},\cdots,\left\{\frac{g_s^{(N)}}{N}k\right\}\right) \tag{2}$$

$\{x\}$ は小数部分を表し、g_i は分点総数 N に応じて決まる整数である。$I = \int_{[0,1]^s} f(x)dx$ に対して比較的低次元 ($s \leq 4$ 程度) で積分問題で関数が滑らかな場合有用であるとされている（杉原・室田 [1]）。ここでは、N, g_i の決め方が重要である。特に積分誤差が小さくなる条件を満たす格子点を優良格子点と呼ぶ (条件については杉原・室田 [1] を参照せよ)。

二次元の場合はフィボナッチ数列 $f_1 = 1, f_2 = 1, f_n = f_{n-1} + f_{n-2}(n > 2)$ を使って、$N = f_n, g_1 = f_1, g_2 = f_{n-1}$ が適当だとされている。（杉原・室田 [1]）また、(2) の式をただ適応するのを格子点法とここでは呼ぶことにする。

実際に適応する際は

$$I_N = \frac{1}{N}\sum_{k=0}^{N-1} f\left(\phi_p\left(\left\{\frac{g_1^{(N)}}{N}k\right\}\right),\cdots,\phi_p\left(\left\{\frac{g_s^{(N)}}{N}k\right\}\right)\right)\prod_{i=1}^{s}\phi_p'\left(\left\{\frac{g_i^{(N)}}{N}k\right\}\right) \tag{3}$$

$$\phi_p(y) = \frac{(2p+1)!}{p!p!}\int_0^y u^p(1-u)^p du$$

このように被積分関数を良い形に変えてから行うといい。$p = 5, g_1 = f_1, g_2 = f_{n-1}, N = f_n$ として変形後と前を比べてみる。

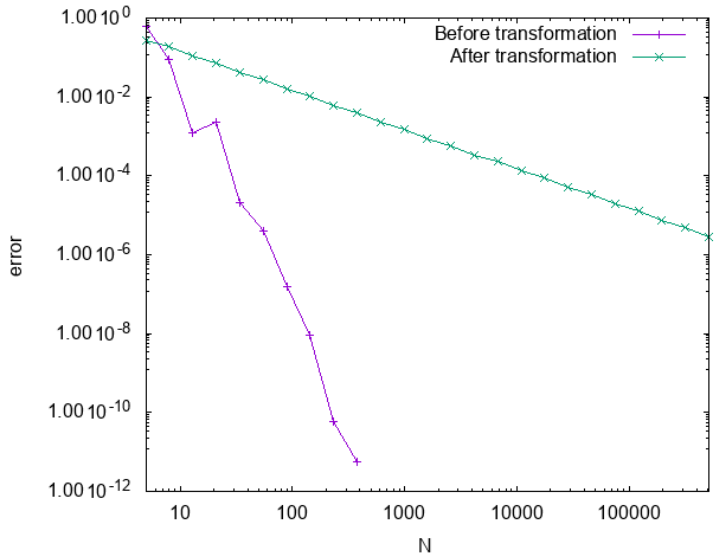

図 2　$\int\int_{[0,1]^2} e^{xy}dxdy$ の数値積分の誤差の比較

4.0.1　優良格子点法のソースコード

```
//優良格子点法による e^xy の 0<x,y<1 の
//数値積分した時の誤差を表示するプログラム。
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#define n (30)

double frac(double x,int i,int N);
double f(double x,double y);
double phi(double x);
double phid(double x);
int main(int argc,char const *argv[]){
  long long int i,j,N;
  double x,y,z;
  double g1,g2;
  double f1,f2,f3;
  double T,S,F,e,I,X,Y;
  //e^xy の積分値
  T=1.317902151454403;

  for(j=5;j<n;j++){
    f1=1;f2=1;
    for(i=1;i<j;i++){
      if(i==1){g1=f1;}
      if(i==j-1){g2=f1;}
      f3=f2+f1;
      f1=f2;
      f2=f3;
    }
    N=f1;
    S=0;
    for(i=1;i<N;i++){
      x=frac(g1,i,N);
      y=frac(g2,i,N);
      X=phi(x);
      Y=phi(y);
      //被積分関数を変形させている
```

```
      F=f(X,Y)*phid(x)*phid(y);
      //被積分関数をそのままの状態
      //F=f(x,y);
      S=S+F;
    }
    I=S/N;
    e=fabs(T-I);
    printf("%lld %e\n",N,e);
  }
  return 0;
}

//被積分関数
double f(double x,double y){
  double F=0;
  F=exp(x*y);
  return F;
}
//p=5 の時の phi の値
double phi(double x){
  double F=0;
  F=(pow(x,5)*(-252)+1386*pow(x,4)-3080*pow(x,3)+3465*x*x-1980*x+462)*pow(x,6);
  return F;
}
double phid(double x){
  double F=0;
  F=(pow(x,5)*(-252*11)+1386*10*pow(x,4)-3080*9*pow(x,3)+3465*8*x*x-1980*7*x+462*6)*pow(x,5);
  return F;
}
double frac(double x,int i,int N){
  double F;
  F=x*i;
  F=F%N;
  F=F/N;
  return F;
}
}
```

4.1　3 次元の優良格子点法

3 次元の場合は、2 次元の時のように簡単な方法で良い整列は生成でもないと考えられていてしらみつぶしの探索により N を固定して誤差が最小になるような g_i を探す必要がある。下の表は鳥居・杉浦の探索法 [2] による N に対する誤差が最小になるように g_i を求めた表である。この方法で求めた N, g_i を使用して数値計算する方法をここでは最良格子点法と呼ぶ。(Best Lattice Points)

N	g_1	g_2	g_3
185	1	26	64
266	1	27	69
418	1	90	130
597	1	63	169
828	1	285	358
1010	1	140	237
1459	1	256	373
1958	1	202	696
2440	1	638	1002
3237	1	456	1107
4044	1	400	1054
5037	1	580	1997

図 3　鳥居・杉浦の探索法 [2] による結果

表を参考にして N, g_1, g_2, g_3 を配列に保存して最良格子点法で数値計算をしてみる。

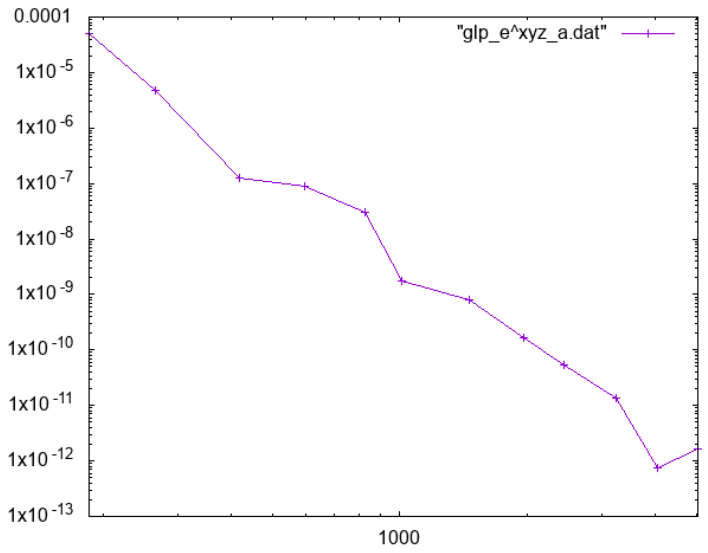

図 4　$\int\int\int [0,1]^3 e^{xyz} dxdydz$ の数値積分の誤差、(横軸:N[100 : 10000] 縦軸:誤差)

4.1.1　最良格子点法のソースコード

```
//探索法により求めた N,g_i を利用した優良格子点法による e^xyz の 0<x,y,z<1 の
//数値積分した時の誤差を表示するプログラム。
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#define n (30)

double f(double x,double y,double z);
double phi(double x);
double phid(double x);
int main(int argc,char const *argv[]){
  long long int i,j,k,N;
  double x,y,z;
  double a,b,c;
  double f1,f2,f3;
  double T,S,F,e,I,X,Y,Z;
  //探索法により求めた N,g_i を配列に入れたもの
  double MAX[12]={185,266,418,597,828,1010,1459,1958,2440,3237,4044,5037};
  double g1[12]={1,1,1,1,1,1,1,1,1,1,1,1};
  double g2[12]={20,27,90,63,285,140,256,202,638,456,400,580};
  double g3[12]={64,69,130,169,358,237,373,696,1002,1107,1054,1997};
  T=1.146499072528642;
  for(j=0;j<12;j++){
    //printf("%0.15f %f %f\n",a,b,c);
    S=0;
    N=MAX[j];
    a=g1[j]/N;
    b=g2[j]/N;
    c=g3[j]/N;
    for(i=1;i<N;i++){
      x=(double)i*a-floor((double)i*a);
      y=(double)i*b-floor((double)i*b);
      z=(double)i*c-floor((double)i*c);
      X=phi(x);
      Y=phi(y);
      Z=phi(z);
      F=f(X,Y,Z)*phid(x)*phid(y)*phid(z);
```

```
      //printf("%f\n",f(X,Y));
      S=S+F;
    }
    I=S/N;
    e=fabs(T-I);
    //printf("%lf\n",I);
    printf("%lld %e\n",N,e);
  }
  return 0;
}

//被積分関数
double f(double x,double y,double z){
  double F=0;
  F=exp(x*y*z);
  return F;
}

double phi(double x){
  double F=0;
  F=pow(x,11)*(-252)+1386*pow(x,10)-3080*pow(x,9)+3465*pow(x,8)-1980*pow(x,7)+462*pow(x,6);
  return F;
}
double phid(double x){
  double F=0;
  F=pow(x,10)*(-252*11)+1386*10*pow(x,9)-3080*9*pow(x,8)+3465*8*pow(x,7)
    -1980*7*pow(x,6)+462*6*pow(x,5);
  return F;
}
```

5　フィボナッチ数列を利用した 3 次元の格子点法

2 次元の場合はフィボナッチ数列 $f_1 = 1, f_2 = 1, f_n = f_{n-1} + f_{n-2}(n > 2)$ を使って、$N = f_n, g_1 = f_1, g_2 = f_{n-1}$ とされていたので、3 次元の場合は、$N = f_n, g_1 = f_1, g_3 = f_{n-1}$ とし g_2 を f_1 と f_{n-1} の間の値を使って数値計算を行ういいと思ったので、とりあえず、$N = f_n, g_1 = f_1, g_2 = f_{n-9}, g_3 = f_{n-1}$ とおき、$\int\int\int_{[0,1]^3} e^{xyz} dxdydz$ の数値積分を行う。

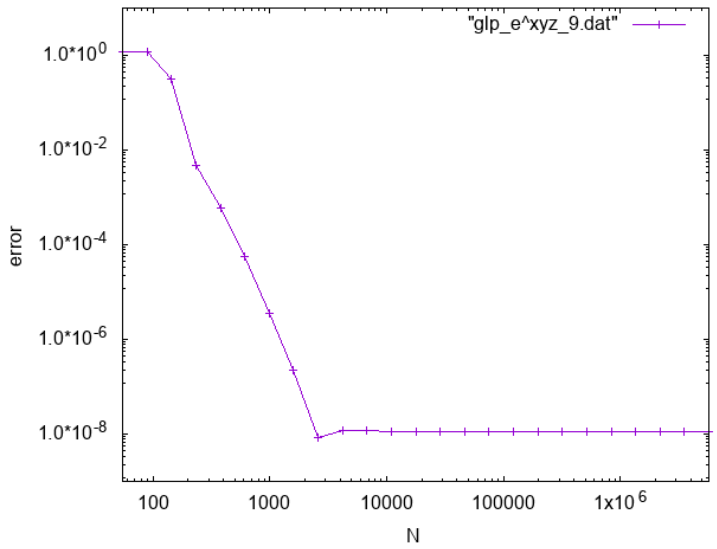

図 5　$\int\int\int_{[0,1]^3} e^{xyz} dxdydz$ の数値積分の誤差、(横軸:N 縦軸:誤差)

途中までかなりの精度を保っているが、10^3 以降精度が変化しなくなった。
ここで $g_2 = f_{n-k}$ とおき、$N = f_n, g_1 = f_1, g_2 = f_{n-k}, g_3 = f_{n-1}$ として k を 2 から 9 まで変化させてどのように変化するかを確認してみると、

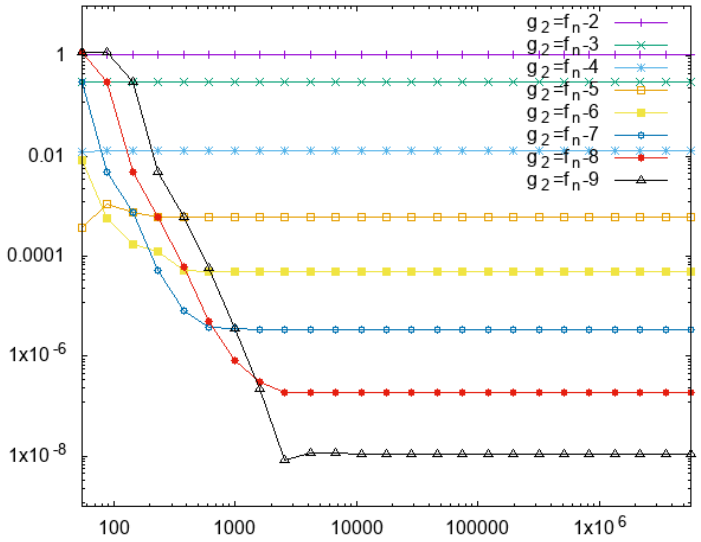

図 6　$\int\int\int_{[0,1]^3} e^{xyz}dxdydz$ の数値積分の誤差 (横軸:N 縦軸:誤差)

g_2 の値を変えることにより、精度に変化が生じることがわかった。k が大きくなると精度が高くなっている。そこで、g_2 の値を f_1 と f_{n-1} の間をとって、$N = f_n, g_1 = f_1, g_2 = f_{n/2}, g_3 = f_{n-1}$ で確認してみると、

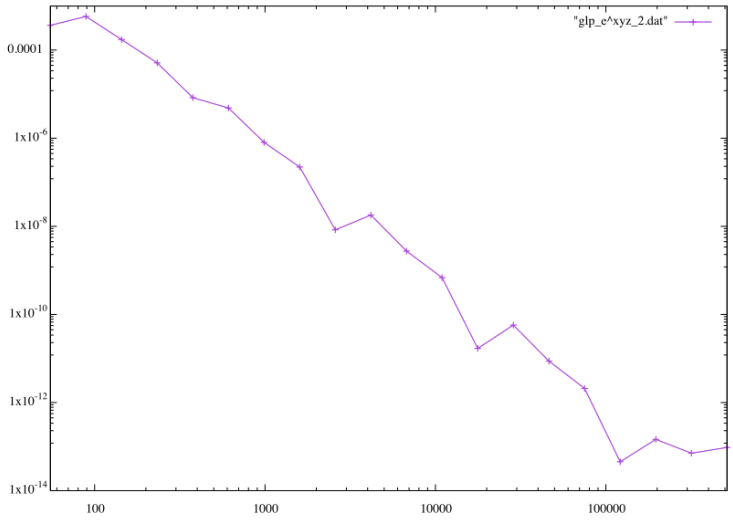

図 7　$\int\int\int_{[0,1]^3} e^{xyz}dxdydz$ の数値積分の誤差 (横軸:N 縦軸:誤差)

$N = 10^5$ で誤差が 10^{-13} になる。ここで、先ほどのモンテカルロ法、最良格子点法により求めた N, g_1, g_2, g_3 を利用した格子点法と比べてみる。

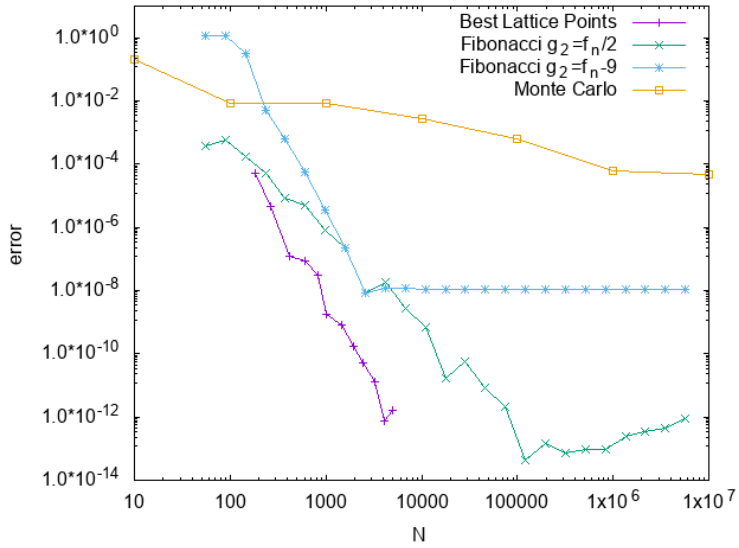

図 8　$\int\int\int_{[0,1]^3} e^{xyz} dxdydz$ の数値積分の誤差比較

一番精度が良いのは最良格子点法である。フィボナッチ数列を利用した格子点法は精度が高いが、途中から精度が悪くなるのが欠点である。

5.1　プログラムのソースコード

```
//フィボナッチ数列を利用した優良格子点法による e^xyz 0<x,y,z<1 の数値積分
//数値積分した時の誤差を表示するプログラム。
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#define max (40)

double f(double x,double y,double z);
double phi(double x);
double phid(double x);
int main(int argc,char const *argv[]){
  long long int i,j,N,k;
  double x,y,z;
  double a,b,c;
  double f1,f2,f3;
  double T,S,F,e,I,X,Y,Z;
  //真値
  T=1.146499072528642;
```

```
  k=atoi(argv[1]);
  for(j=10;j<max;j++){
    f1=1;f2=1;
    for(i=1;i<j;i++){
      //g_i の値を保存
      if(i==1){a=f1;}
      if(i==j/2){b=f1;}
      if(i==j-1){c=f1;}
      f3=f2+f1;
      f1=f2;
      f2=f3;
    }
    N=f1;
    //a,b,c の値を設定
    a=a/N;b=b/N;c=c/N;
    S=0;
    for(i=1;i<N;i++){
      x=(double)i*a-floor((double)i*a);
      y=(double)i*b-floor((double)i*b);
      z=(double)i*c-floor((double)i*c);
      X=phi(x);
      Y=phi(y);
      Z=phi(z);
      F=f(X,Y,Z)*phid(x)*phid(y)*phid(z);
      S=S+F;
    }
    I=S/N;
    e=fabs(T-I);
    printf("%lld %e\n",N,e);
  }
  return 0;
}

//被積分関数
double f(double x,double y,double z){
  double F=0;
  F=exp(x*y*z);
  return F;
}
```

```
double phi(double x){
  double F=0;
  F=pow(x,11)*(-252)+1386*pow(x,10)-3080*pow(x,9)+3465*pow(x,8)-1980*pow(x,7)+462*pow(x,6);
  return F;
}
double phid(double x){
  double F=0;
  F=pow(x,10)*(-252*11)+1386*10*pow(x,9)-3080*9*pow(x,8)+3465*8*pow(x,7)
    -1980*7*pow(x,6)+462*6*pow(x,5);
  return F;
}
```

6　その他の方法での三次元の格子点法

フィボナッチ数列とは別の数列を使って格子点法をやってみる。ここではトリボナッチ数列 $t_{n+2} = t_{n+2} + t_{n+1} + t_n$ を利用した格子点法を考えてみる。$N = t_n, g_1 = t_1, g_2 = t_{n-k}, g_3 = t_{n-1}$ として考えてみる。$2 <= k <= 9, k = n/2$ としてそれぞれグラフにしてみる。

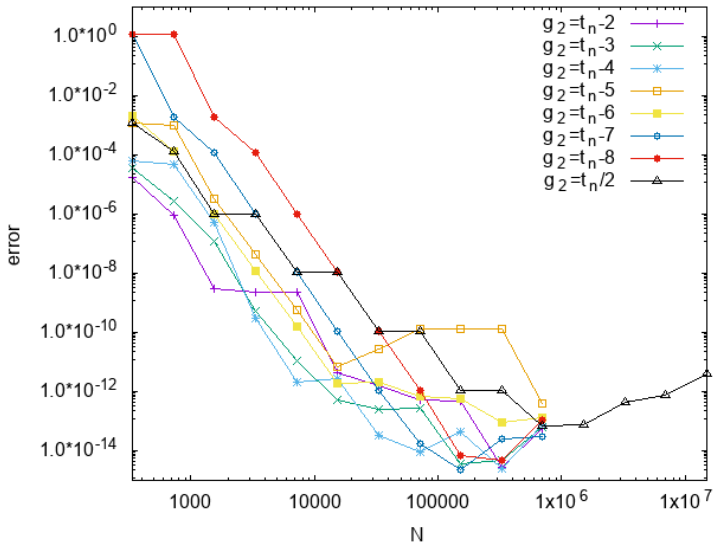

図 9　$\int\int\int_{[0,1]^3} e^{xyz} dxdydz$ の数値積分の誤差比較

フィボナッチ数列とは異なり g_2 の値の変化で大きく変化することがない。多少の違いが見られるが、$N = 10^6$ で誤差が約 10^{-12} くらいになっている。

ここで、トリボナッチ数列を利用した $(N = t_n, g_1 = t_1, g_2 = t_{n-2}, g_3 = t_{n-1})$ と置いた格子点法と、フィボナッチ数列 $(N = f_n, g_1 = f_1, g_2 = f_{n/2}, g_3 = f_{n-1})$ とおいた格子点法、最良格子点法、モンテカルロ法それぞれを比較してみる。

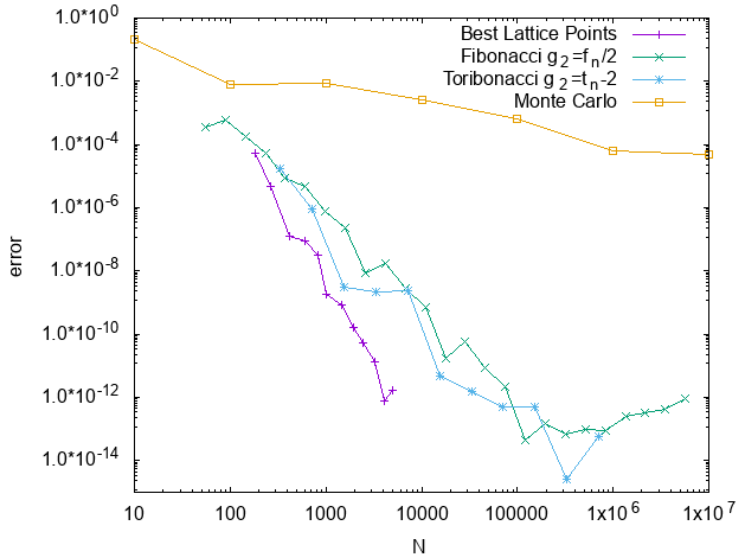

図 10　$\int\int\int_{[0,1]^3} e^{xyz} dxdydz$ の数値積分の誤差比較

最良格子点法の方が精度がよく、フィボナッチ数列とトリボナッチ数列の格子点法はそこまで大きな差はない。

6.1　プログラムソースコード

```
//トリボナッチ数列を利用した優良格子点法による e^xyz 0<x,y,z<1 の
//数値積分した時の誤差を表示するプログラム。
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>
#define max (21)

double f(double x,double y,double z);
double phi(double x);
double phid(double x);
int main(int argc,char const *argv[]){
  long long int i,j,N,k;
  double x,y,z;
  double a,b,c;
  double f1,f2,f3,f4;
  double T,S,F,e,I,X,Y,Z;
  T=1.146499072528642;
  k=atoi(argv[1]);
```

```
  for(j=10;j<max;j++){
    //各値を設定する（トリボナッチ）
    f1=1;f2=1;f3=1;
    for(i=1;i<j;i++){
      //a,b,c の値を保存
      if(i==1){a=f1;}
      if(i==j-k){b=f1;}
      if(i==j-1){c=f1;}
      f4=f3+f2+f1;
      f1=f2;
      f2=f3;
      f3=f4;
    }
    N=f1;
    a=a/N;b=b/N;c=c/N;
    //printf("%0.15f %f %f\n",a,b,c);
    S=0;
    for(i=1;i<N;i++){
      x=(double)i*a-floor((double)i*a);
      y=(double)i*b-floor((double)i*b);
      z=(double)i*c-floor((double)i*c);
      X=phi(x);
      Y=phi(y);
      Z=phi(z);
      F=f(X,Y,Z)*phid(x)*phid(y)*phid(z);
      //printf("%f\n",f(X,Y));
      S=S+F;
    }
    I=S/N;
    e=fabs(T-I);
    //printf("%lf\n",I);
    printf("%lld %e\n",N,e);
  }
  return 0;
}

//被積分関数
double f(double x,double y,double z){
  double F=0;
  F=exp(x*y*z);
```

```
  return F;
}

double phi(double x){
  double F=0;
  F=pow(x,11)*(-252)+1386*pow(x,10)-3080*pow(x,9)+3465*pow(x,8)-1980*pow(x,7)+462*pow(x,6);
  return F;
}
double phid(double x){
  double F=0;
  F=pow(x,10)*(-252*11)+1386*10*pow(x,9)-3080*9*pow(x,8)+3465*8*pow(x,7)
    -1980*7*pow(x,6)+462*6*pow(x,5);
  return F;
}
```

7 まとめ

最良格子点法は g_i, N を求めるのに時間がかかってしまう欠点があるので、より精度を高くしようとするそれだけ時間がかかってしまう。しかし、精度は他のやり方に比べ高い。3 次元の数列を利用する格子点法は、非常に簡単に g_i, N を求めることができる。その反面最良格子点法と比べると精度は低くなる。

N を多くしてでもより誤差を小さくしたい場合は数列を利用する方法を使い、小さい N で誤差を最小にしたい場合は最良格子点法を使う方が良い。

数列を利用する格子点法は他の数列を利用することで、さらなる可能性があると感じた。また、最良格子点法の仕方についても時間がかからないやり方を考えることができれば、時間が短縮された精度のいい数値積分が可能になるのではないかと考えている。

参考文献

[1] 杉原正顕・室田一雄、数値計算法の数理、岩波書店、1994

[2] 鳥居久訓，杉浦洋: 3,4，5，6 次元の GLP の探索について，日本応用数理学会論文誌, Vol. 3 (1993), pp. 157-175.