計算実験序論 2　実習資料

担当教員　三浦　伸一

平成 28 年度版

実習課題について

この資料は大きく分けて二つの部分からなっています。最初の 3 章は C 言語の入門に、残りの 3 章は乱数とそれを利用した酔歩の計算機シミュレーションに充てられています。そのなかでいくつかの実習課題が出題されています。課題には「必須」、「発展」と「ラベルなし」の 3 種類があります。「必須」とある課題は必ずレポートを作成してください。「ラベルなし」の課題も是非とも取り組んでください。また「発展」とある課題は、余力があれば挑戦して欲しい問題です。

レポートには必ず

学年　所属　名列番号　氏名

を記入してください。作成したレポートは、ひとつの PDF ファイルにまとめ、アカンサスポータルの「計算実験序論２」のウェブページから提出すること。レポート提出の締め切りは、

平成 29 年 2 月 12 日（日）

とします。なお、何らかの理由でレポートを電子的に提出できない場合は、紙媒体に出力したものも受け付けます。

目次

1　C 言語入門 I

1.1　はじめに

このノートでは網羅的な C 言語の文法に関する解説はしませんので、適宜以下の教科書などを参照すること（“習うより慣れろ”的スタンスです)。入門書としては、1) がおすすめです。

1) 柴田望洋「新版　明解 C 言語　入門編」(ソフトバンククリエイティブ)
2) 柴田望洋「新版　明解 C 言語　実践編」(ソフトバンククリエイティブ)
3) B. W. カーニハン、D. M. リッチー「プログラミング言語 C」(共立出版)
4) 藤原博文「C プログラミング専門課程」(技術評論社)
5) おまけ（楽しい本です)：　藤原博文「C プログラミング診断室」(技術評論社)

1.2　最も簡単な例

Hello, world!という文字列を画面に出力しましょう (かならず最初にやることになっています)。

1）hello.c というファイルを作成

```
#include <stdio.h>   // おまじない　ヘッダーファイルといいます
int main(void)
{
    printf(“Hello, world!\n”);  // 標準出力（ターミナル）へ出力
    return 0;
}
```

2） コンパイルをする（コンパイラのオプションについては付録 B を参照のこと）

```
% gcc -Wall -o hello hello.c
```

3） コンパイルが成功したら、hello という実行ファイルができるので、実行する。

```
% ./hello
```

1.3　簡単な四則演算

1）まず test1.c というファイルを作成しましょう。

```
#include <stdio.h>
int main(void)
{
    printf("%d\n", 5+2);       //  足し算   %d は整数型
    printf("%d\n", 5-2);       //  引き算
    printf("%d\n", 5*2);       //  かけ算
    printf("%d\n", 5/2);       //  割り算 1
    printf("%f\n", 5.0/2.0);   //  割り算 2   %f は浮動小数点型

    return 0;
}
```

2）コンパイル、実行。

```
% gcc -Wall -o test1 test1.c
% ./test1
```

割り算 1 と割り算 2 の違いに注意してください。

1.4　変数を使う

test2.c というファイルを作り、実行してみましょう。

```
#include <stdio.h>
int main(void)
{
    int a, b;           // 整数型変数
    double c, d;        // 浮動小数点型変数（倍精度）

    a = 5;      b = 2;
    c = 5.0;    d = 2.0;
    printf("%d  %f\n", a+b, c+d);
    printf("%d  %f\n", a-b, c-d);
    printf("%d  %f\n", a*b, c*d);
    printf("%d  %f\n", a/b, c/d);
    return 0;
}
```

もう少し変数の使い方を練習してみます（test3.c というファイルを作る）。

```
#include <stdio.h>
int main(void)
{
    double a, b;
    double wa, sa, seki, syo;

    a = 5.0;    b = 2.0;
    wa = a+b;   sa = a-b;
    seki = a*b; syo = a/b;

    printf("%f  %f  %f  %f\n", wa, sa, seki, syo);

    return 0;
}
```

1.5 標準入出力

標準入力から数値を入力して四則演算をするプログラムを作りましょう（test4.c）。

```
#include <stdio.h>
int main(void)
{
    double a, b, wa, sa, seki, syo;
    scanf("%lf", &a);    // 標準入力から入力  %lf は浮動小数点型
    scanf("%lf", &b);    // 変数名の前にある&に注意

    wa = a+b;    sa = a-b;
    seki = a*b;    syo = a/b;

    printf("%f  %f  %f  %f\n", wa, sa, seki, syo);

    return 0;
}
```

入出力に多少凝ってみましょう（test5.c)。

```
#include <stdio.h>
int main(void)
{
    double a, b;
    puts("Input two real numbers");    // 文字列を標準出力へ出力
    printf("Real number a:");  scanf("%lf", &a);
    printf("Real number b:");  scanf("%lf", &b);

    printf("a + b = %f\n", a+b);
    printf("a - b = %f\n", a-b);
    printf("a * b = %f\n", a*b);
    printf("a / b = %f\n", a/b);

    return 0;
}
```

課題 1（必須）　三つの実数を入力して、合計値と平均値を画面に出力するプログラムを作成しなさい。

課題 2（必須）　底辺と高さを入力して、三角形の面積を画面に出力するプログラムを作成しなさい。

2　C 言語入門 II

2.1　2 章の内容

- プログラムの流れの繰り返し: for 文、do 文
- プログラムの流れの分岐: if 文
- 配列を使う

2.2　for 文の使い方

for 文

```
for（式 1 ; 式 2 ; 式 3）{
  文;
}
```

例 1: 1 を 10 回たしあわせる。

```
#include <stdio.h>
int main(void)
{
    int i, sum=0;  // 変数の初期化も行う
    for (i = 0; i < 10; i = i+1)  {
        sum = sum +1;
    }
    printf( “Sum = %d\n” , sum);
    return 0;
}
```

例 2: ちょっと手を抜く。

```
#include <stdio.h>
int main(void)
{
  int i, sum=0;
  for (i = 0; i < 10; i++) {  //  通常はこのように書きます
    sum += 1;  // sum = sum + 1 と同じ。
  }            // sum++ としてもよい（整数型のみ)。
  return 0;
}
```

後置増分演算子 a++（整数 a の値を一つだけ増やす）
後置減分演算子 a--（整数 a の値を一つだけ減らす）
複合代入演算子　a += b　←　a = a + b
　　a -= b　←　a = a - b

例 3: 1 を n 回たしあわせる ($\sum_{i=1}^{n} 1$)。

```
#include <stdio.h>
int main(void)
{
  int i, n, sum=0;
  scanf("%d", &n);
  for (i = 0; i < n; i++) {  // 慣用句
    sum++;
    printf("i = %d  Sum = %d\n", i+1, sum);
  }
  return 0;
}
```

例 4: 1 から n までの自然数を足しあわせる ($\sum_{i=1}^{n} i$)。

```
#include <stdio.h>
int main(void)
{
  int i, n, sum=0;
  scanf("%d", &n);
  for (i = 0; i < n; i++) {
    sum += (i + 1);  // i の現在の値に注意 i++ は最後に実行される
    printf("i = %d  Sum = %d\n", i+1, sum);
  }
  return 0;
}
```

2.3 do 文の使い方

do 文

```
do {
  文;
} while(式);
```

2.2 節の例 4（1 から n までの自然数を足しあわせる）を do 文を使って実行してみましょう。

```
#include <stdio.h>
int main(void)
{
  int i=0, n, sum=0;
  scanf("%d", &n);
  do {
    sum += (i + 1);
    printf("i = %d  Sum = %d\n", i+1, sum);
    i++;
  } while (i < n);  // 回数だけでなく一般的な条件の場合にも使える
    return 0;
}
```

2.4 if 文の使い方

if 文

```
if (式) {
  文;
}
```

例 1: 整数を入力して、正の数なら「Number is positive」と画面に出力する。

```
#include <stdio.h>
int main(void)
{
  int n;
  scanf("%d", &n);
  if (n > 0) {
    printf("Number is positive.\n");
  }
  return 0;
}
```

例 2: 整数を入力して正の数なら「Number is positive」、0 なら「Number is zero」、さもなくば「Number is negative」と画面に出力する。

```
#include <stdio.h>
int main(void)
{
  int n;
  scanf("%d", &n);
  if (n > 0) {
    printf("Number is positive.\n");
  } else if (n == 0) {  // 条件式に注意
    printf("Number is zero.\n");
  } else {
    printf("Number is negative.\n");
  }
  return 0;
}
```

条件式の色々

```
整数型変数 n
if (n > 0),  if (n >= 0), if (n < 0), if (n <= 0),
if (n == 0), if (n != 0) etc…

浮動小数点型変数 x
if (x > 0.0), if (x >= 0.0), if (x < 0.0), if (x <= 0.0) etc…
```

2.5 配列の使い方

ファイルからデータを読み込み、プログラム内で配列を使ってそのデータの処理をしてみましょう。以下の 10 人分の体重が記入されたファイルを用意します（ファイル名：weight.dat）。

```
55.2  98.5  66.3  55.1  48.3
61.1  50.0  68.3  48.8  51.1
```

このデータを標準入力から読み込み、体重の平均値を計算するプログラムを作りましょう。

```
#include <stdio.h>
#define N 10  // 定数を定義する
int main(void)
{
    int i;
    double weight[N];  // サイズ N の配列: weight[0], …, weight[N-1]
    double av_weight = 0.0;
```

```
    for (i = 0; i < N; i++) {
        // 標準入力からデータを 1 つずつ読み込む &を忘れないように
        scanf("%lf", &weight[i]);
    }

    for (i = 0; i < N; i++) {
        av_weight += weight[i];  // すべての人の体重を足しこむ
    }

    printf("Average weight = %f\n", av_weight/N);  // 平均値を出力

    return 0;
}
```

C 言語では配列の添字は 0 から始まるために、サイズ N の配列の場合、添字は N-1 までになります。上の例で weight[N] と宣言された配列の要素は、weight[0], …, weight[N-1] となります（確かに N 個ありますね！）。この点はフォートランとは異なりますので注意が必要です[*1]。実行ファイル名を av_weight とすると、以下のようにすれば体重の平均値が画面に出力されます[*2]。

```
% ./av_weight < weight.dat
```

課題 **3**（必須）

1) 自然数 n を入力して次の計算を実行し、結果を画面に出力するプログラムを作成しなさい：$\sum_{i=1}^{n} x_i$. ここで $x_i = 0.1 \times i$ と定義されています。x_i は実数であることに注意すること。

2) 整数 n を入力し、その階乗 $n! = n \times (n-1) \times \cdots \times 1$ を出力するプログラムを作成しなさい[*3]。但し $0! = 1$ と負の整数では階乗は定義されないことに注意してください（場合分けをして処理すること）。

[*1] ちなみに上のプログラムは配列の使用法の例として作ったので、単に体重の平均値が欲しい場合は配列を用意する必要はありません （わかりますか？）。

[*2] ファイルを使った標準入出力の制御は付録 C を参照してください。

[*3] 階乗は n の増大と共に急速に大きな値を持つようになります。int で宣言した場合は、通常の環境では $n = 13$ 以上でオーバーフローします。

3　C 言語入門 III

3.1　3 章の内容

関数を使う、関数を作る

3.2　数値積分の例題

まず台形公式 (trapezoidal rule) を用いた数値積分の問題を考えてみましょう。積分を以下の和で近似し数値的に実行する方法を台形公式と呼びます。

$$\int_{x_{\min}}^{x_{\max}} dx f(x) = \sum_{i=1}^{N} \frac{h}{2} \left\{ f(x_i) + f(x_{i+1}) \right\}.$$

ここで $h = (x_{\max} - x_{\min})/N$, $x_i = x_{\min} + (i-1) \times h$ と定義されています。

1) 台形公式の意味を図など用いながら説明してください。

2) 積分 $\int_0^1 dx x^2$ を台形公式を用いて数値的に実行してください。$N = 100$ とし、計算値を解析解から得られる値と比較してください。また N の値を色々と変えて数値積分を実行し、その精度について検討してください。

3) N を段階的に変えて数値積分の精度を系統的に上げていく方法について考えてみましょう。まず 1 ステップ目は $N = 2$ とし、2 区間のみを用いて数値積分を実行します。次のステップでは各区間を半分にして同様に数値積分を実行します。各ステップで各区間を半分にするという操作を繰り返していくと ℓ ステップ目には $N = 2^\ell$ の区間を使って数値積分を実行することになります。ステップが進む毎に数値積分の精度はあがっていくはずであり、前のステップの積分値と現在のステップでの積分値の差の絶対値が 10^{-6} 以下になることをこの繰り返し計算の終了条件とします。このプログラムを作成しなさい。

   ヒント: N は 1 ステップ進むごとに 2 倍になります。do 文を使うと簡単に実装できます。

プログラム例

```
#include <stdio.h>
int main(void)
{
  int i=0, L=0, N=1;
  double h, x1, x2, sum, sum_prev=0.0, diff;

  do {
      L++;   N = 2*N;   h = 1.0/N;

      // 台形公式の部分
      sum = 0.0;
      for (i = 0; i < N; i++) {
          x1 = i*h;  x2 = (i+1)*h;
          sum += 0.5*h*(x1*x1 + x2*x2);
      }

      // 前回のステップでの積分値との差の絶対値
      diff = sum - sum_prev;
      if (diff < 0.0) {
          diff = -diff;
      }

      printf("%d step: Numerical = %f, Exact = %f\n", L, sum, 1.0/3.0);
      sum_prev = sum;

  } while (diff > 1.0e-6);  // 終了条件のチェック

  return 0;
}
```

このプログラムを関数を使って、書きかえてみましょう。

3.3 手続きの単位をまとめて関数をつくる

1）$f(x)$ の引数を与えて、その関数値を返す関数 func()

```
double func(double x)
{
    double fx;
    fx = x*x;
    return fx;
}
```

2）台形公式の部分をまとめる: trapez()

```
double trapez(int n, double x_min, double x_max)
{
    int i;
    double h, x1, x2, sum=0.0;
    h = (x_max - x_min)/(double) n;
    for (i = 0; i < n; i++) {
        x1 = x_min + i*h;    x2 = x_min + (i+1)*h;
        sum += 0.5*h*(func(x1) + func(x2));
    }
    return sum;
}
```

3）実数 a を引数とし、その絶対値を返す関数 my_fabs()

```
double my_fabs(double a)
{
    if (a < 0.0) {
        a = -a;
    }
    return a;
}
```

これらの関数を使うと、最初のプログラムは次のように書きかえることができます。

```
#include <stdio.h>

double func(double x)  {
  double fx;
  fx = x*x;
  return fx;
}

double trapez(int n, double x_min, double x_max) {
  int i;
  double h, x1, x2, sum=0.0;
  h = (x_max - x_min)/(double) n;
  for (i = 0; i < n; i++) {
    x1 = x_min + i*h;    x2 = x_min + (i+1)*h;
    sum += 0.5*h*(func(x1) + func(x2));
  }
  return sum;
}

double my_fabs(double a)  {
  if (a < 0.0) a = -a;
  return a;
}

int main(void)
{
  int L = 0, N=1;
  double x_min=0.0, x_max=1.0, sum, sum_prev=0.0, diff;
  do {
    L++;   N = 2*N;
    sum = trapez(N, x_min, x_max);
    diff = sum - sum_prev;    sum_prev = sum;
    printf("%d step: Numerical = %f, Exact = %f\n", L, sum, 1.0/3.0);
  } while (my_fabs(diff) > 1.0e-6);
  return 0;
}
```

3.4 標準ライブラリにある数学関数を使う

実数の絶対値を返す関数は標準ライブラリにあります。ライブラリにあるものは、是非ともその関数を利用するようにしましょう。

```
#include <stdio.h>
#include <math.h>  // 新しいヘッダーファイル!
                   // 数学ライブラリを使う時は、これを読み込む
int main(void)
{
  double a = -1.0;
  a = fabs(a);
  printf("Absolute value = %f\n", a);
  return 0;
}
```

コンパイル時に数学ライブラリをリンクすることを忘れないこと[*4]。上記プログラムのファイル名を `test_abs.c` とすると以下のようになります。

```
% gcc -Wall -lm -o test_abs test_abs.c
% ./test_abs
```

課題 4（必須） 積分 $\int_0^{10} \left\{x^4 + 2.0 \times x^2\right\}$ を台形公式を用いて計算するプログラムを上記のプログラムを改良して作りなさい。ここで繰り返し計算の終了条件は、前のステップの積分値と現在のステップでの積分値の差の絶対値が 10^{-3} 以下としましょう。

課題 5（発展） 方程式 $f(x) = 0$ の数値解を二分法と呼ばれる方法を使って求めることを考えましょう。二分法とは、2 点の間に解が一つあることがわかっている場合、解の存在範囲を逐次半分にして数値解を求める方法です。具体的には以下の手順により数値解を求めます：

1) あいだに解が一つ存在する 2 点 (x_1, x_2) を設定します。ここで $x_1 < x_2$ とします。
2) 2 点の中点 $x = (x_1 + x_2)/2$ における $f(x)$ を計算します。
3) $f(x)$ の符号が左端 x_1 における $f(x_1)$ の符号と等しければ、まだ $f(x)$ は 0 に達していないので、この中点を新たに左端とします $(x_1 \leftarrow x)$。逆に左端と異なる符号を持てば、既に 0 を通過しているのでこの中点を右端とします $(x_2 \leftarrow x)$。2) に戻ります。

$f(x) = x^2 - 4x + 1$ とし、0 と 2 の間にある解を二分法により求めるプログラム作成してください。なお、中点における $f(x)$ の絶対値が 10^{-5} 以下となることを終了条件とし、その時の x を数値解として出力してください。

[*4] コンパイラのオプションについては付録 B を参照してください。

4　一様乱数と正規乱数

課題 6（必須）　標準ライブラリ関数 `rand()` を用いて、区間 $[0,1)$ での一様乱数のプログラムを作成しなさい。またそのプログラムがこの区間の一様乱数を生成していることを図示しなさい。

課題 7（必須）　中心極限定理を用いた簡便法およびボックス・ミュラー法を用いて、正規乱数を生成するプログラムを作成しなさい。またそれぞれの方法で生成した乱数が標準正規分布の確率密度関数に従っていることがはっきりとわかるように図示しなさい。

4.1　一様乱数

コンピュータで生成する乱数の中で最も基本的なものは、区間 $[0,1)$ での一様乱数です。連続的確率変数 x が以下の確率密度関数 $f(x)$

$$f(x) = \begin{cases} 1 & (0 \leq x < 1) \\ 0 & (x < 0, x \geq 1) \end{cases}$$

に従うとき、区間 $[0,1)$ での一様乱数と言います。確率密度関数が区間内で x の値に依らず一定であることが一様乱数と呼ぶ所以でしょう。

まず、この一様乱数の生成に必要な標準ライブラリ関数 `rand()` の説明をします。この関数は 0 から定数 `RAND_MAX` まで、つまり区間 $[0,$ `RAND_MAX`$]$ にある整数をランダムに返す関数です。関数の宣言と `RAND_MAX` の具体的な値は`"stdlib.h"`で与えられているので、このヘッダーファイルをプログラムの先頭で読み込まなくてはなりません。この関数 `rand()` を以下のように使えば、簡単に区間 $[0,1)$ での一様乱数を生成することができます（これを `uniform_rn` としましょう）。

```
uniform_rn = rand()/(RAND_MAX+1.0);
```

ここで `RAND_MAX` に 1.0 を加えていることに注意して下さい[*5]。上記の操作を多数回実行すれば、生成される実数は、区間 $[0,1)$ で一様に分布しているはずです。なお乱数列を初期化するために、プログラムの最初に関数 `srand()` を実行することを忘れないでください。乱数列の種、つまり初期値を設定します（関数の引数として渡します）。種が同じ場合は、常に同じ乱数列が生成されます。

[*5] 加えなくても間違いではありませんが、区間の端が異なります（区間 $[0,1]$ になる）。

次に生成された実数が区間内に一様に分布していることを示しましょう。このために区間全体を幅 Δx の小区間に分割し、乱数を生成するたびにその実数が属する区間をチェックしてヒストグラムを作成します。

```
for (i = 0; i < N_MAX; i++) {
    uniform_rn = rand()/(RAND_MAX+1.0);
    ibin = (int) (uniform_rn/dx);  // (int) は、整数型への型変換
    hist[ibin] += 1.0;
}
```

上記のように N_MAX 回乱数を生成すると、配列 hist[ibin] の各要素には、対応する区間の実数が現れた回数が入っています。勿論、この時点で配列の要素をすべて足せば N_MAX になっているはずです。図示するのは確率密度関数としたいので、以下のような規格化を最後にしておきます。

```
for (i = 0; i < NGRID; i++)
    hist[i] = hist[i]/N_MAX/dx;
```

ここで NGRID は、配列 hist[i] のサイズです。区間の幅 Δx で割ることも忘れないでください。

このような連続変数のヒストグラムは、今後もよく出てきますので、ここでしっかりと作成法を身につけておきましょう。付録 G にあげたプログラムを用いて (N_MAX $=$ 10000)、分布を計算した例が図 4.1 に示されています。

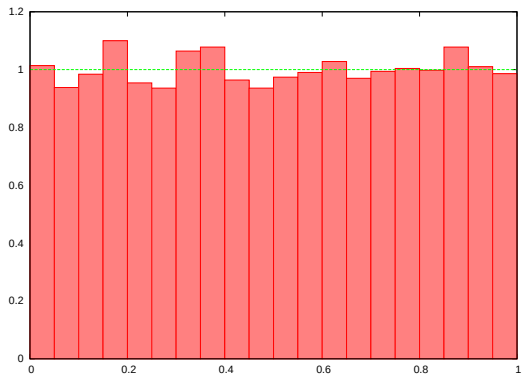

図 4.1　区間 $[0, 1)$ での一様乱数

4.2　正規乱数

一様乱数に対して、標準正規分布 $N(0,1)$ に従う乱数のことを正規乱数と呼びます。正規乱数は、前節で作成した区間 $[0,1)$ での一様乱数を用いて生成することができます。ここでは中心極限定理を用いた簡便な処方と、一様乱数を正規分布に従うような変数に変換することにより生成する方法（ボックス・ミュラー法）について説明しましょう。

4.2.1　中心極限定理を用いた簡便法

まず独立な確率変数をいくつか用意し、その和で新たな確率変数を導入しましょう。中心極限定理は、和をとる変数を増やすことによりその新しい確率変数が正規分布に従うようになるということを保証しています。ここでは、「一様乱数を 12 個加えて 6 を引く」とする簡単な処方に従いましょう。区間 $[0,1)$ での独立な一様乱数を 12 個用意します。それぞれを $x_1,\ldots,x_{12}$ と表記します。各確率変数の平均と分散は、

$$\langle x_i\rangle = \int_0^1 x_i dx_i = \frac{1}{2}$$
$$\Big\langle (x_i - \langle x_i\rangle)^2 \Big\rangle = \big\langle x_i^2\big\rangle - \langle x_i\rangle^2 = \int_0^1 x_i^2 dx_i - \langle x_i\rangle^2 = \frac{1}{12}, \qquad i = 1,\ldots,12$$

となります。これを使って新しい確率変数 y を以下のように導入します:

$$y = x_1 + \cdots + x_{12} - 6.$$

変数 y は、上述の「一様乱数を 12 個加えて 6 を引く」としたものです。この確率変数の平均と分散を計算してみましょう[*6]。

$$\text{平均}\quad \langle y\rangle = \left\langle \sum_{i=1}^{12} x_i - 6 \right\rangle = \sum_{i=1}^{12} \left\{ \langle x_i\rangle - \frac{1}{2} \right\} = 0$$

$$\text{分散}\quad \big\langle y^2\big\rangle = \left\langle \left(\sum_{i=1}^{12} \left(x_i - \frac{1}{2} \right) \right)^2 \right\rangle = \sum_{i=1}^{12} \left\langle \left(x_i - \frac{1}{2} \right)^2 \right\rangle = 1.$$

平均 $\mu = 0$、分散 $\sigma^2 = 1$ となりました。これで確率変数 y は標準正規分布に従ってくれそうです。

[*6] 付録 E を参照のこと。

例えば以下のようにすると、確率変数 y の値を一つ生成することができます。

```
y = 0.0;
for (j = 0; j < N_RN; j++) y += rand()/(RAND_MAX+1.0);
y -= 6.0;
```

ここで N_RN $= 12$ です。

さて、一様乱数のプログラム例では、乱数列の種（変数名 seed）はプログラムの中に直接数値を書きました。種が同じであれば、何度実行しても同じ乱数列を生成します。プログラムを実行するたびに、種を自動的に変えたい場合、次の技巧がよく使われます[*7]。

```
srand((unsigned int) time(NULL));
```

関数 time() は引数が NULL の場合、現在のカレンダー時間を返します[*8]。関数 srand の引数は unsigned int 型なので time の返り値をこの型へ変換しています。慣れないうちは、この部分はおまじないだと思ってもらって結構です。このライブラリ関数は "time.h"で宣言されているので、プログラムの先頭でこのヘッダーファイルを読み込むことを忘れないでください。

図 4.2 に簡便法を使って生成された乱数 1 万個の分布が示してあります。付録 G のプログラム例を参考にしてください。なお、標準偏差を σ とすると大凡 $\pm 3 \sim 4\sigma$ 程度の範囲でヒストグラムを作成すれば十分です（理由が説明できますか？）。

4.2.2 ボックス・ミュラー法

もう一つの正規乱数の生成法としてボックス・ミュラー法を取り上げます。まず区間 $[0, 1)$ での一様乱数を 2 個用意します。それぞれ x_1, x_2 と表記します。この一様乱数を使って以下の変換により、新しい確率変数 y_1, y_2 を導入しましょう。

$$y_1 = \sqrt{-2\log x_1}\cos(2\pi x_2)$$
$$y_2 = \sqrt{-2\log x_1}\sin(2\pi x_2).$$

この変数 y_1, y_2 はそれぞれ標準正規分布に従います。実際、変換前後で対応する区間での確率は同じなので、二つの一様乱数の同時分布関数を $f(x_1, x_2) = f(x_1)f(x_2)$ とすると

$$f(x_1, x_2)dx_1dx_2 = g(y_1, y_2)dy_1dy_2 \tag{4.1}$$

[*7] デバッグの際には種を固定したほうが、むしろわかりやすくなります。

[*8] 関数 time は 1970 年 1 月 1 日からの経過時間を秒単位で返します。

`double sqrt(double x)`	x の平方根
`double sin(double x)`	x の正弦
`double cos(double x)`	x の余弦
`double log(double x)`	自然対数

表 1　数学ライブラリ関数の例

となります。ここで $g(y_1, y_2)$ は、変換後の変数が従う確率密度です。この関数は、変換のヤコビ行列式を J とすると、

$$g(y_1, y_2) = f\left(x_1\left(y_1, y_2\right), x_2\left(y_1, y_2\right)\right)|J| = \frac{1}{\sqrt{2\pi}} e^{-\frac{y_1^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y_2^2}{2}}$$

となり、確かに y_1, y_2 は独立に標準正規分布に従うことがわかります。ヤコビ行列式は、

$$x_1 = e^{-\frac{1}{2}\left(y_1^2 + y_2^2\right)}$$
$$x_2 = \frac{1}{2\pi} \tan^{-1} \frac{y_2}{y_1}$$

に注意すれば、

$$J = \frac{\partial(x_1, x_2)}{\partial(y_1, y_2)} = \begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = -\frac{1}{\sqrt{2\pi}} e^{-\frac{y_1^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{y_2^2}{2}}$$

と計算できます[*9]。

このプログラムを作成するためには、表 1 にある数学ライブラリ関数を使う必要があります。関数の引数はすべて倍精度の実数です。これらの数学関数は、`"math.h"`で宣言されているので、このヘッダーファイルをプログラムの先頭で読み込まなくてはなりません。また円周率 π もこのヘッダー内で数値が与えられています。定数名は、`M_PI` です。

[*9] $(\tan^{-1} x)' = 1/(x^2 + 1)$

ここで正規乱数を返す関数 gasdev() を作ってみましょう。

```
double gasdev(void)
{
  double x1, x2, y1, y2;

  x1 = rand()/(RAND_MAX+1.0);
  x2 = rand()/(RAND_MAX+1.0);
  y1 = sqrt(-2.0*log(x1))*cos(2.0*M_PI*x2);
  y2 = sqrt(-2.0*log(x1))*sin(2.0*M_PI*x2);

  return y1;
}
```

$y2$ は利用していないのでもったいないのですが、簡単のためこうしておきます[*10]。

この方法で乱数を 1 万個生成し、その乱数が従っている分布を計算したものが図 4.3 に示してあります。付録 G にプログラム例をあげておきますので、参考にしてください。

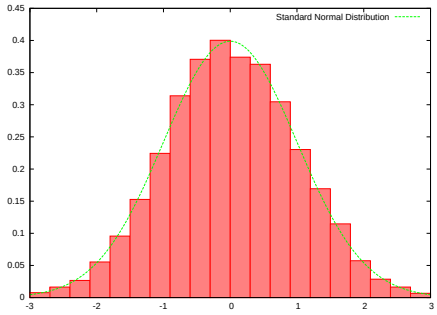

図 4.2　簡便法による正規乱数。緑の破線は正規分布の確率密度関数。

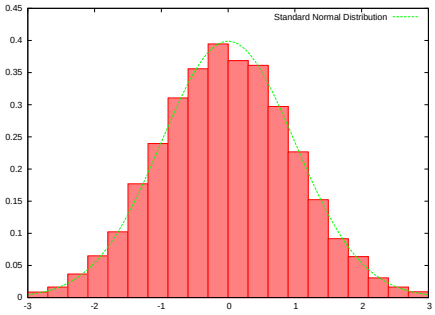

図 4.3　ボックス・ミュラー法による正規乱数。緑の破線は正規分布の確率密度関数。

[*10] $y2$ を利用する方法もあるのですが、多少技巧的なので別の機会に譲ります。関心のある人は、W. H. Press 他「Numerical Recipes in C［日本語版］」技術評論社の 217 ページを参照してください。

5　一次元格子点上の酔歩

課題 8（必須）　酔っぱらいが一次元格子点上を移動しています。各時刻（各ステップ）で酔っぱらいは左右等確率で一歩進むとしましょう。この酔歩のプログラムを作成しなさい。原点から出発して数十ステップ実行し、得られた移動の経路を図示しなさい（いくつか異なる乱数列で試すこと）。また、ずっと長いステップ実行すると移動経路はどうなりますか？計算結果をもとに考察しなさい。

課題 9　たくさんの酔っぱらいからなる集団を考えましょう。酔っぱらいは、各人勝手に格子点間をふらふらしています。時刻 0 にすべての酔っぱらいは原点に集合しており、その後移動を始めます。時刻 n での酔っぱらいの位置の分布を図示しなさい（いくつかの n に対して試すこと）。また長時間経過後には、酔っぱらいの分布は正規分布でよく記述されることを確率密度関数を重ねて図示することにより、示しなさい[*11]。

課題 10（発展）　酔っぱらいが時刻 0 からある時刻までの間に原点の右側（正の領域）にいる割合を a としましょう[*12]。この割合 a に対する分布 $f(a)$ を求め、その形状が意味することを議論しなさい（計算する前に予想がつきますか？）。なお長時間経過後には、

$$f(a) = \frac{1}{\pi\sqrt{a(1-a)}}$$

に近づいていくことが知られています（逆正弦法則[*13]）。

5.1　酔っぱらいを生成する

前節の乱数を利用して、酔歩のプログラムを作成しましょう。簡単のため、まず 1 次元格子点上の酔歩を取り上げます。数直線 x 上を原点から出発し、単位時間ごとに 1 コマずつ右か左に移動する酔っぱらいがいます。この人は完全な酩酊状態にあるので、左右どちらに移動するかは各時刻ごとに確率 $1/2$ でランダムに決まるとします。この設定のも

[*11] 確率密度関数として表す場合は、ヒストグラムを幅 Δx で割ることを忘れずに。

[*12] n ステップ中，正の領域にいたステップ数を n_+ とすると，$a = n_+/n$。

[*13] 逆正弦法則については次の文献を参照のこと：W. フェラー「確率論とその応用 I 上」（紀伊国屋書店）第 III 章.

とで、時々刻々酔っぱらいの移動経路を追跡していきます。この酔歩 (random walk) は、ブラウン運動の確率過程モデルとなっています。

確率 p で何らかの処理をしたいときは、区間 $[0,1)$ での一様乱数を使うと簡単に実装することができます。この乱数が区間 $[0,p)$ にある確率 Prob は、

$$\mathrm{Prob}[0 \le x < p] = \int_0^p dx = p$$

です。つまり p より小さい一様乱数は、確率 p で生成されることになります。この確率で処理するプログラムは以下のようになるでしょう。

```
rn = rand()/(RAND_MAX+1.0);
if (rn < p) {
    やりたい処理;
}
```

酔歩のプログラム例を付録 G にあげます。例は NSTEP ステップ（時刻 NSTEP）までのシミュレーションを実行します。

このプログラムを 3 回実行し、その移動の様子を示したものが図 5.1 にあります。プログラム例では、乱数の種をカレンダー時間から取ってきているので、実行するたびに移動の様子は異なることに注意してください。

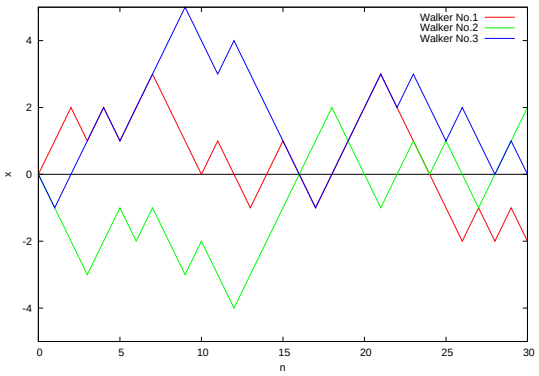

図 5.1　一次元格子点上の酔歩。縦軸は格子点上の位置 x、横軸は時間（ステップ数）n を表す。異なる乱数列を用いて 3 回酔歩のシミュレーションを実行し、その移動の様子が重ね書きしてある。

5.2　束にして考える

前小節の例のように、酔っぱらいの移動経路には様々な可能性があります。この酔っぱらいが従っている確率分布を記述するために、たくさんの可能な移動経路を束にして考えましょう。このために（迷惑きわまりないですが）酔っぱらいを大量に生成して、その集団での位置の分布が時間とともにどのように変化するのか見てみることにします。

付録 G のプログラム例では酔っぱらいを 1 万人 (N_WALKER = 10000) 生成して、その全体を 1 ステップづつ進めています。酔っぱらいの間にはやりとり（相関）はなく、各人がそれぞれ勝手にふらふらしていることに注意してください。図 5.2 - 5.7 にいくつかの計算例が示してあります。

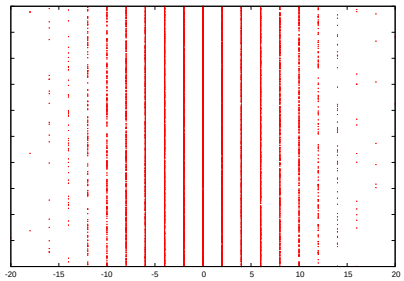

図 5.2　酔っぱらいの分布 ($n = 30$ ステップ)。横軸は位置 x。酔っぱらいごとに縦軸方向にずらして出力。

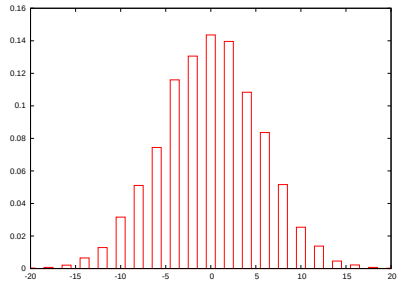

図 5.3　酔っぱらいの居場所のヒストグラム ($n = 30$ ステップ)。横軸は酔っぱらいの位置 x。

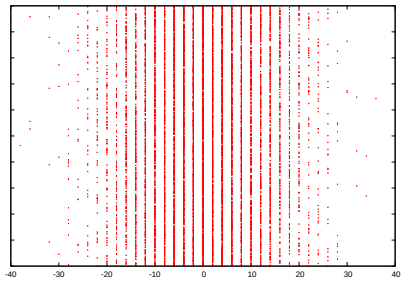

図 5.4　酔っぱらいの分布 ($n = 100$ ステップ)。横軸は位置 x。酔っぱらいごとに縦軸方向にずらして出力。

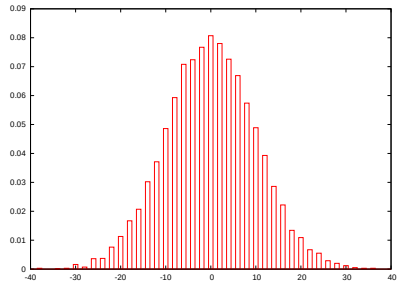

図 5.5　酔っぱらいの居場所のヒストグラム ($n = 100$ ステップ)。横軸は酔っぱらいの位置 x。

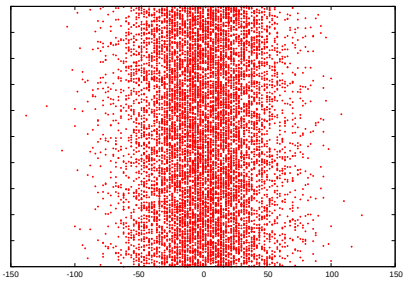

図 5.6　酔っぱらいの分布 ($n = 1000$ ステップ)。横軸は位置 x。酔っぱらいごとに縦軸方向にずらして出力。

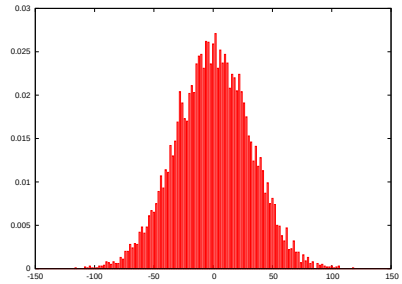

図 5.7　酔っぱらいの居場所のヒストグラム ($n = 1000$ ステップ)。横軸は酔っぱらいの位置 x。

6　平面上の酔歩

課題 11　平面上での酔歩を考えましょう。各ステップでの移動距離は定数 1 に固定するのではなく、正規分布に従っています。この時、時刻 n での酔っぱらいの位置を $\boldsymbol{r}(n) = (x_n, y_n)$ とすると、その移動は以下の規則で与えられます。

$$\begin{aligned} x_n &= x_{n-1} + f^x_{n-1} \\ y_n &= y_{n-1} + f^y_{n-1} \end{aligned} \qquad (n = 1, 2, \ldots)$$

ここで f^x_{n-1}, f^y_{n-1} は、

$$\langle f^\alpha_i \rangle = 0, \qquad \langle f^\alpha_i f^\beta_j \rangle = \Delta^2 \delta_{\alpha\beta}\delta_{ij} \qquad (\alpha, \beta = x, y)$$

を満たす正規乱数（分散が Δ^2）です。この酔歩のシミュレーションのプログラムを作成し、いくつかその移動経路を図示しなさい。またその経路の特徴を議論しなさい。

課題 12（発展）　酔っぱらいの集団を生成し、経路を束にして考えることにより確率分布を表しましょう。移動の様子を特徴づけるために平均二乗変位 $R^2(t)$ という量を導入します[*14]。これは各酔っぱらいが時刻 0 から時刻 n までに移動した変位の二乗を、多数生成した酔っぱらいにわたって平均したものであり、以下のように定義されます。

$$\begin{aligned} R^2(n) &= \langle |\boldsymbol{r}_1(n) - \boldsymbol{r}_1(0)|^2 \rangle \\ &= \frac{1}{N_w} \sum_{i=1}^{N_w} |\boldsymbol{r}_i(n) - \boldsymbol{r}_i(0)|^2. \end{aligned}$$

ここで $\boldsymbol{r}_i(n)$ は、時刻 n での i 番目の酔っぱらいの位置であり、N_w は酔っぱらいの総数です。平均二乗変位 $R^2(n)$ を計算し、時間の関数としてプロットしなさい。また中心極限定理から平均二乗変位の n 依存性を求めなさい。ここで変位は以下のように書けることに注意すること[*15]。

$$\begin{aligned} \boldsymbol{r}_1(n) - \boldsymbol{r}_1(0) &= \sum_{m=1}^{n} \Delta\boldsymbol{r}_1(m) \\ \Delta\boldsymbol{r}_1(m) &= \boldsymbol{r}_1(m) - \boldsymbol{r}_1(m-1). \end{aligned}$$

[*14] これは酔っぱらいの位置を確率変数とした場合の分散にあたります。

[*15] 各ステップでの変位 $\Delta\boldsymbol{r}$ は正規分布に従っているので、任意の n に対して中心極限定理が成り立ちます。

6.1　平面上に酔っぱらいを生成する

酔っぱらいの動く範囲を直線上から平面上に広げて考えましょう。また 1 ステップでの移動距離は、正規分布に従っています。付録 G にプログラム例を示します。なお、ここでは分散は $\Delta^2 = 1$ としています。

図 6.1 に、このプログラムによって計算された移動経路が 3 回分重ねて示してあります。

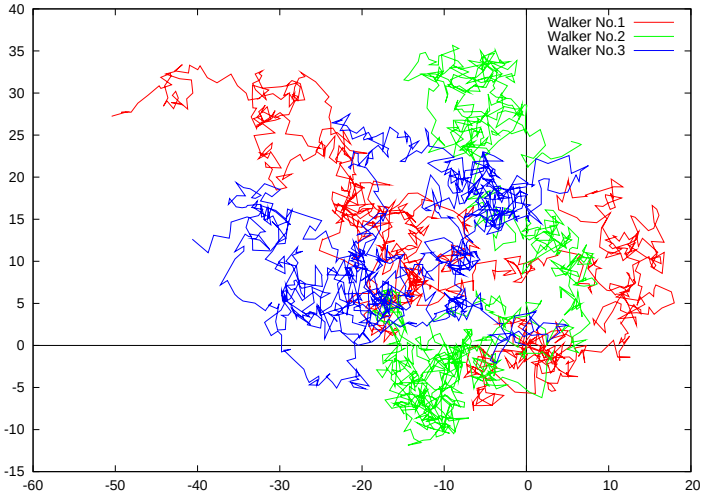

図 6.1　平面上の酔歩。3 つの異なる乱数列により生成された経路が重ねて図示してある。

7　エーレンフェストの壷

N 個のコインがある。それぞれのコインには 0 から $N-1$ までの数字が刻印されており、その数字に重なりはない。ここで 2 つの壷 A, B を用意する。最初にすべてのコインは壷 A に入っている。壷とコインを併せて系と呼ぶことしよう。系の状態は、壷に入っているコインの数で指定される。次のルールに従って、系の状態を更新することを考えよう。

1) N 個のコインから、ひとつランダムに選ぶ。
2) そのコインが入っている壷を確認する。
3) そのコインをもう一方の壷に移す。

上記のルールを繰り返し適用することにより、系の状態を時間発展させる。

課題 13（発展）　上述のルールにしたがってコイン-壷系を時間発展させるプログラムを作成し、壷 A に入っているコインの個数を時間の関数として図示しなさい。コインの数は、$N = 10, 100, 1000, 10000$ を試してみること。また、長時間後に壷 A に入っているコインの個数について考察しなさい。

参考文献

[1] 和達三樹、十河清「キーポイント　確率・統計」岩波書店.
講義で利用した教科書です。実習資料作成の際にも参考にしました。

[2] ファインマン，レントン，サンズ「ファインマン物理学 I 力学」岩波書店.
6 章は、確率や酔歩の話題が取り扱われています。

[3] 米沢富美子「ブラウン運動」共立出版.
プラウン運動に関する楽しいモノグラフなので、一度読んでみてください。資料作成の際にも参考にしました。

[4] 寺本英「ランダムな現象の数学」吉岡書店.
酔歩に関連した問題の詳しい解析があります。

[5] 大沢文夫「大沢流　手づくり統計力学」名古屋大学出版会.

[6] W. H. Press 他「Numerical Recipes in C［日本語版］」技術評論社.
タイトルどおり数値計算関係のレシピ集で、すぐに使えるプログラムが載っているので便利です。理論的な解説もコンパクトにまとめられています。ただし実際に使う場合は浮動小数点型変数が単精度 (`float`) で実装されている部分を倍精度 (`double`) に変更したほうが良いでしょう。

[7] J. M. ティッセン「計算物理学」シュプリンガー・フェアラーク東京.
計算物理学に関する公汎な問題が取り上げられています。プログラム例もウェブ上で公開されています（フォートランで実装されているようですが)。

[8] 標準ライブラリで提供されている `rand()` は、実はあまり性能のよい乱数生成法を使っていません。簡単な用途には問題ありませんが、乱数を大量に生成し、かつその品質が問題になる場合はおすすめできません（参考文献 [4] に合同式法の問題点がまとめられています)。例えば、以下のメルセンヌ・ツイスタという方法は、最近よく使われています。

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/mt.html

このウェブページでは、C のプログラムも公開されています。

付録 A　プログラミングの注意　初歩の初歩

1) プログラムはきれいに書く
   汚いプログラムはバグの温床です。

2) 字下げは系統的に行う
   プログラムの論理構造とインデントの対応を正しくつけましょう。

3)「慣用句」を覚える
   決まりきった書き方を身につけましょう。

4) デバッグは系統的に行う
   ここまでは正しいという地点を少しずつ進めていきましょう。

5) 自分のバクの傾向を知る
   同じ間違いを繰り返すものです。

6) 力任せなデバッグ
   とにかく何でも書き出してみましょう。予想外の数値が出てきたりします。

7) 告白によるデバッグ
   他人に自分のプログラムを説明してみましょう。意外と色々気が付くものです。

8) 気の利いたデバッグ
   デバッガという便利なツールもあります。例えば、`gdb` (`ddd`) など。

付録 B　コンンパイラのオプション

1) test1.c という C のプログラムを作成したとする。gcc でコンパイルしたい場合、以下のようにする：

```
% gcc -Wall -o test1 test1.c
```

それぞれのオプションの意味は次の通りである。

- -Wall　必須ではないがコンパイラが警告 (Warning) をすべて出力してくれるので、このオプションはつけるように習慣づける。仮にコンパイルが成功しても、警告が出ている場合はおかしなことをしている場合が多い。

- -o　実行ファイル名を自分で指定する。上記の例では、コンパイルが成功すると test1 という名前の実行ファイルができる（実行プログラム名は、ソースプログラム名にあわせる必要はない)。-o test1 を省略するとデフォルトのファイル名 a.out となる。

2) test2.c という C のプログラムを作成した。このプログラムでは数学ライブラリを使用している（プログラムの先頭で「#include <math.h>」と書いてある場合)。gcc でコンパイルしたい場合、 以下のようにする：

```
% gcc -Wall -lm -o test2 test2.c
```

- -lm　数学ライブラリをリンクする。

付録 C　標準入力と標準出力

scanf() や printf() は標準入力、標準出力を制御する関数である。デフォルトでは、標準入出力はターミナルに接続しているが、入出力先をファイルにしたい場合は以下のようにする（リダイレクトという)。ここで実行ファイル名を test とし、ファイル input には入力データが記述されているとする。

C.1　UNIX/LINUX 環境の場合

1) 標準入力　ターミナルから手で入力していたデータが、ファイル input にそのまま書いてある

```
% ./test < input
```

2) 標準出力　ターミナルに出力されていたデータをそのままファイル output に書き出す

```
% ./test > output
```

3) 標準入出力を共にファイルから行う場合

```
% ./test < input > output
```

百聞は一見に如かず。簡単な例を自分で作って試してみましょう。

C.2　Borland 統合環境の場合

「ビルド」から「引数を指定して実行」を選択すると，小窓が現れるので，そこに引数を書き込む。

付録 D　Gnuplot で作成した図のファイルへの出力

ここでは gnuplot を使って作成した図をファイルへ出力する方法について簡単に説明します。まず gnuplot を起動し、何らかの図を作成したとします。以下のようすると、この図をポストスクリプトのファイルにすることができます。

```
gnuplot> set terminal postscript
gnuplot> set output "graph1.eps"
gnuplot> replot
```

ここでファイル名は、"graph1.eps"としています。勿論、自分の好きなファイル名にすることができます。できあがったファイルは、"GSview"などで確認することができます。

棒グラフでプロットしたい場合は（ファイル名"hist.dat"）

```
gnuplot> plot "hist.dat" with boxes
```

とすればよい。

付録 E　独立な確率変数の和と積の期待値

二つの独立な連続的確率変数 x と y を考えよう[*16]。各々、確率分布 $f_X(x)$ と $f_Y(y)$ に従うとする。同時確率密度 $f_{XY}(x,y)$ は x と y が独立であるために、$f_{XY}(x,y) = f_X(x)f_Y(y)$ と積の形でかける。まず確率変数の和の期待値を考えよう。

$$\begin{aligned}\langle x+y\rangle_{XY} &= \int dx \int dy\ (x+y) f_{XY}(x,y) \\ &= \int dx\ x f_X(x) \underbrace{\int dy f_Y(y)}_{=1} + \underbrace{\int dx f_X(x)}_{=1} \int dy\ y f_Y(y) \\ &= \langle x\rangle_X + \langle y\rangle_Y .\end{aligned}$$

独立な確率変数の和の期待値は、各々の分布で計算した期待値の和で書ける。同様に x と y 各々の任意の関数 $F(x)$ と $G(y)$ の和の期待値は、各々の期待値の和で書くことができる。

$$\langle F(x)+G(y)\rangle_{XY} = \langle F(x)\rangle_X + \langle G(y)\rangle_Y .$$

一方、積の期待値は

$$\begin{aligned}\langle xy\rangle_{XY} &= \int dx \int dy\ xy f_{XY}(x,y) \\ &= \int dx\ x f_X(x) \int dy\ y f_Y(y) \\ &= \langle x\rangle_X \langle y\rangle_Y\end{aligned}$$

となる。つまり独立な確率変数の積の期待値は、各々が従う分布による期待値の積になる。同様に

$$\langle F(x)G(y)\rangle_{XY} = \langle F(x)\rangle_X \langle G(y)\rangle_Y$$

となる。

上述の関係式を用いると、確率変数の和で定義された新たな確率変数の平均と分散を容易に計算することができる。それぞれの分布での平均と分散を

$$\begin{aligned}&\langle x\rangle_X = \mu_X && \langle y\rangle_Y = \mu_Y \\ &\langle (x-\mu_X)^2\rangle_X = \sigma_X^2 && \langle (y-\mu_Y)^2\rangle_Y = \sigma_Y^2\end{aligned}$$

[*16] 以下の議論は離散的確率変数の場合でも同様です。各自確認すること。

とすると、平均は直ちに

$$\langle x+y\rangle_{XY} = \mu_X + \mu_Y$$

となる。一方、分散は

$$\begin{aligned}
&\left\langle \{(x+y)-(\mu_X+\mu_Y)\}^2 \right\rangle_{XY} \\
&= \left\langle \{(x-\mu_X)+(y-\mu_Y)\}^2 \right\rangle_{XY} \\
&= \left\langle (x-\mu_X)^2 \right\rangle_X + \left\langle (y-\mu_Y)^2 \right\rangle_Y + 2\underbrace{\langle (x-\mu_X)\rangle_X}_{=0}\underbrace{\langle (y-\mu_Y)\rangle_Y}_{=0} \\
&= \sigma_X^2 + \sigma_Y^2
\end{aligned}$$

である。

上述の議論は和をとる変数の数が増えても同様に成り立つ。n 個の独立な確率変数を用意する: $x_i,\ i=1,\ldots,n$。ここで各々の平均と分散を $\mu_{X_i}, \sigma_{X_i}^2$ と表記しよう。その和で確率変数を定義すると $(\sum_{i=1}^n x_i)$、平均は

$$\left\langle \sum_{i=1}^n x_i \right\rangle = \sum_{i=1}^n \mu_{X_i}$$

となる。同様に分散は、

$$\left\langle \left(\sum_{i=1}^n x_i - \sum_{i=1}^n \mu_{X_i} \right)^2 \right\rangle = \sum_{i=1}^n \sigma_{X_i}^2$$

となる。なお、確率変数が独立でない場合はこの付録の公式をそのまま用いることはできないので注意を要する。

付録 F　中心極限定理

定理　確率変数 $x_1, \ldots, x_n$ が互いに独立に同一の確率分布に従い、その平均・分散は

$$\langle x_1 \rangle = \cdots = \langle x_n \rangle = \mu$$
$$\langle (x_1 - \mu)^2 \rangle = \cdots = \langle (x_n - \mu)^2 \rangle = \sigma^2$$

である。このとき、

$$\bar{x} = \frac{1}{n}(x_1 + \ldots + x_n)$$

に対して

$$\bar{z} = \frac{\bar{x} - \mu}{\sigma/\sqrt{n}}$$

とすると、n を大きくしたとき、$\bar{z}$ の分布は標準正規分布 $N(0,1)$ に近づく。

証明　最初に基礎的な事柄を整理しておこう。平均 μ, 分散 σ^2 の確率変数 y が従う確率密度を $f(y)$ とする。この時、特性関数 $\hat{f}(\xi)$ は

$$\hat{f}(\xi) = \int_{-\infty}^{+\infty} dy e^{i\xi y} f(y) = \langle e^{i\xi y} \rangle$$

と定義される。この特性関数の対数を $i\xi$ に関してべき級数展開したものをキュミュラント展開という:

$$\hat{f}(\xi) = e^{\sum_{\ell=1}^{\infty} \kappa_\ell \frac{(i\xi)^\ell}{\ell!}}.$$

ここで κ_ℓ は、ℓ 次のキュミュラントである。最初の二つを書くと

$$\kappa_1 = \langle y \rangle = \mu, \qquad \kappa_2 = \Big\langle \big(y - \langle y \rangle\big)^2 \Big\rangle = \sigma^2$$

となる。特に y が正規分布に従う場合、

$$\hat{f}(\xi) = \langle e^{i\xi y} \rangle = e^{i\mu\xi - \frac{1}{2}\sigma^2\xi^2}$$

となり、3 次以上のキュミュラントはすべて 0 になるという著しい特徴がある。

さて、証明に移ろう。まず変数 z_i を以下のように定義する。

$$z_i = \frac{x_i - \mu}{\sigma}, \qquad i = 1, \ldots, n.$$

この確率変数の平均と分散は

$$\langle z_i \rangle = \frac{1}{\sigma}(\langle x_i \rangle - \mu) = 0$$
$$\left\langle (z_i - \langle z_i \rangle)^2 \right\rangle = \left\langle z_i^2 \right\rangle = \frac{1}{\sigma^2} \left\langle (x_i - \mu)^2 \right\rangle = \frac{1}{\sigma^2}\sigma^2 = 1$$

である。この変数を用いると $\bar{z}$ は、

$$\bar{z} = \frac{1}{\sqrt{n}} \sum_{i=1}^{n} z_i$$

と表すことができる。変数 $\bar{z}$ に対する特性関数は、

$$\begin{aligned} \hat{f}(\xi) &= \langle e^{i\xi\bar{z}} \rangle \\ &= \langle e^{i\xi \frac{z_1}{\sqrt{n}}} \cdots e^{i\xi \frac{z_n}{\sqrt{n}}} \rangle \end{aligned}$$

となる。付録 E で示したように、独立な確率変数の積の期待値は、期待値の積となるので、

$$\begin{aligned} \hat{f}(\xi) &= \langle e^{i\xi \frac{z_1}{\sqrt{n}}} \rangle \cdots \langle e^{i\xi \frac{z_n}{\sqrt{n}}} \rangle \\ &= \hat{f}_1(\xi) \cdots \hat{f}_n(\xi). \end{aligned}$$

ここで $\hat{f}_i(\xi)$ は、z_i に対する特性関数である。この特性関数をキュミュラント展開しよう:

$$\begin{aligned} \hat{f}_i(\xi) &= \langle e^{i\xi \frac{z_i}{\sqrt{n}}} \rangle \\ &= e^{\kappa_1 \frac{i\xi}{\sqrt{n}} + \kappa_2 \frac{1}{2!} \left(\frac{i\xi}{\sqrt{n}}\right)^2 + \kappa_3 \frac{1}{3!} \left(\frac{i\xi}{\sqrt{n}}\right)^3 + \cdots}. \end{aligned}$$

2 次までのキュミュラントは、$\kappa_1 = \langle z_i \rangle = 0, \kappa_2 = \left\langle (z_i - \langle z_i \rangle)^2 \right\rangle = 1$ であるので

$$\hat{f}_i(\xi) = e^{\frac{1}{2}\left(\frac{i\xi}{\sqrt{n}}\right)^2 + \frac{\kappa_3}{6}\left(\frac{i\xi}{\sqrt{n}}\right)^3 + \cdots}$$

と書くことができる。これを $\bar{z}$ に対する特性関数に代入すると、

$$\begin{aligned} \hat{f}(\xi) &= e^{-\frac{1}{2}\xi^2 + \frac{\kappa_3}{6\sqrt{n}}(i\xi)^3 + \cdots} \\ &\simeq e^{-\frac{1}{2}\xi^2} \end{aligned}$$

となり、n が大きくなると $i\xi$ の 3 次以上の項は無視できる。これは標準正規分布 $N(0,1)$ の特性関数であり、n が大きくなれば確率密度関数は

$$f(\bar{z}) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\bar{z}^2}$$

となることが示された。

確率変数 $\bar{x}$ に対しては

$$f(\bar{z})d\bar{z} = \frac{1}{\sqrt{2\pi\sigma^2/n}} e^{-\frac{(\bar{x}-\mu)^2}{2\sigma^2/n}} d\bar{x} = f(\bar{x})d\bar{x}$$

となり、正規分布 $N(\mu, \sigma^2/n)$ に従うことがわかる。これは、平均は個別の確率変数のものと同じで、標準偏差が $1/\sqrt{n}$ で小さくなることを意味している。また、導出過程から明らかなように確率変数 x_i $(i = 1, \ldots, n)$ が正規分布に従っているときは、中心極限定理は任意の n に対して成り立つ。さらに

$$x(n) = x_1 + \cdots + x_n$$

とした場合は、変数 $x(n)$ は正規分布 $N(n\mu, n\sigma^2)$ に従うことを確認できるだろう。

付録 G　プログラム例

区間 [0, 1) での一様乱数

```
#include <stdio.h>
#include <stdlib.h>

#define N_MAX 10000
#define NGRID 20

int main(void)
{
  int ibin, i;
  unsigned int seed = 1121; // 乱数の種 (seed)
  double uniform_rn, dx, xgrid;
  double hist[NGRID];

  srand(seed); // 乱数列の種を設定．プログラムの最初に一度実行．

  for (i = 0; i < NGRID; i++) hist[i] = 0.0; // 配列の初期化

  dx = 1.0/NGRID; // 小区間の幅．全区間の幅は 1.0

  for (i = 0; i < N_MAX; i++) {
    uniform_rn = rand()/(RAND_MAX+1.0);
    ibin = (int) (uniform_rn/dx);  // (int) は，整数への型変換
    hist[ibin] += 1.0;
  }

  for (i = 0; i < NGRID; i++)
     hist[i] = hist[i]/N_MAX/dx; // 最後に規格化

  // 出力する．x の値は小区間の中央値としています．
  for (i = 0; i < NGRID; i++) {
    xgrid = i*dx + 0.5*dx;
    printf("%f  %f\n", xgrid, hist[i]);
  }

  return 0;
}
```

正規乱数（中心極限定理を用いた簡便法）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N_MAX 10000
#define NGRID 20
#define N_RN 12

int main(void)
{
  int ibin, i, j;
  double y, hist[NGRID];
  double xgrid, dx;
  double width = 6.0; // 全区間の幅. [-0.5*width, 0.5*width)

  // 乱数列の種を設定. プログラムの最初に一度実行.
  srand((unsigned int) time(NULL));

  // 配列の初期化
  for (i = 0; i < NGRID; i++)
    hist[i] = 0.0;

  // 小区間の幅
  dx = width/NGRID;

  for (i = 0; i < N_MAX; i++) {

    // 正規乱数を一つ生成
    y = 0.0;
    for(j = 0; j < N_RN; j++)
      y += rand()/(RAND_MAX+1.0);
    y -= 6.0;

    // 自分で設定した全区間の幅 width 内にある乱数のみカウント
    // 配列のサイズを越えないようにするために想定した範囲外に
    // 生成された乱数は除外します.
    // 区間 [-0.5*width, 0.5*width) に注意.
    if (y >= -0.5*width && y < 0.5*width) {
      ibin = (int) ((y + 0.5*width)/dx);
      hist[ibin] += 1.0;
    }

  }
```

```
  // 規格化して確率密度関数とする
  for (i = 0; i < NGRID; i++)
    hist[i] = hist[i]/N_MAX/dx;

  // 確率密度関数を出力
  // x の値は，小区間の中央値としています．
  for (i = 0; i < NGRID; i++) {
    xgrid = -0.5*width + i*dx + 0.5*dx;
    printf("%f  %f\n", xgrid, hist[i]);
  }

  return 0;
}
```

正規乱数（ボックス・ミュラー法）

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N_MAX 10000
#define NGRID 20

// ボックス・ミュラー法により正規乱数を生成する関数
double gasdev(void)
{
  double x1, x2, y1, y2;

  x1 = rand()/(RAND_MAX+1.0);
  x2 = rand()/(RAND_MAX+1.0);
  y1 = sqrt(-2.0*log(x1))*cos(2.0*M_PI*x2);
  y2 = sqrt(-2.0*log(x1))*sin(2.0*M_PI*x2);

  return y1;
}

int main(void)
{
  int ibin, i;
  double y, hist[NGRID];
  double xgrid, dx, width = 6.0;

  // 乱数列の種を設定．プログラムの最初に一度実行．
  srand((unsigned int) time(NULL));

  // 配列の初期化
  for (i = 0; i < NGRID; i++)
    hist[i] = 0.0;

  // 小区間の幅
  dx = width/NGRID;

  for (i = 0; i < N_MAX; i++) {

    // 正規乱数を一つ生成
    y = gasdev();

    // 自分で設定した全区間の幅 width 内にある乱数のみカウント
    // 配列のサイズを越えないようにするために想定した範囲外に
```

```
    // 生成された乱数は除外します.
    // 区間 [-0.5*width, 0.5*width) に注意.
    if (y >= -0.5*width && y < 0.5*width) {
      ibin = (int) ((y + 0.5*width)/dx);
      hist[ibin] += 1.0;
    }

  }

 // 規格化して確率密度関数とする
 for (i = 0; i < NGRID; i++)
   hist[i] = hist[i]/N_MAX/dx;

  // 確率密度関数を出力
  // x の値は, 小区間の中央値としています.
  for (i = 0; i < NGRID; i++) {
    xgrid = -0.5*width + i*dx + 0.5*dx;
    printf("%f  %f\n", xgrid, hist[i]);
  }

  return 0;
}
```

一次元格子点上の酔歩（酔っぱらい一人版）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NSTEP  30

int main(void)
{
  int istep = 0,  x = 0;
  double rn;

  // 乱数列の種を設定．プログラムの最初に一度実行．
  srand((unsigned int) time(NULL));

  printf("%d %d\n", istep, x); // 最初の位置を出力

  for (istep = 1; istep <= NSTEP; istep++) {

    // 区間 [0, 1) での一様乱数をひとつ生成
    rn = rand()/(RAND_MAX+1.0);

    // 確率 0.5 で右 (x++), 確率 0.5 で左 (x--) へ 1 コマ移動
    if (rn < 0.5) {
      x++;
    } else {
      x--;
    }

    printf("%d %d\n", istep, x); // 各ステップでの位置を出力
  }

  return 0;
}
```

一次元格子点上の酔歩（酔っぱらい多人数版）

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define NSTEP 30
#define N_WALKER  10000
#define NGRID 60  // NSTEP x 2

int main(void)
{
  int i, ibin, istep = 0;
  int x[N_WALKER];
  double rn;
  double hist[NGRID+1]; // 配列のサイズ NGRID+1 に注意

   // 乱数列の種を設定．プログラムの最初に一度実行．
   srand((unsigned int) time(NULL));

  for (i = 0; i < NGRID+1; i++)
    hist[i] = 0.0;

  // すべての酔っぱらいは最初原点にいる
  for (i = 0; i < N_WALKER; i++)
    x[i] = 0;

  for (istep = 1; istep <= NSTEP; istep++) {

    // 酔っぱらいひとりずつ 1 コマ進める
    for (i = 0; i < N_WALKER; i++) {

      // 区間 [0, 1) での一様乱数をひとつ生成
      rn = rand()/(RAND_MAX+1.0);

      // 確率 0.5 で右 (x++), 確率 0.5 で左 (x--) へ 1 コマ移動
      if (rn < 0.5) {
        x[i]++;
      } else {
        x[i]--;
      }

    }

  }
```

```
  // NSTEP 終了後の酔っぱらいの位置の分布を調べる.
  // 自分で設定した区間 [-NGRID/2, NGRID/2] 内にある場合のみ
  // カウントする.
  for (i = 0; i < N_WALKER; i++) {
    ibin = x[i] + NGRID/2;
    if (ibin >= 0 && ibin < NGRID+1) {
      hist[ibin] += 1.0;
    }
  }

  // 分布を出力. 位置 x は離散値（整数）であることに注意
  for (i = 0; i < NGRID+1; i++)
    printf("%d %f\n", i-NGRID/2, hist[i]/N_WALKER);

  return 0;
}
```

平面上の酔歩

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define NSTEP 1000

// ボックス・ミュラー法により正規乱数を生成する関数
double gasdev(void)
{
  double x1, x2, y1, y2;

  x1 = rand()/(RAND_MAX+1.0);
  x2 = rand()/(RAND_MAX+1.0);
  y1 = sqrt(-2.0*log(x1))*cos(2.0*M_PI*x2);
  y2 = sqrt(-2.0*log(x1))*sin(2.0*M_PI*x2);

  return y1;
}

int main(void)
{
  int istep = 0;
  double x = 0.0, y = 0.0;
  double grn1, grn2;

  srand((unsigned int) time(NULL));

  printf("%d  %f  %f\n", istep, x, y);

  for(istep = 1; istep <= NSTEP; istep++){
    grn1 = gasdev();  grn2 = gasdev();
    x += grn1;  y += grn2;
    printf("%d  %f  %f\n", istep, x, y);
  }

  return 0;
}
```