50　乱数

乱数とはさいころの目のようにでたらめ[*20]に表れる数字のことです。乱数は，乱雑[*21]な現象を記述する場合だけでなく，決定的な量の近似値を計算するためにもしばしば用いられるます。ここでは Sage における乱数の基本的な使い方と簡単な応用を紹介します。以下のプログラムは Sage notebook で動作する事を確認していますが，`plot` 等の Sage の命令を除けば，純粋な Python プログラムとしても動作します。）

Python で使用される乱数はメルセンヌ・ツイスター (Mersenne twister) と呼ばれるアルゴリズムによって生成されます。これはある決まった手続きによって数を生成してゆくものですが，これは実用上十分なランダム性をもち非常に高速に生成する事ができるアルゴリズムです。ちなみにメルセンヌ・ツイスターは 1996 年頃日本人（松本眞氏・西村拓士氏）によって開発されました。これは巨大なメルセンヌ素数 $2^{19937}-1$ が素数であることを利用して作られるます。

コンピューターで作る乱数は，ある決定的な手続きで生成されます。したがって，それ自体にはパターンがあるので本当の意味での乱数ではありません。このようなものを擬似乱数といいます。どのような数列なら真の乱数といえるのかというのは難しい問題[*22]です。ひとつの擬似乱数は，ある種の問題を考えるときには，十分乱雑だけど，他の問題を考える場合には，乱雑さが不十分かもしれません。ここではどのようにして良い乱数を生成するのかという問題には触れずに，乱数の使い方と，その簡単な応用例を紹介したいと思います。

50.1　乱数の基本的な使い方

Python には `random` という標準ライブラリがあり，様々なタイプの乱数を生成することができます。

まずは次のプログラムを実行してみましょう：

```
1 import random     # ライブラリ random を呼び出す
2 random.seed(0)    # 乱数種を 0 とする
3 print "[0,1) の中の実数をランダムに生成する"
4 for i in range(5):
5       print random.random()      #[0,1) の実数をランダムにプリントする
```

実行結果の例

```
[0,1) の中の実数をランダムに生成する
0.844421851525
0.75795440294
0.420571580831
0.258916750293
0.511274721369
```

`random.seed(0)` は乱数の種の設定ですが，これについては後で説明します。`random.random()` で区間 $[0,1)$ の実数をランダムに作ることができます。[*23]

さて，上のプログラムをもう一度実行して見ましょう。実行結果は先ほどと全く同じになります。また隣の席の人とも同じになっているはずです。この数列は一見すると乱雑ですが，ある決定的な手続きで作られています。このようにある種の数学的な手続きで作られる数列だけど，一見するとパターンがなく乱数のように見えるものを疑似乱数といいます。

上の乱数には再現性がありますが，数値計算では乱数の再現性がない事にはあまり意味はなく，むしろ再現性のある方が有益な場合が多くあります。たとえば，ある数値計算をしていて予想外の結果が出たときに，それが偶然なのか必然なのかの原因を知りたいと思っても，同じ状況を再現できなければ，原因を突き止める事

[*20] 出鱈目
[*21] 乱雑の英語約は random（ランダム）だけど，二文字まで同じなのが不思議
[*22] 数学的なというより思想上の問題
[*23] $[0.0, 1.0)$ の中にある浮動小数点数が無作為に抽出されます。

が難しくなるでしょう[*24]。

乱数の列は乱数種 (random seed) ごとに決まっており，乱数種を変えることにより異なる乱数を使うことができます。例えば，上のプログラムで random.seed(1) のように乱数種 0 代わりに 1 を使えば，異なる乱数が得られます。乱数種を変えてみましょう。

```
1 import random
2 random.seed(1)
3 for i in range(5):
4     print random.random()
```

実行結果の例

```
0.134364244112
0.847433736937
0.763774618977
0.255069025739
0.495435087092
```

パスワードの生成などの時には，いつ実行しても同じ結果が出てしまうのは困るかもしれません。そういう場合には，そのときの時刻や乱雑なキー入力，マウスの動きから乱数を作ればよいでしょう。放射性物質の崩壊時間を利用すれば，真の乱数を作ることが出来ます[*25]。

プログラムを実行するたびに異なる乱数を使いたければ乱数種の指定を random.seed() とします。この場合，プログラムの実行時刻から乱数種が決まります：

```
1 import random
2 random.seed()   # 乱数種を時間から定める
3 print "[0,1)の中の実数をランダムに生成する"
4 for i in range(4):
5     print random.random()
```

実行結果の例

```
[0,1) の中の実数をランダムに生成する
0.0104079725999
0.605201486317
0.0631615144758
0.592872489434
```

このプログラムは毎回異なる乱数を生成します。このプログラムを何度か実行して結果が異なることを確認してみましょう。このプログラムの実行する時刻が全く同じでない限り，同じ結果は得られません。

$[0,1)$ の一様分布だけでなく，さまざまな乱数が用意されています。代表的なものを紹介します。

乱数の種類

- random() ： $[0,1)$ の中の実数をランダムに返す。
- randint(a,b) ：$a \leq N \leq b$ であるようなランダムな整数 N を返す
- choice(list) ： list の中からランダムに要素を取り出す。
- uniform(a,b) ：$[a,b]$ の中の実数をランダムに返す。
- gauss(mu,sigma) ：平均 mu，標準偏差 sigma のガウス分布に従うランダムな実数を返します。

乱数生成の命令はすべて random. で始まり，乱数の種類に応じてその後の命令を指定します。

たとえば random.choice(....) はライブラリの中の choice という命令を呼び出すことを意味します。たとえばリスト ['a','b','c'] からランダムに要素を取り出すには random.choice(['a','b','c']) のようにします。上記以外にも，多くの種類の乱数が用意されていますが，必要に応じて調べて下さい。

次のプログラムは 1 から 8 までの整数 1,2,3,4,5,6,7,8 をランダムに 20 個表示します：

[*24] 幽霊の目撃情報のように

[*25] 量子力学によれば，放射性物質の崩壊時間は真に確率的であり，それを正確に予測することは理論上不可能です

```
import random
random.seed(0)
for i in range(10):     # 1から8までの整数をランダムに抽出する
    print random.randint(1,8),
```

実行結果の例

```
7 7 4 3 5 4 7 3 4 5 8 5 3 7 5 3 8 8 7 8
```

次のプログラムではリスト x=["red", "green", "blue", "black", "white", "yellow"] の中から要素がランダムに抽出されます：

```
x = ["red", "green", "blue", "black", "white", "yellow"]
for i in range(10):           # リスト xの要素をランダムに10個とりだす
    print random.choice(x),
```

実行結果の例

```
green white yellow white blue red blue black yellow yellow
```

次は random.gauss(0,1) は平均 0，分散 1 のガウス分布に従う乱数です：

```
# 平均0，分散1の正規分布に従う乱数
for i in range(10):
     print random.gauss(0,1),
```

実行結果の例

```
-1.98153307955 0.288243745581 -0.119123311085 1.80432993192
-0.160362179057 -0.0506597136487 -0.190873889601 -0.990606238348
0.673029984025 -1.32408246447
```

一般に平均 μ，分散 σ^2 のガウス分布は

$$P(x) := \frac{1}{\sigma\sqrt{2\pi}} e^{(x-\mu)^2/(2\sigma^2)}$$

で定義され，実数値をとる乱数がガウス分布に従うとは，その乱数が $[a,b]$ に入る確率が

$$\int_a^b P(x)dx$$

である事と定義されます。この乱数が，本当にガウス分布であるかどうかは，ヒストグラムを書いてみるとよく分かります。

```
import random
random.seed(0)
lis = [random.gauss(0,1) for j in range(20000)] # ガウス分布に従う乱数のリスト
H = histogram( lis , bins=40, normed=True, color='linen')  # lisのヒストグラム
p(x) = 1/sqrt(2*pi)*e^(-x^2/2)    # gauss分布
P = plot( p(x), (x,-5,5) )        # p(x)をプロット
(H+P).show()
```

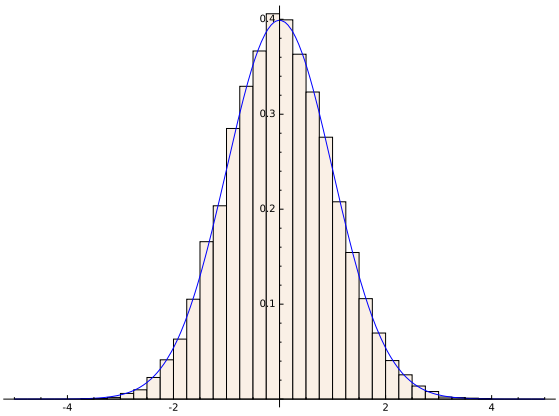


図 35　ガウス分布のヒストグラム

50.2　モンテカルロ法

50.2.1　円周率の近似

コンピューターにおける乱数の使い円周率を計算します。円周率を計算するアイディアは次の通りです：2 次元の領域 $[-1,1]\times[-1,1]$ に一様に n 個の点をばらまいて，半径 1 の円の中にある点の数を数えます。円の中の点の数を `count(n)` とするとき `4count(n)/n` が円周率を近似します。$[-1,1]\times[-1,1]$ の面積は 4 なので近似的に

$$(\text{ばらまいた点の数}) : (\text{円の中にある点の数}) \cong 4 : \text{円周率} \times (\text{半径}=1)^2$$

なので

$$\text{円周率} \cong 4\frac{\text{円の中にある点の数}}{\text{ばらまいた点の数}}$$

となります。ばらまく点の数を無限に多くする極限で上式の右辺は円周率に収束します。

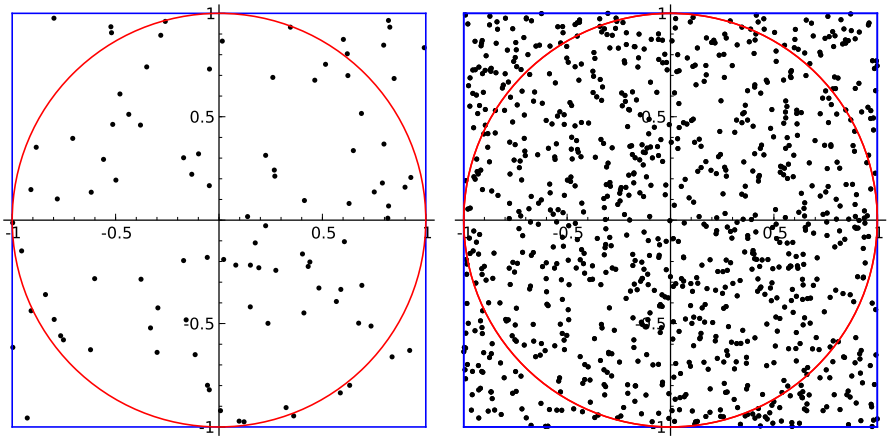


図 36　$[-1,1]^2$ に一様に点をばらまく

上のことをプログラムにすると次のようになります：

```
1 # モンテカルロ法で円周率を求める
2 import random
```

```
3  random.seed(0)
4
5  def count(n):
6      "n個の点を[-1,1]*[-1,1]にばらまいて，円の中にある点の個数を数える関数"
7      p=0     # 以下でカウントする値をpに入れる
8      for i in range(n):
9          x=random.uniform(-1,1)     # 乱数のx座標
10         y=random.uniform(-1,1)     # 乱数のy座標
11         if x^2+y^2<1:            # もしx^2+y^2<1ならば
12             p=p+1                    # pを一つ増やす
13     return p                         # カウントした点の数pを返す
14
15 n=100    # サンプル点の個数
16
17 pai = 4.0*count(n)/n     # これが円周率
18 print pai               #paiを出力する
```

実行結果の例

```
3.28000000000000
```

上のプログラムを実行してみましょう。出てきた結果が円周率の近似値です。サンプル点 n の値を 1 から 10000 ぐらいの範囲で変化させて，どのぐらい円周率の真の値 3.1415926535··· に近くなるか試してみましょう。やってみると上のプログラムはそれほど正確な円周率の値を出さないことが分かります。

次に，上のプログラムで n について計算した円周率を pai(n) と定義して，これを n を変化させた結果を見てみましょう。

```
1  # ここまでは上のプログラム
2  # の13行目までと同じにする。
3
4  def pai(n):
5      return 4.0*count(n)/n
6
7  for i in range(1,51):
8      print 400*i, pai(400*i)
```

実行結果の例

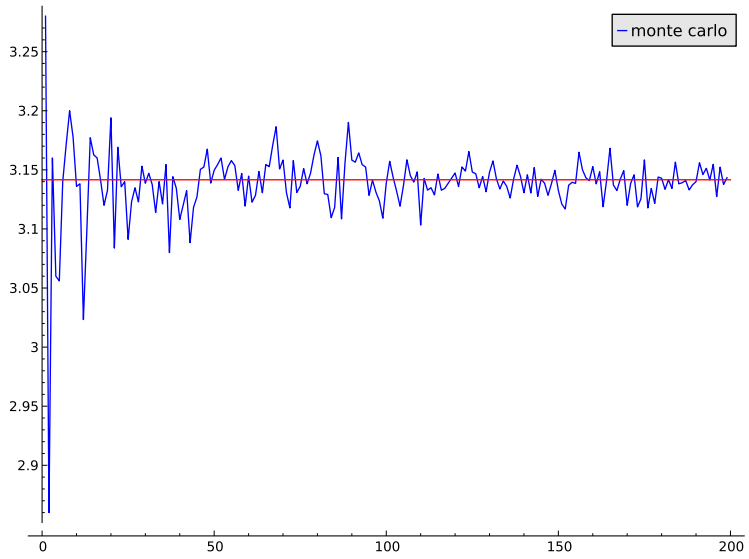
```
400 3.07000000000000
800 3.05000000000000
1200 3.14000000000000
......
19200 3.13895833333333
19600 3.11795918367347
20000 3.13840000000000
```

20000 個のサンプル点を取って計算してもそれほ 0.01 の桁が異なっています。これは確率論の基本的な事実で，分散を計算すればすぐにわかります。この方法ではサンプル点の数を n としたときに，円周率の誤差はおよそ $1/\sqrt{n}$ の割合になります。

さて，上のプログラムを改良して収束の様子をグラフで表してみましょう：

```
1  lis = []
2  for i in range(1,200):
3      lis.append([i,pai(100*i)])     # ペア[i,pi(100*i)]のリストを作る
4  p1 = list_plot(lis, plotjoined=True, legend_label='monte carlo')
5  p2 = line([(0,pi),(200,pi)], color='red')     # 円周率の真の値
6  (p1+p2).show()
```

計算には少し時間がかかりすぎる場合は上の `100*i` を `10*i` に変えてから実行してみましょう。次のようなグラフが表示されます：



50.2.2 収束の早さの比較

さて円周率は次のような交代級数で計算することもできます：

$$\pi = 4\arctan(1) = 4\int_0^1 \frac{1}{1+x^2}dx = 4\int_0^1 \sum_{n=0}^{\infty}(-1)^n x^{2n}dx = 4\sum_{n=0}^{\infty}(-1)^n\frac{1}{2n+1}$$
$$= 4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\cdots\right)$$

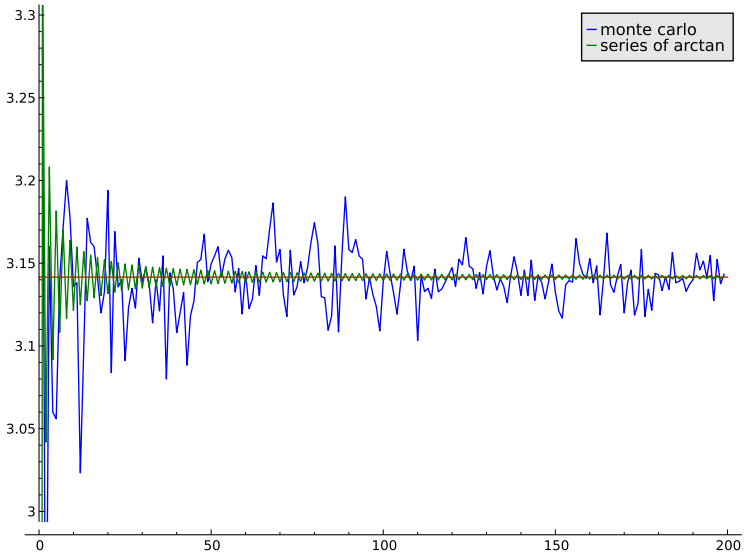
この級数の n 番目までの和の円周率の誤差は明らかに $1/n$ 程度です。このアルゴリズムを用いて計算して円周率と先ほどの乱数で計算した円周率との収束性を比較してみましょう：上のプログラムに続いて次を入力します：

```
# arctanの展開級数で求めた円周率の収束の様子をみる
def pi_arctan(n):
    "arctan(1)の級数展開のn個の和により近似した円周率"
    atan=0
    for j in range(n):
        atan=atan+(1.0/(2*j+1))*((-1)^j)
    return 4*atan

lis2=[]
for i in range(200):
    lis2.append((i,pi_arctan(5*i)))

p3 = list_plot(lis2, plotjoined=True, color='green',ymin=+3, ymax=3.3)
(p1+p2+p3).show()
```

arctan(1) の級数展開で計算した円周率の収束も必ずしも良いとは言えませんが，計算すればするほど，真の値に近づく事が分かります。一方モンテカルロ法は収束の速度は遅く，なかなか真の値に近づかない事が分



かります。モンテカルロ法で誤差 Δ の割合で結果を得るにはおよそ $1/\Delta^2$ 個のサンプル点を取る必要があります。モンテカルロ法が効力を発揮するのは高次元の積分値を求めるときです。

50.3　4 次元球の体積のモンテカルロ法による計算

4 次元球とは $\mathbb{R}^4$ の集合

$$B_4 = \{x = (x_1, x_2, x_3, x_4) \in \mathbb{R}^4 | x_1^2 + x_2^2 + x_3^2 + x_4^2 \leq 1\} \tag{35}$$

です。ここでは $[-1, 1]^4$ の中に点をばらまいて，B_4 の中にある点を数える事によって B_4 の体積を計算します：

$$(\text{ばらまいた点の数}) : (B_4 \text{ の中にある点の数}) \cong 2^4 : B_4\text{の体積}$$

なので

$$B_4\text{の体積} \cong 16 \times \frac{B_4 \text{ の中にある点の数}}{\text{ばらまいた点の数}}$$

です。したがって 4 次元球の体積を計算するためには次のようなプログラムを書けばよいでしょう：

```
1  ' モンテカルロ法で4次元単位球の体積を求める'
2  import random;    random.seed(0)
3  def count(n):
4      p=0     # カウントする数をpとする
5      for i in range(n):
6          x=random.uniform(-1,1)     # 乱数のx座標
7          y=random.uniform(-1,1)     # 乱数のy座標
8          z=random.uniform(-1,1)     # 乱数のz座標
9          w=random.uniform(-1,1)     # 乱数のw座標
10         if x^2+y^2+z^2+w^2 < 1:      # もしx^2+y^2+z^2+w^2<1ならば
11             p=p+1                  # pを一つ増やす
12     return p                       # カウントした点の数pを返す
13
14 n=2000    # サンプル点の個数
```

```
15 vol=16.0*count(n)/n      # これが体積
16 print vol                # volを出力する
```

上のプログラムを実行して出力を確認しましょう。 厳密な値は

$$|B_4| = \frac{\pi^2}{2} \cong 4.9348 \tag{36}$$

です。計算で得られた結果はこの値に近いでしょうか？

50.4　モンテカルロ法による積分

実関数 $f(x)$ に対して積分 $\int_0^1 f(x)dx$ を乱数を使って計算することを考えます。

まず，区間 $[0,1]$ に一様な乱数 X_j をばらまきます。このとき，X_j が区間 $(a,b) \subset [0,1]$ に入る確率は $b-a$ です。このとき

$$\int_0^1 f(x)dx = \frac{1}{N}\sum_{j=1}^{N} f(X_j) + \left(\frac{1}{\sqrt{N}}\text{程度の誤差}\right) \tag{37}$$

が成り立ちます[*26]。したがって，$N \to \infty$ とすれば

$$\int_0^1 f(x)dx = \lim_{N\to\infty}\frac{1}{N}\sum_{j=1}^{N} f(X_j) \tag{38}$$

となります。この事実を使って，積分 $\int_0^1 f(x)dx$ を近似することができます。積分区間が $[0,1]$ でないときには，適当にスケールを変えればよい。多重積分

$$\int_0^1 \cdots \int_0^1 f(x_1,\cdots,x_n)dx_1\cdots dx_n \tag{39}$$

を計算したいなら n 次元空間 $[0,1]^n$ に一様な乱数をばらまいてから和をとればよいということになります。つまり，$(X_j^1,\cdots,X_j^n)$, $j=1,\cdots,N$ を $[0,1]^n$ にばらまいた N 個の一様な乱数として上の多重積分を

$$\frac{1}{N}\sum_{j=1}^{N} f(X_j^1,\cdots,X_j^n) \tag{40}$$

によって近似します。この場合でも $1/\sqrt{N}$ のオーダーの誤差がともないます。

50.5　練習問題

$f(x,y,z) = \cos(x-y^2)e^{-x^2-y^2}/(x^2+z^2+1)$ とする。積分

$$\int_0^1 dx \int_0^2 dy \int_0^3 dz f(x,y,z) \tag{41}$$

をモンテカルロ法により計算するPython のプログラムを作成せよ。ただし，

- サンプル点の個数は 10000 個。
- 端末から実行したときに実行結果は積分の数値のみをプリントするようにする。
- ファイル名は `MonteCalroIntegral.py` とすること。

Python で $\cos(x)$ や e^x などの関数を使うときには，ファイルの先頭に

```
1 from math import *
```

と書いておけばよい。このとき $\cos(x)$, e^x はそれぞれ `cos(x)`, `exp(x)` で表す。

[*26] 重複大数の法則によれば誤差は $\log\log N/\sqrt{N}$ 程度だけど，$\log\log N$ は計算機で使用する範囲の N に対してはほぼ定数とおもってかまわない。