Python を利用した核計算 (2) — 確率論的手法 —

日本原子力研究開発機構　長家　康展

1　はじめに

確率論的手法とは一般的に言えばモンテカルロ法のことである。方程式を離散化して数値的に解く決定論的手法と対比して、モンテカルロ法は確率論的手法と呼ばれる。モンテカルロ法は乱数 (random number) を用いて数学的問題を解く数値解析手法の総称であり、原子炉物理の分野においては参照解を与える手法としてよく用いられる手法である。

原子炉物理の入門者は、参照解を与える手法であると聞けば、決定論的手法よりも難解な理論を用いているのではないかとイメージするため、モンテカルロ法を勉強しようとは思わないかもしれない。中級者でも、プロダクションレベルのモンテカルロ計算コード MVP[1] や MCNP[2] が普及しているので、計算手法の原理をよく知らずに使っているかもしれない。本稿は、モンテカルロ計算の基礎手法やモンテカルロ計算アルゴリズムを学びたい読者に対し、プログラミング言語 Python を用いて実際にプログラムを書いて実行することによりそれらを理解してもらうことを目的としている。

本稿では以下の順序に従って解説する。

1. 乱数の発生方法、サンプリング手法について解説し、モンテカルロ法で円周率 π を計算する。
2. 中心に点線源を配置した球体系における中性子束を計算する。
3. 均質球体系に対する 1 群計算により実効増倍率を計算する。
4. 均質球体系に対する 2 群計算により実効増倍率を計算する。

Python 言語で記述した上記のサンプルプログラムは、Python に精通していなくてもコードを見ただけで何をしているのかを理解できるように記述している。計算モデルについてもできる限り単純なモデルを用い、アルゴリズムそのものを理解できるようなコードとしている。読者においては、実際にコードを実行することにより、コードで遊びながらモンテカルロ法を学んでいただければ幸いである。

2　モンテカルロ法の基礎手法

2.1　乱数の生成

乱数はモンテカルロ法の基礎であり、モンテカルロ計算において非常に重要な役割を果たしている。モンテカルロ計算では、どの事象が起こるのかを確率論的に決定し、忠実に対象となる現象をシミュレーションしているが、乱数を用いてある事象に対する確率分布からのサンプリングを行い、事象を決定している。

乱数を発生する仕組みは乱数ジェネレータ (random number generator) と呼ばれ、通常、乱数はソフトウェア的に (計算機上のアルゴリズムにより) 生成される。しかし、計算機上の変数が有限ビットで表わされるため、ソフトウェア的に真の乱数を生成することは不可能であり、乱数列は周期を持つことになる。即ち、同じ乱数が現れると、それ以降、前の乱数列と全く同じ乱数列になってしまうことになる。それゆえ、このようなソフトウェア的な乱数ジェネレータで生成した乱数は擬似乱数 (pseudorandom number) と呼ばれる。

擬似乱数を生成するアルゴリズムはこれまで数多くのものが提案されてきているが、粒子輸送モンテカルロ計算では Lehmer によって提案された線形合同法 (linear congruential generator)[3] が最も多く用いられている。この方法では次の漸化式により、0 から 1 の間の一様乱数を生成する。

$$S_i = (aS_{i-1} + c) \bmod m \tag{1}$$

$$\xi_i = \frac{S_i}{m} \tag{2}$$

ここで、a, c, m, S_i は正の整数である。また、m を法 (modulus) とする剰余演算から分かるように S_i のとりうる範囲は $0 \leq S_i < m$ であり、ξ_i のとりうる範囲は $0 \leq \xi_i < 1$ である。しばしば、c は 0 とされることが多い

が、このときのアルゴリズムは乗算合同法 (multiplicative congruential generator) と呼ばれる。これに対し、$c \neq 0$ のときは混合合同法 (mixed congruential generator) と呼ばれる。上記漸化式はある初期値 S_0 を与える必要があり、その初期値はシード (seed) と呼ばれる。シードを変えることにより異なる乱数列を生成することができる。計算結果がおかしいと思われるとき (すなわち、発生確率の非常に小さな結果が得られた可能性があるとき)、シードを変えて再計算し、その結果が偶然得られたものであるかどうかを確かめることができる。

それではまず手始めに Python 言語で乱数を発生させてみよう。上記のアルゴリズムを Python スクリプトで記述し、乱数を発生させてもよいが、Python には標準ライブラリで `random` モジュールという擬似乱数を生成するためのモジュールが既に用意されている。本稿では `random` モジュールを使って乱数を発生させることにする。Python を対話モードで起動し、以下のようなコマンドを入力してみよう。Windows 上の Anaconda Prompt から対話モードを起動すると以下のようになる。

```
(base) C:\Users\nagaya> python
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit
(AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

">>>" は Python 対話モードのプロンプトである。(もし読者が Jupyter Notebook を使えるのであれば、以下で示すコマンドは Jupyter Notebook のセルに入力して実行するのが便利である。その場合はプロンプトが "`In [ ]`" となっているはずである。適当に読み換えて欲しい。)

```
1 >>> import random
2 >>> random.random()
```

$[0, 1)$ 区間の数値が 1 つ出力されるはずである。このコマンドについて解説すると、1 行目の "`import random`" は、`random` モジュールをインポートし、このモジュールで定義されている関数を利用できるようにしている。2 行目の "`random.random()`" は `random` モジュールの `random()` 関数を呼び出している。

`random()` 関数を呼び出したとき、違う値が返ってくることを確認するために複数回 `random()` 関数を呼び出してみよう。以下のコマンドは、`for` 文を使って 3 回 `random()` 関数を呼び出すスクリプトである。

```
1 >>> for i in range(3):
2 ...     random.random()
3 ...
```

2 行目のコマンドを記述する前にインデント (半角スペース 4 つ) を追加する必要がある。これは Python 言語の規則であり、`for` 文や `if` 文のブロック内ではインデントしなければならない。3 行目の空白行は Enter キーを入力し、この `for` 文を実行している。このスクリプトを実行すると 3 回違う数値が出力され、`random.random()` という同じコマンドを実行しても異なる数値が得られることが確認できるはずである。

モンテカルロプログラムを記述するには、`random.random()` 関数だけで十分であるが、乱数列のシードの指定の仕方についても説明しておこう。`random` モジュールでシードを設定するには `random.seed()` 関数を用いる。

```
1 >>> random.seed(1)
2 >>> random.random()
```

上記の例ではシードを 1 として乱数列を初期化した後、乱数を 1 つ生成している。以下のように乱数を発生させる度に初期化すれば、同じ数値の乱数が得られる。

```
1 >>> for i in range(3):
2 ...     random.seed(1)
3 ...     random.random()
4 ...
```

以上で Python 言語による乱数の生成法が分かったと思う。ここで 1 つ説明しておかなければならないが、Python 言語標準である `random` モジュールの乱数ジェネレータは線形合同法ではなく、メルセンヌ・ツイスタと呼ばれる乱数ジェネレータが使われている [4]。メルセンヌ・ツイスタは、線形合同法と比べて周期も長く (周期 $2^{19937} - 1$)、乱数の質がよいことが知られているが、ジャンプ・アルゴリズム[a]がなかったため (最近開発されたばかりである)、一般的な粒子輸送モンテカルロコードでは採用されていない。

[a] "jump ahead", "skip ahead" とも呼ばれる。

例題 1: 円周率 π の計算

Python の `random` 関数を用いてモンテカルロ法により円周率 π を計算してみよう。計算の原理は単純である。xy 平面上に単位長さの正方形と半径 1 の円を (正方形の内側に接するように) 描いてランダムに (x, y) 座標を選択する。選択した座標が円の内側に入る確率を計算する。座標を一様に選択すれば、その確率は "円の面積/正方形の面積" となり、選択する座標の数を大きくしていけば π に近づいていくことになる。プログラムを作成する際には、`random` 関数が $[0, 1)$ の区間の一様乱数を生成するので、第 1 象限だけで考えることにする。計算のイメージを図示すると図 1 のようになる。

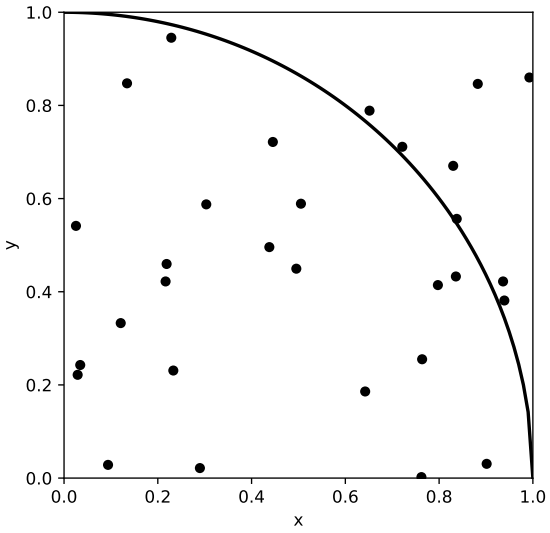


図 1: モンテカルロ法を用いた円周率の計算

Python を対話モードで起動し、座標を 1 点だけランダムに選択し、円の内側に入っているのか判定してみよう。

```
1 >>> import random
2 >>> random.seed(1)
3 >>> x = random.random()
4 >>> y = random.random()
5 >>> print(x,y)
6 0.13436424411240122 0.8474337369372327
```

1 行目と 2 行目は既に説明済みである。3 行目は x 座標、4 行目は y 座標をランダムに選択している。5 行目で選択された座標を `print` 関数で出力している。単純に ">>> x, y" としても出力できる。何をさせているのかを明示させたいときや出力形式を指定するときに `print` 関数を用いる。以下対話形式で変数の値を出力させるときには、`print` 関数を使わない形式を用いる。この座標が円の内側に入っているかどうかを判別するには、原点からの距離を計算して円の半径と比較 (プログラム上は 2 乗値 $R^2 = 1$ で比較) すればよい。

```
1 >>> r2 = x**2 + y**2
2 >>> r2
3 0.7361976885952998
```

`r2` は原点から座標までの直線距離の 2 乗である。`r2` < 1 であるので円の内側にあることは明白であるが、後々のために明示的に判定してみる。

```
1 >>> if r2 <= 1:
2 ...     print(True)
3 ... else:
4 ...     print(False)
5 ...
6 True
```

Ture と False は文字列ではないので、ダブルクォーテーションで囲まれてはいないことに注意されたい。これらは Python のブール型定数である。

上記のスクリプトを組み合わせれば、π をモンテカルロ法で計算するプログラムは以下のようになるだろう。

```
>>> import random
>>> random.seed(1)
>>> n_points = 100000
>>> n_counts = 0
>>> for i in range(n_points):
...     x = random.random()
...     y = random.random()
...     r2 = x**2 + y**2
...     if r2 <= 1:
...         n_counts += 1
...
>>> n_counts
78446
```

3 行目の n_points は選択した全座標の数であり、このケースでは 10 万点の座標を選択している。4 行目の n_counts はそのうち円の内側に入った数であり、78,446 点が円の内側に入ったということである。この結果を用いて、円の内側に入る確率に 4.0 をかけると円周率 π を計算することができる。

```
>>> float(n_counts)/float(n_points)*4.0
3.13784
```

float は整数値を浮動小数点形式に変換するための関数である。数値計算では“整数/整数=整数” となり、丸め誤差が発生するので、整数を用いた除算に対しては浮動小数点形式に変換するのが常套手段である。Python3 では“整数/整数=実数” と解釈されることになったので、float 関数を用いなくてもよいが、型変換していることを明示しておいた方がよいだろう。参照解の $\pi = 3.14159265\ldots$ と比べると 2 桁目までしか一致していないが、選択する座標の数を大きくしていけば参照解に近づくことを確認することができる。

2.2 サンプリング法

サンプリング法はモンテカルロ法の基礎となる手法であり、上記で述べた一様乱数を用いてある確率分布からどのような事象が起こるかを決定する。今、事象 $E_1, \cdots, E_n$ は独立で、それぞれ確率 $p_1, \cdots, p_n$ で起こるものとする。このとき、

$$p_1 + \cdots + p_n = 1 \tag{3}$$

であり、乱数 ξ が

$$p_1 + \cdots + p_{i-1} \leq \xi < p_1 + \cdots + p_i \tag{4}$$

の範囲にあるとすると、事象 E_i が起こったと決定する。

一例として、中性子が原子核と衝突し、どのような事象が起こるかを決定することを考えてみる。簡単のため、核分裂反応、捕獲反応、散乱反応しか起こらないとし、それらの断面積をそれぞれ $\Sigma_\mathrm{f}, \Sigma_\mathrm{c}, \Sigma_\mathrm{s}$ とする。全断面積は

$$\Sigma_\mathrm{t} = \Sigma_\mathrm{f} + \Sigma_\mathrm{c} + \Sigma_\mathrm{s} \tag{5}$$

であり、各反応が起こる確率は $p_1 = \Sigma_\mathrm{f}/\Sigma_\mathrm{t}, p_2 = \Sigma_\mathrm{c}/\Sigma_\mathrm{t}, p_3 = \Sigma_\mathrm{s}/\Sigma_\mathrm{t}$ である。このとき、乱数 ξ の値により次のように反応を決定することができる。

$$0 \leq \xi < \frac{\Sigma_\mathrm{f}}{\Sigma_\mathrm{t}} \quad \rightarrow \quad \text{核分裂反応} \tag{6}$$

$$\frac{\Sigma_\mathrm{f}}{\Sigma_\mathrm{t}} \leq \xi < \frac{\Sigma_\mathrm{f}}{\Sigma_\mathrm{t}} + \frac{\Sigma_\mathrm{c}}{\Sigma_\mathrm{t}} \quad \rightarrow \quad \text{捕獲反応} \tag{7}$$

$$\frac{\Sigma_\mathrm{f}}{\Sigma_\mathrm{t}} + \frac{\Sigma_\mathrm{c}}{\Sigma_\mathrm{t}} \leq \xi < 1 \quad \rightarrow \quad \text{散乱反応} \tag{8}$$

図 2 はこのサンプリングに対する乱数と各反応の起こる確率を示している。

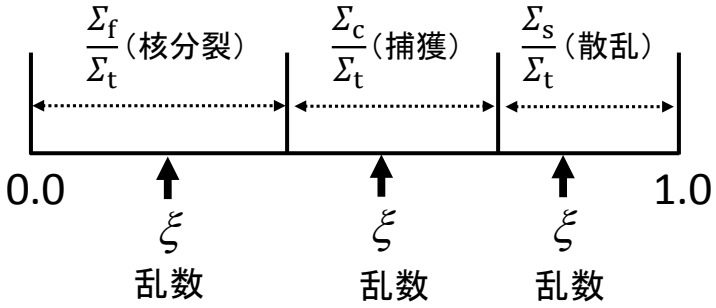


図 2: 中性子反応に対する確率

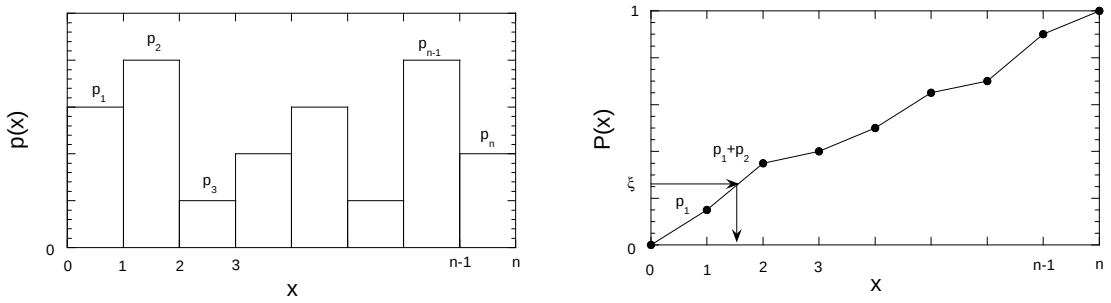
この例では、確率密度関数 (probability density function, PDF) がステップ関数で与えられる場合のサンプリングを示している。このときの確率密度関数は一般的に次のように定義することができる。

$$p(x) = p_i, \quad (i-1 \leq x \leq i) \ \text{ for } \ i = 1, 2, \cdots, n \tag{9}$$

この関数を図で表わすと図 3(a) のようになる。そこで、累積分布関数 (cumulative distribution function, CDF) を次のように定義する。

$$P(x) = \int_0^x p(t)dt, \quad (0 \leq x \leq n) \tag{10}$$

すると、図 3(b) に示すように $P(0) = 0, P(n) = 1$ の単調増加関数となる。



(a) 確率密度関数　　(b) 累積分布関数

図 3: 離散確率密度関数と累積分布関数

ここで、乱数 ξ と累積分布関数を次のように結びつけると、x は ξ の関数として一意的に決定することができる。

$$\xi = P(x) \tag{11}$$

即ち、x が $i-1 \leq x < i$ に入る確率は P_i であり、それに対応した事象を決定することができる。従って、乱数を使って事象を決める際に必ず (11) 式の逆関数

$$x = P^{-1}(\xi) \tag{12}$$

を求める必要がある。

例題 2: 中性子反応事象の決定

上記の方法で各事象が確率どおりにサンプリングできるのかどうか、実際にプログラムして確かめてみよう。上で示した例と同じく、中性子は原子核と衝突して核分裂反応、捕獲反応、散乱反応しか起こらないものとする。各断面積の値は、$\Sigma_{\rm f} = 0.24, \Sigma_{\rm c} = 0.26, \Sigma_{\rm s} = 0.5$ とする。Python を対話モードで起動し、以下の変数を定義しよう。

```
>>> SigF = 0.24; SigC = 0.26; SigS = 0.5
```

各変数がどの反応断面積を表すのかは明らかである。1 行で書いているが、もちろん 3 行に分けてもよい。核反応が起こる確率を計算するために全断面積を定義する。

```
>>> SigT = SigF + SigC + SigS
>>> SigT
1.0
```

全断面積を用いて各反応の起こる確率は次のように計算できる。

```
>>> prob_fis = SigF/SigT; prob_cap = SigC/SigT; prob_sct = SigS/SigT
```

これらの確率を用いて、(6) 式から (8) 式に現れている累積分布関数の節点を計算すればよい。この例の場合、$\Sigma_{\mathrm{f}}/\Sigma_{\mathrm{t}}$ と $\Sigma_{\mathrm{f}}/\Sigma_{\mathrm{t}} + \Sigma_{\mathrm{c}}/\Sigma_{\mathrm{t}}$ の 2 つの値だけ計算すれば十分であるが、事象の数が増えたときにも適用できるように一般的な形式で累積分布関数を計算しよう。(多群問題である群から他の群へ散乱するときにもここで述べるサンプリング法を使うことができる。) 確率密度関数のデータを numpy モジュールの配列へ格納する。

```
>>> import numpy as np
>>> prob = np.array([prob_fis, prob_cap, prob_sct])
>>> prob
array([0.24, 0.26, 0.5 ])
```

1 行目で numpy モジュールを np というエイリアス名で使えるようにインポートし、2 行目で numpy の array に確率データを格納している。3 行目は prob の中のデータを確認するためのコマンドで、4 行目がその出力である。numpy モジュールの配列を利用したのは、numpy モジュールの便利な関数を利用し、累積分布関数データを作成するためである。作成方針は以下のとおりである。

1. 1 行 3 列の prob データを、3 つ縦に並べて 3 行 3 列の行列を作成する。
2. 下三角行列に変換する。
3. 各行の和をとる。

numpy の tile 関数を使って 3 行 3 列の行列を作成する。

```
>>> tmp1 = np.tile(prob, (3,1))
>>> tmp1
array([[0.24, 0.26, 0.5 ],
       [0.24, 0.26, 0.5 ],
       [0.24, 0.26, 0.5 ]])
```

tile 関数の引数“(3,1)” は prob を 3×1 で並べるという指定である。numpy の tril 関数を使って下三角行列に変換する。

```
>>> tmp2 = np.tril(tmp1)
>>> tmp2
array([[0.24, 0.  , 0.  ],
       [0.24, 0.26, 0.  ],
       [0.24, 0.26, 0.5 ]])
```

numpy の sum 関数を使って各行の和をとる。

```
>>> tmp3 = tmp2.sum(axis=1)
>>> tmp3
array([0.24, 0.5 , 1.  ])
```

sum 関数の引数“axis=1” は各行の和をとることを指定している。以上をまとめると

```
>>> np.tril(np.tile(prob, (3,1))).sum(axis=1)
array([0.24, 0.5 , 1.  ])
```

として 1 行で記述することができる。このままでは、prob の要素数が 3 個の場合しか対応しないので、“3” を“prob.size” に変えると一般的な場合に対応させることができる。

```
>>> np.tril(np.tile(prob, (prob.size, 1))).sum(axis=1)
array([0.24, 0.5 , 1.  ])
```

以上で、確率データを累積確率データに変換するコマンドが分かったので、cum_prob という変数に格納しておこう。

```
>>> cum_prob = np.tril(np.tile(prob, (prob.size ,1))).sum(axis=1)
```

累積確率データが用意できたので、1 回だけ乱数を生成してどの反応が起きるのかをサンプリングにより決定してみよう。最も単純な方法は、乱数と累積確率データの節点を順番に比較していき、乱数値より節点が大きくなったときのインデックス値を探索すればよい。これを Python で実装すれば以下のようになるだろう。

```
>>> import random
>>> random.seed(1)
>>> rn = random.random()
>>> rn
0.13436424411240122
>>> for i, v in enumerate(cum_prob):
...     if rn < v:
...         event = i
...         break
...
>>> event
0
```

インデックス 0 は核分裂反応であるので正しくサンプリングできていることが分かる。for 文の範囲を指定するために enumerate 関数が用いられているが、cum_prob 配列の値を順番に取り出し、インデックスを i、値を v に格納している。

インデックス出力ではどの反応が起きたのか分かりにくいので、明示して出力させると以下のようになる。

```
>>> if event == 0:
...     print("fission")
... elif event == 1:
...     print("capture")
... else:
...     print("scattering")
...
fission
```

ここで示した探索アルゴリズムは原始的な方法であり、データ数が大きくなると計算時間がかかるようになる。実用的には、二分木探索などの方法を使うことが望ましいが、本稿ではあまり大きなデータは扱わないので、この方法を採用することにする。

ここまでくれば複数個の乱数を生成してどの反応が起きたのかを決定するのは簡単である。10 回反応を決定する場合は以下のようになる。

```
>>> import random
>>> random.seed(1)
>>> import random
>>> for j in range(10):
...     rn = random.random()
...     print("rn␣=␣", rn, "␣", end="")
...     for i, v in enumerate(cum_prob):
...         if rn < v:
...             event = i
...             break
...     if event == 0: print("fission")
...     elif event == 1: print("capture")
...     else: print("scattering")
...
rn =  0.13436424411240122  fission
rn =  0.8474337369372327  scattering
rn =  0.763774618976614  scattering
rn =  0.2550690257394217  capture
```

```
rn =  0.49543508709194095  capture
rn =  0.4494910647887381  capture
rn =  0.651592972722763  scattering
rn =  0.7887233511355132  scattering
rn =  0.0938595867742349  fission
rn =  0.02834747652200631  fission
```

生成した乱数に応じて正しく事象が選択されているのが確認できる。

次にもっと試行回数を多くして、カウント数から計算した発生確率が最初に入力した確率になるのかどうか確認してみよう。試行回数を `n_trials` で定義し、各事象のカウンタを `n_fis`, `n_cap`, `n_sct` として各事象が発生するたびに 1 ずつカウントするようにコードを変更する。試行回数は 100,000 とし、非常に多くなるので、`print` 出力は削除する。

```
>>> random.seed(1)
>>> n_trials = 100000; n_fis = 0; n_cap = 0; n_sct = 0
>>> for j in range(n_trials):
...     rn = random.random()
...     for i, v in enumerate(cum_prob):
...         if rn < v:
...             event = i
...             break
...     if event == 0: n_fis += 1
...     elif event == 1: n_cap += 1
...     else: n_sct += 1
...
>>> n_fis, n_cap, n_sct
(24106, 25934, 49960)
>>> float(n_fis)/float(n_trials)
0.24106
>>> float(n_cap)/float(n_trials)
0.25934
>>> float(n_sct)/float(n_trials)
0.4996
```

核分裂反応確率は 0.24、捕獲反応確率は 0.26、散乱反応確率は 0.5 であったので、各事象の起こる頻度はほぼ確率どおりとなっていることが確認できる。

これまで確率密度関数が離散的な (ステップ関数で表わされる) 場合について述べてきたが、一般的には連続的な確率密度関数が与えられる場合がある。この場合についても同様にして、乱数を用いてある事象をサンプリングすることができる。確率密度関数が区間 $a \leq x < b$ で定義され、図 4(a) のような関数 $p(x)$ で与えられたとすると、累積分布関数は次式のようになる。

$$P(x) = \int_a^x p(t)dt \tag{13}$$

累積分布関数は図 4(b) に示すように単調増加関数になるので、乱数が $[0, 1)$ の区間で一様分布していれば、乱数 ξ に対応した $x(\xi)$ を一意的に決定することができる。即ち、

$$x(\xi) = P^{-1}(\xi) \tag{14}$$

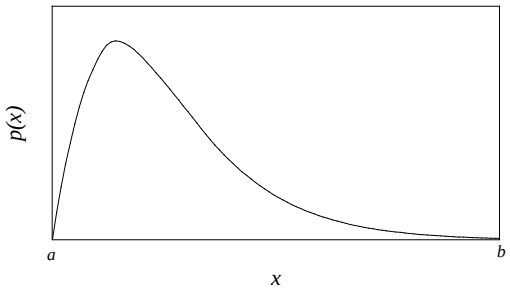
である。

連続確率密度関数の例として、乱数を用いて飛行距離を決定することを考える。今、中性子が原点 $(x = 0)$ にあり、最初の衝突を距離 x と $x + dx$ の間で起こす確率 $p(x)dx$ は

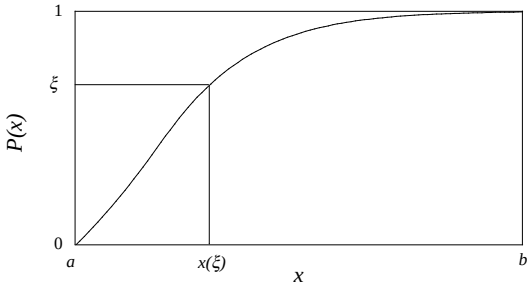
$$p(x)dx = \Sigma_\mathrm{t} \exp(-\Sigma_\mathrm{t} x)dx \tag{15}$$

で与えられる。ここで、$0 \leq x \leq \infty$ である。(15) 式を x のとりうる範囲で積分すると

$$\int_0^\infty p(x)dx = \int_0^\infty \Sigma_\mathrm{t} \exp(-\Sigma_\mathrm{t} x)dx = 1 \tag{16}$$



(a) 確率密度関数



(b) 累積分布関数

図 4: 連続確率密度関数と累積分布関数

であり、$p(x)$ は確率密度関数とみなすことができる。乱数 ξ から飛行距離 x を決定するために累積分布関数を計算すると

$$P(x) = \int_0^x \Sigma_\mathrm{t} \exp(-\Sigma_\mathrm{t} x) dx = 1 - \exp(-\Sigma_\mathrm{t} x) \tag{17}$$

であり、次式より飛行距離 x を決定することができる。

$$\xi = 1 - \exp(-\Sigma_\mathrm{t} x) \tag{18}$$

$$x = -\frac{1}{\Sigma_\mathrm{t}} \ln(1 - \xi) \tag{19}$$

また、$(1-\xi)$ の分布と ξ の分布は同じであるから、次式で飛行距離 x を決定してもよい。

$$x = -\frac{1}{\Sigma_\mathrm{t}} \ln \xi \tag{20}$$

例題 3: 飛行距離の決定

(20) 式で決定した飛行距離が (15) 式の確率密度関数に従っているのかどうかプログラムして確かめてみよう。全断面積を 1.0 cm^{-1} として、1 回だけサンプリングして飛行距離を決定すると以下のようになる。

```
1 >>> import random
2 >>> import math
3 >>> random.seed(1)
4 >>> SigT = 1.0
5 >>> - math.log(random.random())/SigT
6 2.0072009271185753
```

2 行目で `math` モジュールをインポートしているが、`log` 関数を利用するためである。このサンプリングを複数回繰り返すには、`for` ループを用いて繰り返すだけでよいが、この例題では飛行距離の頻度分布を作成することが目的である。そのために距離をいくつかの領域に分け、その領域に入った数をカウントするというプログラムを作成する[b]。これがモンテカルロ計算で言うところの「タリーをとる」という作業にあたるものである。モンテカルロ計算では、タリーをとるために分割された領域のことを「ビン」と呼ぶことが多い。エネルギーに対しては「エネルギービンに分けてタリーをとる」という言い方になる。ここでも各領域まで飛行してきた中性子の数をカウントするためのビンを用意して頻度分布を計算してみよう。飛行距離は理論的に無限大の距離まで到達する可能性があるが、全断面積が 1.0 cm^{-1} であるので、5 cm ぐらいまで距離をスコアリングすればよいであろう。ビンの数は後で変更するものとして、とりあえずは 10 個に設定し、1 次元配列としてカウンタを用意すると Python スクリプトは以下のようになるだろう。

```
1 >>> import random
2 >>> import math
3 >>> import numpy as np
```

[b] Python には優れた統計処理モジュールが用意されており、それを用いて頻度分布を作成することが可能であるが、プログラミングを学ぶということと後でモンテカルロ法の統計処理をすること考えて自前で頻度分布を作成する。

```
>>> SigT = 1.0
>>> n_trials = 100000
>>> dist_max = 5.0
>>> n_bins = 10
>>> n_counts = np.zeros(n_bins)
>>> bin_width = dist_max/float(n_bins)
```

3 行目は numpy モジュールの配列を利用するためにインポートし、エイリアス名として np と定義している。4 行目は全断面積、5 行目は試行回数で 10 万回、6 行目はスコアリングする最大飛行距離、7 行目はビンの数である。8 行目はカウンタで、numpy モジュールの zeros 関数を用いて 0 に初期化している。n_counts とすれば、内部の状態を確認することができる。

```
>>> n_counts
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

9 行目はビンの幅で、後で使用するために定義している。カウントする準備が整ったので、n_trials 回繰り返してカウントする。

```
>>> random.seed(1)
>>> for i in range(n_trials):
...     dist = - math.log(random.random())/SigT
...     ibin = int(dist/bin_width)
...     if ibin < n_bins: n_counts[ibin] += 1
...
```

4 行目でサンプリングで決定した飛行距離に対するビンのインデックスを決定している。飛行距離がビン幅の何倍になっているかを計算し、整数値に切り捨てれば対応するビンのインデックスとなる (Python では配列のインデックスが 0 から始まることに注意)。5 行目は、飛行距離が dist_max を超える可能性があるため、インデックスが n_bin より小さいときだけスコアリングするようにしている。結果を出力すると以下のようになる。

```
>>> n_counts
array([39166., 24244., 14206.,  8900.,  5309.,  3255.,  1873.,  1210.,
         707.,   416.])
```

この結果は各ビンのカウント数であるので、試行回数で割れば各ビンに入る確率となる。

```
>>> prob = n_counts/float(n_trials)
>>> prob
array([0.39166, 0.24244, 0.14206, 0.089  , 0.05309, 0.03255, 0.01873,
       0.0121 , 0.00707, 0.00416])
```

(15) 式の確率密度関数と比較するために、この結果を確率密度に変換する。即ち、ビンの幅で割ると以下のようになる。

```
>>> pdf_counts = prob/bin_width
>>> pdf_counts
array([0.78332, 0.48488, 0.28412, 0.178  , 0.10618, 0.0651 , 0.03746,
       0.0242 , 0.01414, 0.00832])
```

この結果をヒストグラムとしてプロットするので、ビンの中央値のデータをリストで作成しておこう。

```
>>> x_counts = bin_width*(np.arange(n_bins) + 0.5)
>>> x_counts
array([0.25, 0.75, 1.25, 1.75, 2.25, 2.75, 3.25, 3.75, 4.25, 4.75])
```

np.arange は 0 を始点として n_bins 個の整数値を array 配列として生成する。カウントで得られた確率密度関数と参照解である (15) 式の確率密度関数を比較するために、参照解をグラフにプロットするためのデータを用意しよう。

```
>>> x_ref = np.linspace(0, dist_max, 201)
>>> y_ref = SigT*np.exp(- SigT*x_ref)
```

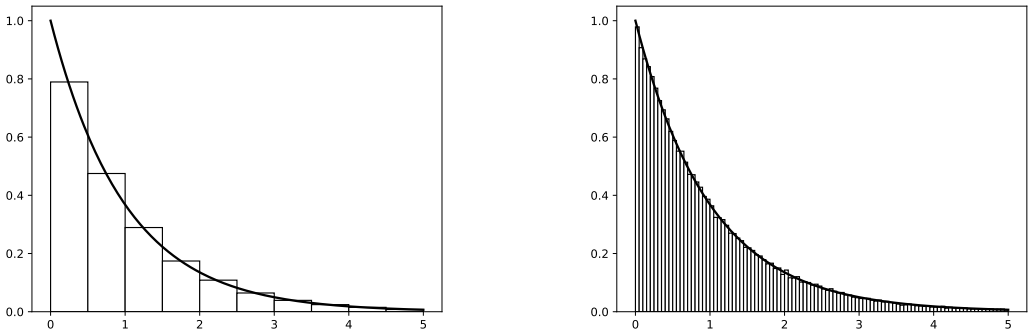
1 行目の np.linspace 関数は、0 から dist_max の区間で等間隔に 201 点を生成する。(区間 $[0, \texttt{dist\_max}]$ を 200 領域に等間隔で分割する。)

Python でグラフをプロットするためによく用いられるパッケージとして matplotlib が知られている。このツールの `pyplot` モジュールをインポートしてグラフをプロットしてみよう。

```
1 >>> import matplotlib.pyplot as plt
2 >>> plt.bar(x_counts, pdf_counts, width=bin_width, color="white", edgecolor="black")
3 <BarContainer object of 10 artists>
4 >>> plt.plot(x_ref, y_ref, color="black")
5 [<matplotlib.lines.Line2D object at 0x00000000068E4080>]
6 >>> plt.show()
```

カウントから得られた得られた結果は `bar` 関数でヒストグラムとしてプロットしている。オプションが 3 つあるが、`width=`はヒストグラムの柱の幅である。これを `bin_width` としないと柱が離れてしまう。`color=`は柱の色で白に指定し、`edgecolor=`は柱の輪郭の色で黒を指定している。これらを指定しないとデフォルトの色でベタ塗されてしまうので、参照解と比較が見にくくなる。参照解は `plot` 関数でプロットし、`color=`で線の色を黒に変更している。3 行目と 5 行目はユーザの入力値ではなく、Python からの出力値である。

プロットした結果は、図 5(a) のようになる。参照解と大体一致しているのが分かるがビン数が小さいため解像度があまりよくない。ビン数を 100 にして計算した結果が図 5(b) であり、(20) 式から得られるサンプリング値は、(15) 式の確率密度関数に従うことが確認できる。図 5 の横軸は “飛行距離 [cm]” で、縦軸は “確率密度” である。Python スクリプトにおいて重要なコマンドのみを示すため、図 5 には意図的に軸ラベルや凡例をつけていない。`pyplot` モジュールには `xlabel`, `ylabel`, `legend` などの関数が用意されているので、読者の方でプロット図を修飾してほしい。



(a) ビン数=10　　(b) ビン数=100

図 5: 飛行距離に対する確率密度の比較

---

2 変数の連続確率密度関数の例として、等方的に散乱された粒子の飛行方向を決定することを考えてみよう。微小立体角 $d\Omega$ は極角 (polar angle)θ と方位角 (azimuthal angle)ϕ を用いて次式で表わされる。

$$d\Omega = d\theta \sin\theta d\phi \tag{21}$$

(21) 式を全立体角で積分すると

$$\int d\Omega = \int_0^{\pi} d\theta \sin\theta \int_0^{2\pi} d\phi = 4\pi \tag{22}$$

であるから、単位立体角に粒子が散乱する確率 $p(\Omega)d\Omega$ は

$$p(\Omega)d\Omega = \frac{d\Omega}{4\pi} = \frac{d\theta \sin\theta}{2}\frac{d\phi}{2\pi} \tag{23}$$

で与えられる。極角と方位角はそれぞれ独立であるから、

$$p(\mu)d\mu = \frac{1}{2}d\mu, \tag{24}$$

$$\mu = \cos\theta, \tag{25}$$

$$p(\theta)d\theta = \frac{1}{2\pi}d\theta \tag{26}$$

と分離することができる。よって、乱数 ξ_1, ξ_2 を用いて

$$\int_{-1}^{\mu} \frac{d\mu}{2} = \xi_1, \tag{27}$$

$$\int_{0}^{\phi} \frac{d\phi}{2\pi} = \xi_2 \tag{28}$$

とすると、極角と方位角を次のように決定することができる。

$$\mu = 2\xi_1 - 1, \tag{29}$$

$$\phi = 2\pi\xi_2 \tag{30}$$

例題 4: 等方散乱に対する角度の決定

等方散乱に対する飛行方向は (29) 式と (30) 式でサンプリングした結果から決めることができることが分かった。これらの式を使って、立体角の 4π 方向についてまんべんなくサンプリングが行われているかどうかを確認してみよう。まず、(μ, ϕ) を 1 セットサンプリングして決定してみる。

```
1 >>> import random
2 >>> import math
3 >>> random.seed(1)
4 >>> costh = 2.0*random.random() - 1.0
5 >>> phi = 2.0*math.pi*random.random()
6 >>> costh, phi
7 (-0.7312715117751976, 5.324583204732311)
```

4 行目の `costh` は (29) 式の μ であり、5 行目の `phi` は (30) 式の ϕ である。この極座標で表された点を xyz 座標で表現すると以下のようになる。

$$u = \sin\theta\cos\phi \tag{31}$$

$$v = \sin\theta\sin\phi \tag{32}$$

$$w = \cos\theta \tag{33}$$

ここで

$$\sin\theta = \sqrt{1 - \cos^2\theta} \tag{34}$$

である。以上の式に従って (u, v, w) 座標を計算すると

```
1 >>> sinth = math.sqrt(1.0 - costh**2)
2 >>> u = sinth*math.cos(phi)
3 >>> v = sinth*math.sin(phi)
4 >>> w = costh
5 >>> u, v, w
6 (0.3919709387244532, -0.5582121095618475, -0.7312715117751976)
```

となる。上記の計算を複数回繰り返し、得られた (u, v, w) 座標を保存して 3 次元的にプロットしてみる。試行回数を 1,000 回と、(u, v, w) 座標を座標別で保存するためのコンテナを用意する。

```
1 >>> random.seed(1)
2 >>> n_trials = 1000
3 >>> x = []; y = []; z = []
```

3 行目はコンテナとして空のリストを用意している。

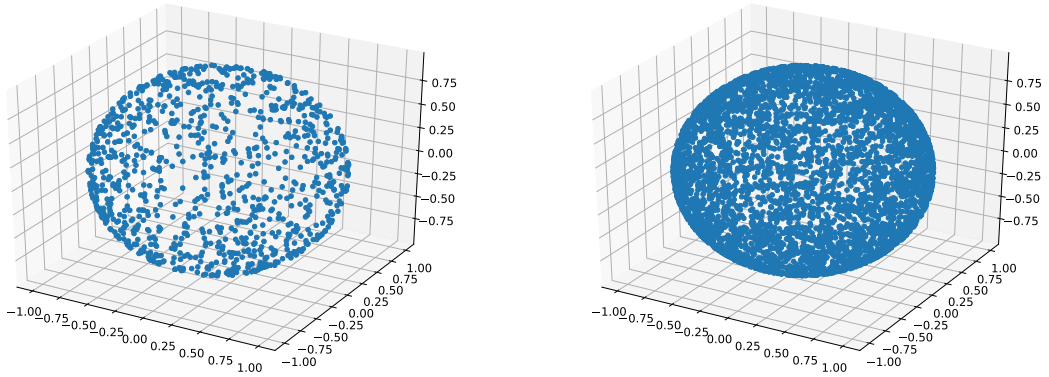
```
1 >>> for i in range(n_trials):
2 ...     costh = 2.0*random.random() - 1.0
3 ...     phi = 2.0*math.pi*random.random()
4 ...     sinth = math.sqrt(1.0 - costh**2)
5 ...     u = sinth*math.cos(phi)
6 ...     v = sinth*math.sin(phi)
7 ...     w = costh
8 ...     x.append(u); y.append(v); z.append(w)
```

角度のサンプリングを繰り返し、7 行目でそれぞれの座標に対する値をコンテナに追加している。あとは得られたデータをプロットするだけである。matplotlib.pyplot モジュールと mpl_toolkits.mplot3d モジュールから Axes3D クラスをインポートしてデータを 3 次元プロットする。

```
1 >>> import matplotlib.pyplot as plt
2 >>> from mpl_toolkits.mplot3d import Axes3D
3 >>> fig = plt.figure()
4 >>> ax = Axes3D(fig)
5 >>> ax.plot(x, y, z, "o", ms=4, mew=0.5)
6 [<mpl_toolkits.mplot3d.art3d.Line3D object at 0x00000000050009E8>]
7 >>> plt.show()
```

3 行目は Figure オブジェクト (プロット用のキャンバス) を fig という名前で生成し、ここに 3 次元プロットの軸を生成し (4 行目)、その軸に対して点をプロットしている (5 行目)。plot のオプションで、"o"は点での表示、ms=は点の大きさ、mew=はマーカの枠線の太さを指定している。

プロットの結果は図 6(a) のようになり、半径 1 の球面状に点が散布しているのが分かる。図は回転させることができるので、回転させてまばらになっていないかどうか確認してほしい。座標点の数が 1,000 の場合は隙間が見られるのでサンプリングが偏っているようにも見えるが、座標点の数を 5,000 まで増やしたときは図 6(b) のようになり、ほぼまんべんなく球面上の点が選ばれているのが分かる。



(a) 座標点数=1,000　　(b) 座標点数=5,000

図 6: サンプリングにより決定した飛行方向

これまで 3 種類の確率密度関数からのサンプリング方法を紹介してきたが、モンテカルロコードで粒子の輸送をシミュレーションする際には、さらに多くのサンプリング法が用いられる。これら確率密度関数は評価済み核データに基づいて決められるが、様々な関数からのサンプリングが必要となり、既に述べたように累積分布関数の逆関数を求めなければならない。累積分布関数の逆関数は上で述べたような簡単に求められるものから、非常に高度な数学的手法を必要とするものまで様々である。これまで開発されてきた累積分布関数の逆関数を求める方法は参考文献 [5] にまとめられており、実用上必要となるほぼすべての確率密度関数からのサンプリングが可能である。累積分布関数の逆関数が求めることができない確率密度関数は、棄却法 (rejection method)[6] によりサンプリングが可能であり、逆関数からの直接サンプリングよりも速い場合もある。

2.3　誤差評価

モンテカルロ法では確率密度関数からのサンプリングにより事象や値を決定し、物理現象を模擬してゆく。確率過程により物理現象を模擬するので、求めたいパラメータは平均値として評価される。それに伴い評価値は平均値の回りにばらつき、そのばらつきは統計誤差として評価される。モンテカルロ法では誤差評価は重要であり、本節では簡単に統計理論の用いられる定義と結果をレビューし、モンテカルロ計算の統計誤差の計算方法について説明する。

2.3.1 期待値

X を確率変数とするとその期待値は次式で定義される。

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx \equiv \bar{x}, \tag{35}$$

ここで、$f(x)$ は確率密度関数で、$\bar{x}$ は確率変数 X の真の平均値とも呼ばれる。もし、$f(x)$ から独立な n 個のサンプルをとり、それらを確率変数 $X_1, X_2, \cdots, X_n$ で表わすと、それぞれの確率変数の期待値は

$$E[X_i] = E[X] = \bar{x} \tag{36}$$

で、(35) 式の真の平均値と等しい。また、これらの確率変数の平均値 $\overline{X} = \sum_{i=1}^{n} X_i/n$ も確率変数とみなすことができ、その期待値は

$$E[\overline{X}] = E\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} E\left[X_i\right] = E[X] = \bar{x} \tag{37}$$

で、$\overline{X}$ は真の平均値 $\bar{x}$ の不偏推定量 (unbiased estimator) になっている。以上の式が示していることは、X_i や $\overline{X}$ の期待値は真の平均値 $\bar{x}$ に等しいが、サンプリングした値は $\bar{x}$ とはならないことを示している。つまり、サンプリングした n 個の平均値は $\bar{x}$ の周りに拡がりを持って分布する。

そこで、必要となるのは拡がりを表わす尺度である。この尺度を導入するためにここでいくつかの式を定義しておく。確率変数 X の実関数 $g(X)$ に対する期待値を次のように定義する。

$$E[g(X)] = \int_{-\infty}^{\infty} g(x) f(x) dx \equiv \bar{g} \tag{38}$$

そのとき、(36) 式に対応して

$$E[g(X_i)] = E[g(X)] = \bar{g} \tag{39}$$

である。

2.4 分散

$\bar{x}$ の周りの $\overline{X}$ の拡がりを評価するために (38) 式の $g(X)$ に $\bar{x}$ の周りの 2 次モーメント

$$g(X) = (X - \bar{x})^2 \tag{40}$$

を代入する。このとき、分散 (variance) は次のように定義される。

$$\sigma^2(X) \equiv E\left[(X - \bar{x})^2\right] = \int_{-\infty}^{\infty} (x - \bar{x})^2 f(x) dx \tag{41}$$

(41) 式は良く知られた次式に書き直すことができる。

$$\sigma^2(X) = \overline{x^2} - \bar{x}^2 \tag{42}$$

この分散もしくは標準偏差 (standard deviation) はしばしば $\hat{x}$ の周りの拡がりを表わす尺度として用いられる。標準偏差は分散の平方根であり、次式で定義される。

$$\sigma(X) = \sqrt{\overline{x^2} - \bar{x}^2} \tag{43}$$

さて、ここで確率変数 $\overline{X}$ の分散を確率変数 X の分散で表わすことを考える。

$$\sigma^2(\overline{X}) = E\left[(\overline{X} - \bar{x})^2\right]. \tag{44}$$

を定義すると、

$$\begin{aligned}
\sigma^2(\overline{X}) &= E\left[\left(\frac{1}{n}\sum_{i=1}^{n} X_i - \bar{x}\right)^2\right] && (45)\\
&= E\left[\left(\frac{1}{n}\sum_{i=1}^{n} (X_i - \bar{x})\right)^2\right] && (46)\\
&= \frac{1}{n}\sigma^2(X) && (47)
\end{aligned}$$

となり、従って、標準偏差は次式で与えられる。

$$\sigma(\overline{X}) = \frac{\sigma(X)}{\sqrt{n}} \tag{48}$$

この式はモンテカルロ計算における統計誤差の重要な性質を表わしている。n 個のサンプルを確率密度関数 $f(x)$ から取ってきて求めた平均の標準偏差は $1/\sqrt{n}$ の割合で減少していくことを示す。

期待値の場合と同様にして、真の分散 $\sigma^2(X)$ は知ることができない。そこで、不偏推定量から真の分散を推定することになる。この不偏推定量は次式

$$U^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}\right) \tag{49}$$

で定義され、その期待値は

$$E\left[U^2\right] = \sigma^2(X) \tag{50}$$

である。実際には分散の不偏推定量は (49) 式からではなく次の式によって計算される。

$$U^2 = \frac{n}{n-1}\left(\overline{X^2} - \overline{X}^2\right) \tag{51}$$

ここで、

$$\overline{X^2} = \frac{1}{n}\sum_{i=1}^{n} X_i^2 \tag{52}$$

である。このとき標本標準偏差 (sample standard deviation) は次のようになる。

$$U = \sqrt{\frac{n}{n-1}\left(\overline{X^2} - \overline{X}^2\right)} \tag{53}$$

通常サンプル数は十分大きいので、分散の推定量と標準偏差の推定量は

$$U^2 \approx \overline{X^2} - \overline{X}^2 \tag{54}$$

$$U \approx \sqrt{\overline{X^2} - \overline{X}^2} \tag{55}$$

としてもよい近似である。本稿ではプログラムをなるべく簡単に記述するため、(54) 式と (55) 式を用いて分散と標準偏差を計算する。

2.5 中心極限定理

前節において分散の評価の仕方について述べた。しかし、分散はただサンプル平均値 $\hat{x} = \sum_{i=1}^{n} x_i/n$ やサンプル x_i の拡がりを示しているに過ぎない。ここでは、分散と推定された平均値の信頼度を結びつける関係式を導く。

定量的な信頼性を評価するために、よく知られている中心極限定理 (central limit theorem) を用いる。この定理によると、どのような組の有限の n 個のサンプルに対しても、サンプル平均値 $\hat{x}$ の分布について記述する確率分布関数 $f_n(x)$ が存在する。また、n が無限大に近づくにつれて $\hat{x}$ には特別な極限分布が存在し、それは次のような正規分布となる。

$$f_n(\hat{x}) \approx \frac{1}{\sqrt{2\pi}\sigma(\overline{X})}\exp\left[\frac{-(\hat{x}-\bar{x})^2}{2\sigma^2(\overline{X})}\right], n \to \infty \tag{56}$$

(56) 式は (48) 式を用いて、次のように書き直すこともできる。

$$f_n(\hat{x}) \approx \sqrt{\frac{n}{2\pi}}\frac{1}{\sigma(X)}\exp\left[\frac{-n(\hat{x}-\bar{x})^2}{2\sigma^2(X)}\right], n \to \infty \tag{57}$$

この式を用いて、$\hat{x}$ が $\bar{x}-\epsilon$ と $\bar{x}+\epsilon$ の間に入る確率を知ることができる。その確率を次式で定義する。

$$P\left\{\bar{x}-\epsilon < \hat{x} < \bar{x}+\epsilon\right\} = \int_{\bar{x}-\epsilon}^{\bar{x}+\epsilon} f_n(\hat{x})d\hat{x} \tag{58}$$

(58) 式に (57) 式を代入し、変数を $\sqrt{2/n}t = (\hat{x} - \bar{x})/\sigma(X)$ と変換すると

$$P\{\bar{x} - \epsilon < \hat{x} < \bar{x} + \epsilon\} = \frac{2}{\sqrt{\pi}} \int_0^{\sqrt{n/2}\cdot\epsilon/\sigma} e^{-t^2} dt = \mathrm{erf}\left[\sqrt{\frac{n}{2}} \frac{\epsilon}{\sigma(X)}\right] \tag{59}$$

となる。ここで、$\mathrm{erf}(x)$ は誤差関数である。

この ϵ に適当な値を入れることにより、$\hat{x}$ の拡がりに対応した信頼性を評価することができる。

$$\epsilon = \sigma(\overline{X}) = \frac{\sigma(X)}{\sqrt{n}} \quad \rightarrow \quad P = 0.683$$

$$\epsilon = 2\sigma(\overline{X}) = 2\frac{\sigma(X)}{\sqrt{n}} \quad \rightarrow \quad P = 0.954$$

$$\epsilon = 3\sigma(\overline{X}) = 3\frac{\sigma(X)}{\sqrt{n}} \quad \rightarrow \quad P = 0.997$$

3 中性子輸送モンテカルロ計算

3.1 固定源問題

前節においてモンテカルロ計算プログラムを作成するためのツールは揃ったので、本節ではそれを用いて実際にプログラムを作成していくことにしよう。中性子輸送の問題は、固定源問題と固有値問題の 2 つのカテゴリに分類することができる。固定源問題は通常、核分裂源を伴わない問題であり、線源は最初から決まっているので、モンテカルロ計算プログラムとして取り扱いやすい。まずは均質 1 領域の球体系で単位強度[c]の点線源が中心に置かれている問題に対してモンテカルロ計算プログラムを作成する。球の半径は 10 cm とし、全断面積は 1.0 cm^{-1} とする ($\Sigma_\mathrm{t} = 1.0$)。吸収はなく散乱のみ起きるとし、散乱については実験室系において等方であると仮定する。この問題に対して実効増倍率は計算できないので、径方向に 100 分割して、それぞれの領域における中性子束を計算する。

この固定源問題に対するモンテカルロ計算アルゴリズムは図 7 のようになる。まず最初に、線源からの中性子の発生位置と飛行方向を決定する。この例題の場合は、点線源であるので発生位置は簡単に指定できる。飛行方向は (29) 式と (30) 式から決定する。飛行方向が決まればその方向に沿った衝突点の決定である。(19) 式もしくは (20) 式から次の衝突点を決定する。もし原点から衝突点までの距離が球の半径よりも大きい場合は、中性子が球から漏れたと判定し、最初に戻って次の中性子を発生させる。もし原点から衝突点までの距離が球の半径よりも小さい場合は、球内で散乱が起きたと判定し、新しい飛行方向を決定し、中性子の位置を更新する。その後、“衝突点の決定” のステップに戻って同じ処理を繰り返す。図 7 で示したアルゴリズムを Python 言語で実装すると以下のようになるだろう。

```
1  import random
2  import math
3  import numpy as np
4
5
6  # ... set calculation conditions.
7  random.seed(1)
8  n_particles = 1000
9  radius = 10.0
10 n_bins = 100
11 bin_width = radius/float(n_bins)
12 SigT = 1.0
13
14 # ... prepare tally counters.
15 n_col = np.zeros(n_bins)    # counter for average
16 n_col2 = np.zeros(n_bins)   # counter for variance
17
18 for i in range(n_particles):
19     x0 = 0.0; y0 = 0.0; z0 = 0.0
20     u = 0.0; v = 0.0; w = 1.0
```

[c] モンテカルロ計算では線源から複数の中性子を放出し、線源粒子 1 つ当りの平均量を計算する。与えられた線源強度が S である場合は、線源強度が S となるように、計算された平均量を規格化すればよい。

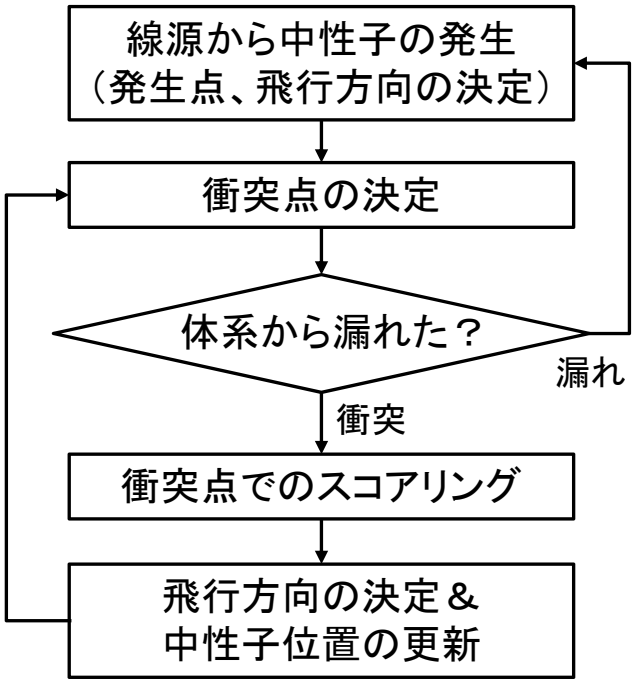


図 7: 固定源問題に対するモンテカルロ計算アルゴリズム

```
21      while True:
22  # ... determine flight distance.
23          dist = -math.log(random.random())/SigT
24  # ... calculate position after flight & airline distance.
25          x1 = x0 + dist*u; y1 = y0 + dist*v; z1 = z0 + dist*w
26  # ... calculate position after flight & airline distance.
27          airl_dist = math.sqrt(x1**2 + y1**2 + z1**2)
28  # ... identify bin id for current position.
29          ibin = int(airl_dist/bin_width)
30          if ibin >= n_bins:
31              break   # ... leaked.
32          n_col[ibin] += 1   # ... collided.
33          n_col2[ibin] += 1
34
35  # ... determine flight direction.
36          costh = 2.0*random.random() - 1.0
37          phi = 2.0*math.pi*random.random()
38          sinth = math.sqrt(1.0 - costh**2)
39          u = sinth*math.cos(phi)
40          v = sinth*math.sin(phi)
41          w = costh
42
43  # ... update position.
44          x0 = x1; y0 = y1; z0 = z1
45
46  # ... calculate bin volumes.
47  vol = [4.*math.pi*((bin_width*float(i+1))**3-(bin_width*float(i))**3)/3.
48         for i in np.arange(n_bins)]
49
50  # ... process statistics.
51  ave = n_col/float(n_particles)
52  ave2 = n_col2/float(n_particles)
53  var = ave2 - ave**2
54  flux_mc = ave/SigT/np.array(vol)
55  flux_mc_stdev = np.sqrt(var/float(n_particles))/SigT/np.array(vol)
56
```

```
57 # ... plot results.
58 import matplotlib.pyplot as plt
59 x = np.linspace(bin_width/2., radius-bin_width/2., n_bins)
60 plt.yscale("log")
61 plt.xlim(0., radius)
62 plt.ylim(0.0001, 100.)
63 plt.errorbar(x, flux_mc, yerr=flux_mc_stdev, fmt="o", markersize=1, capsize=2)
64
65 x_dif = np.linspace(bin_width/2., 10.0, 200)
66 y_dif = 3.*SigT/(4.*math.pi)*(1./x_dif - 1./radius)
67 plt.plot(x_dif, y_dif, linewidth=3)
68
69 plt.show()
```

このプログラムで用いられているコードは、ほとんど前節で説明したものであり、それらについては前節を参照していただきたい。追加で説明が必要なのは、中性子束を評価するためのスコアリング (中性子束のタリー) と誤差評価である。中性子束を評価するために 15 行目で `n_col` というカウンタ (`numpy` の 1 次元配列) を定義している。これは衝突エスティメータによって中性子束を評価するために衝突回数をカウントするための配列である。あるビンにおける衝突率を R_{tot} とすると

$$R_{\mathrm{tot}} = \Sigma_{\mathrm{t}} \phi V \tag{60}$$

で与えられる。ここで、Σ_{t} はビンにおける全断面積、ϕ はビンにおける中性子束、V はビンの体積である。この式から中性子束は

$$\phi = \frac{R_{\mathrm{tot}}}{\Sigma_{\mathrm{t}} V} \tag{61}$$

として計算できる。衝突率は、単位時間当りにビンの領域で起きる衝突回数である。今は単位強度の線源を考えているので、各ビン毎に衝突数をカウントし、全線源粒子数で割れば平均の衝突回数を計算でき、中性子束も計算できることになる。衝突回数のスコアリングは 32 行目で行われ、全線源粒子の追跡が終了した後、51 行目で平均値 (衝突確率) が計算される。54 行目で平均値を全断面積とビンの体積で割って中性子束を計算している。

統計誤差を評価するための手順も衝突回数の平均値を計算するものとほぼ同じである。スコアの 2 乗平均値を計算すればよい。16 行目で `n_col2` というカウンタを用意し、33 行目で 2 乗値をスコアリングしている。53 行目は (54) 式による分散の推定値であり、これをもとに 55 行目において (48) 式から平均値の標準偏差を計算している。

モンテカルロ計算の結果は、59 行目から 63 行目でプロットされる。モンテカルロ計算の結果を誤差付きでプロットするために、`pyplot` モジュールの `plot` 関数ではなく、`errorbar` 関数を用いている。`fmt=''o''` は点でのプロットを指定し、`markersize=1` は点の大きさ、`capsize=2` は誤差棒の上下につける横線の長さを指定している。

この問題に対する拡散理論による解析解は

$$\phi(r) = \frac{3\Sigma_{\mathrm{t}} S}{4\pi} \left[\frac{1}{r} - \frac{1}{R} \right] \tag{62}$$

で与えられる [7, p.73]。ここで、S は中性子の線源強度、R は球の半径である。モンテカルロ計算の結果と拡散理論の解析解を比較するために、(62) 式の関数を 55 行目から 56 行目に定義し、68 行目から 70 行目でプロットしている。

このスクリプトを `mc_sphere_source.py` という名前で保存し、非対話モードで (スクリプトファイルを Python に渡して) 実行してみよう。

```
> python mc_sphere_source.py
```

すると、図 8 に示すような結果が表示される。モンテカルロ計算の結果と拡散理論の解析値を比較すると、線源近傍と球の外周付近に以外ではほぼ一致する結果が得られる。これらの差異は、輸送理論と拡散理論の違いによるものである。モンテカルロ計算による中性子束分布の結果は輸送理論に基づいて計算した結果に等しく、

モンテカルロ計算の方が正しい結果を与えている。球の外周付近で拡散理論の精度を上げようと思えば、(62)式で外挿距離を考慮すればよい。また、線源近傍おける中性子束は

$$\phi(r) \approx \frac{S}{4\pi r^2} \tag{63}$$

で精度よく近似できるので、線源近傍付近では (63) 式と比較するとよい。簡単であるので、読者自身でプログラムを修正し、これらの修正した結果をプロットし、モンテカルロ計算の結果と一致するかどうか確認してみるとよい。図 8 の横軸は“中心からの距離 [cm]”で、縦軸は“中性子束 $[1/\mathrm{cm}^2/\mathrm{s}/(1\text{ 線源粒子})]$”である。軸ラベルや判例の追加は読者への課題としておく。

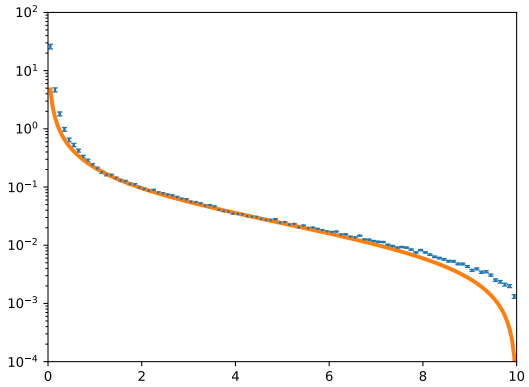


図 8: 中心に点線源を配置した球体系における中性子束分布

3.2 固有値計算

前項で固定源問題に対する中性子輸送モンテカルロ計算の仕組みが理解できたと思うので、次は核計算で最も重要な量である実効増倍率 (固有値) をモンテカルロ計算で計算してみよう。固有値問題では外部線源はなく、中性子源は核分裂源であり、核分裂源分布はあらかじめ分かっていない。そのため、最初は適当に仮定した線源からスタートし、核分裂点を決定して再びそこから新しい中性子を発生させるといった操作が必要となる。このような計算を行うため、「世代」という概念を導入し、あるまとまった数の中性子に対してランダムウォークを行い、その過程で発生する核分裂点と核分裂中性子数を保存する。次の世代では、前の世代で起きた核分裂点から中性子を発生させ、ランダムウォークを行い、同様に核分裂点と核分裂中性子数を保存する。この処理を十分な世代数について繰り返し、実効増倍率をはじめとする核特性量を計算する。このような反復法はべき乗法 (power iteration) としてよく知られているものであり、決定論的手法の源反復もしくは外部反復法と同様の手法である。

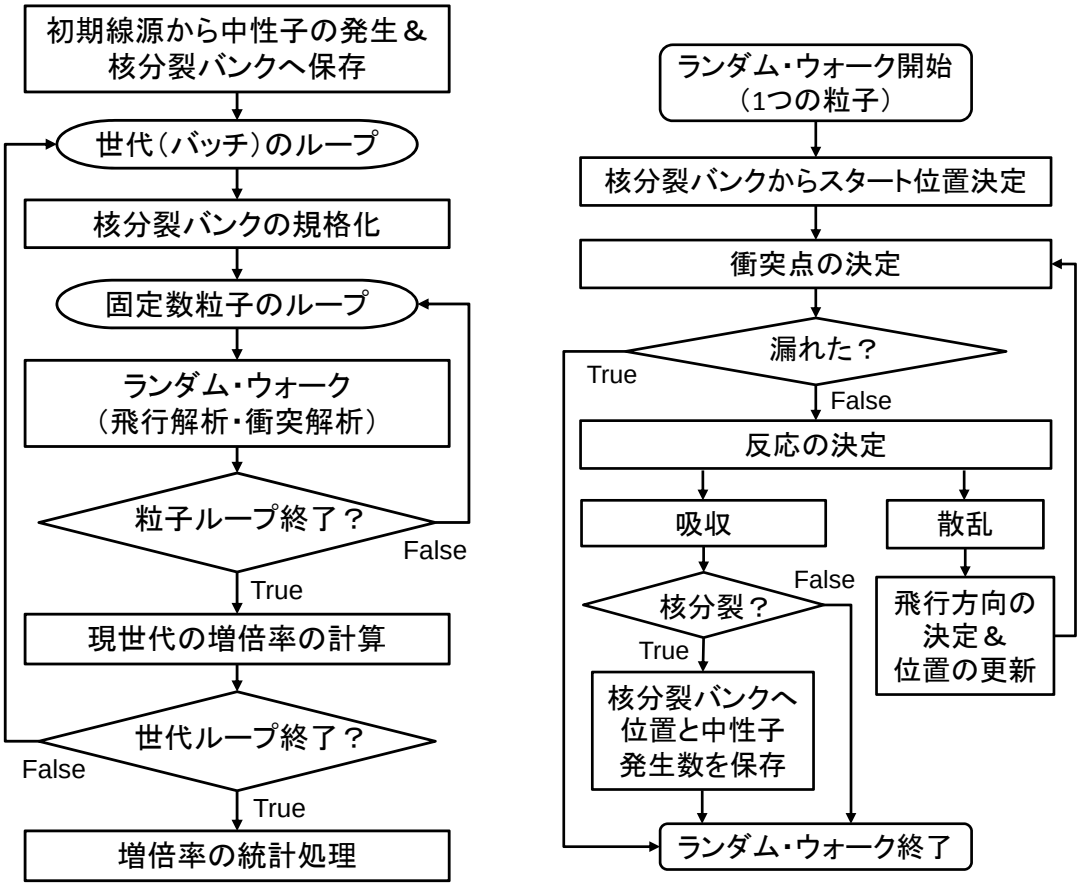
世代反復法を用いた固有値計算のアルゴリズムを図 9 に示す。世代反復法では、ある決まった数の粒子についてランダムウォークを行い、これを複数の世代について繰り返すことが基本的な計算の流れである。図 9(a) において、ある決まった数の粒子のランダムウォークは「固定数粒子のループ」に対応している。その前の「核分裂バンクの規格化」とその後の「現世代の実効増倍率の計算」を含めた一連の処理は「バッチ」とも呼ばれ[d]、1 世代=1 バッチとしてループ処理をして複数バッチについて計算を行う。

世代反復法では、核分裂点における粒子の情報を保存しておくために「核分裂バンク」と呼ばれるデータのコンテナを用意しておく必要がある。「初期線源から中性子の発生」では、適当に仮定した中性子線源分布から発生位置を核分裂バンクへ保存する。この処理は一度だけでよい。後はランダムウォークの過程で新しい核分裂位置に更新される。「世代のループ」に入ると、「核分裂バンクの規格化」を行う。核分裂バンクには、入力で指定した 1 世代当りに発生させる粒子数と同数の核分裂位置の情報が入っているとは限らない。多い場合にはその中から適切に抽出し、少ない場合には核分裂位置の情報を複製する。

[d] ここでは 1 世代分の処理をバッチと呼んだが、コードにより呼び方が異なる。MVP コードでは「バッチ」、MCNP コードでは「サイクル」、MONK コードでは「ステージ」と呼ばれる。

1 バッチの処理において、入力で指定した 1 世代当りに発生させる粒子数すべてに対するランダムウォークが終了 (「固定数粒子のループ」が終了) すれば、「現世代の増倍率の計算」ができる。増倍率 k は、(現世代で発生した核分裂中性子の数)/(発生させた線源粒子の数) で計算ができる。第 i 世代の増倍率を k_i とすると、各世代 (バッチ) の増倍率の結果 $k_1, k_2, \cdots$ が得られる。これをサンプルとして「増倍率の統計処理」を行い、最終的な実効増倍率とする。ただし、最初の複数バッチを統計処理から除く必要がある。最初の核分裂源分布は適当に仮定したものであるので、核分裂源分布が収束したとみなせるようになってからの増倍率の結果だけを用いて統計処理をしなければならない。

図 9(b) は各粒子に対するランダムウォークの計算の流れを示している。図 7 と基本的には変わらない。吸収 (核分裂と捕獲) 反応が考慮するため、「反応の決定」のサンプリング処理が追加されている。また、核分裂反応が起きたときに「核分裂バンクへ位置と中性子発生数を保存」する処理も追加されている。



(a) 世代反復の計算の流れ　　(b) ランダムウォークの計算の流れ

図 9: 固有値計算のアルゴリズム

図 9 で説明したアルゴリズムを Python 言語で実装すると以下のようになるだろう。計算の体系は 1 領域裸の球体系であり、半径 8 cm である。物質に対する断面積は「Python を利用した核計算 (1) — 決定論的手法 —」と同じ巨視的断面積を用いている。すなわち、$\nu\Sigma_{\mathrm{f}} = 0.6, \Sigma_{\mathrm{a}} = \Sigma_{\mathrm{c}} + \Sigma_{\mathrm{f}} = 0.5, \Sigma_{\mathrm{s}} = 0.5$ である。決定論的手法の計算では、1 回の核分裂あたりに放出される核分裂中性子数である ν 値は単独で設定されていないので、ここでは 2.5 と設定した。1 バッチあたりの粒子数 (ヒストリー数) は 1,000、統計処理から除くバッチ数 (スキップバッチ数) は 20、スキップバッチ数を含む全バッチ数は 120 で計算を行う設定となっている。

```
1 import numpy as np
2 import math
3 import random
4 import itertools
5 from scipy.sparse import lil_matrix
6
7
```

```
# ... set calculation conditions.
random.seed(1)
n_batches = 120
n_particles = 1000
n_skip = 20

# ... set sphere radius.
radius = 8.0

# ... set group constants.
NG   = 1    # number of energy groups
NMat = 1    # number of materials.
SigT = np.zeros((NMat, NG)) # total xsec
SigA = np.zeros((NMat, NG)) # absorption xsec
SigP = np.zeros((NMat, NG)) # production xsec
SigF = np.zeros((NMat, NG)) # fission xsec
SigS = [ lil_matrix((NG, NG))  for i in range(NMat) ] # scattering xsec
Chi  = np.zeros((NMat, NG)) # fission spectrum
D    = np.zeros((NMat, NG)) # diffusion coef
SigA[0,:] = [0.5]
SigP[0,:] = [0.6]
NuTot = np.array([[2.5]])
SigF = SigP/NuTot
Chi[0,:] = [1.0]
SigS[0][0,0] = 0.5
SigT[0,:] = SigS[0].toarray().sum(axis=1) + SigA[0]
D[0,:] = 1.0/(3.0*SigT)

# ... prepare probabilities.
prob_fis = SigF/SigA
prob_abs = SigA/SigT

# ... prepare cummulative probabilities.
cum_prob_chi = np.tril(np.tile(Chi, (Chi.size,1))).sum(axis=1)

# ... prepare containers for bank.
bank = []        # unnormalized bank
bank_init = []   # normalized bank (initial bank to start each batch)

# ... prepare containers for tally.
tally_k = []

# ... prepare initial source.
for i in range(n_particles):
    rrr = radius*math.pow(random.random(), 1.0/3.0)
    costh = 2.0*random.random() - 1.0
    phi = 2.0*math.pi*random.random()
    sinth = math.sqrt(1.0 - costh**2)
    x0 = rrr*sinth*math.cos(phi)
    y0 = rrr*sinth*math.sin(phi)
    z0 = rrr*costh
    nu_value_init = 1.0
    bank_init.append(np.array([x0, y0, z0, nu_value_init]))

# ... start loop for batch.
for j in range(n_batches):
    print("batch␣{:>4}".format(j), end="")
    counter_k = np.zeros(2) # ... [score sum, square score sum]

# ... normalize fission source.
    if j != 0:
        for p_info in bank:
            while p_info[3] >= 1.0:
                bank_init.append(np.array(
                    [p_info[0], p_info[1], p_info[2], 1.0]))
                p_info[3] -= 1.0
```

```
74              if p_info[3] >= random.random():
75                  bank_init.append(np.array(
76                      [p_info[0], p_info[1], p_info[2], 1.0]))
77
78          bank.clear()
79
80          if not bank_init:
81              print("XXX␣No␣fission␣neutrons.")
82              exit()
83          for v in itertools.cycle(bank_init):
84              bank_init.append(v)
85              if len(bank_init) >= n_particles: break
86
87  # ... start each particle.
88      for i in range(n_particles):
89          x0 = bank_init[i][0]
90          y0 = bank_init[i][1]
91          z0 = bank_init[i][2]
92
93  # ... determine initial flight direction.
94          costh = 2.0*random.random() - 1.0
95          phi = 2.0*math.pi*random.random()
96          sinth = math.sqrt(1.0 - costh**2)
97          u = sinth*math.cos(phi)
98          v = sinth*math.sin(phi)
99          w = costh
100
101 # ... determine initial energy group.
102         grp = 0
103
104 # ... loop for collisions.
105         while True:
106 # ... determine flight distance.
107             xstot = SigT[0][grp]
108             dist = -math.log(random.random())/xstot
109 # ... calculate position after flight & airline distance.
110             x1 = x0 + dist*u; y1 = y0 + dist*v; z1 = z0 + dist*w
111             airl_dist = math.sqrt(x1**2 + y1**2 + z1**2)
112
113 # ... check whether leaks or not.
114             if airl_dist > radius:
115                 break  # ... leaked.
116
117 # ... analyze collision.
118             if random.random() < prob_abs[0][grp]:
119                 if random.random() < prob_fis[0][grp]:
120 # ... tally k-value.
121                     particle_info = np.array([x1, y1, z1, NuTot[0][grp]])
122                     bank.append(particle_info)
123                 break  # ... absorbed.
124
125 # ... determine flight direction.
126             costh = 2.0*random.random() - 1.0
127             phi = 2.0*math.pi*random.random()
128             sinth = math.sqrt(1.0 - costh**2)
129             u = sinth*math.cos(phi)
130             v = sinth*math.sin(phi)
131             w = costh
132
133 # ... update position.
134             x0 = x1; y0 = y1; z0 = z1
135
136 # ... end of loop for particle.
137
138 # ... tally k-score.
139     sum_fis_neu = 0.0
```

```
140     for p_info in bank:
141         sum_fis_neu += p_info[3]
142     counter_k[0] = sum_fis_neu/float(n_particles)
143     counter_k[1] = counter_k[0]**2
144     print("␣␣k␣=␣{0:.5f}".format(counter_k[0]))
145     tally_k.append(counter_k)
146 
147 # ... clear fission bank.
148     bank_init.clear()
149 
150 # ... end of loop for batch.
151 
152 # ... process statistics.
153 del tally_k[:n_skip]
154 n_active = float(n_batches) - float(n_skip)
155 k_average, k2_average = sum(tally_k)/n_active
156 k_variance = (k2_average - k_average**2)/n_active
157 k_stdev = math.sqrt(k_variance)
158 k_fsd = k_stdev/k_average
159 
160 # ... output results.
161 print("*****␣final␣results␣*****")
162 n_total = n_particles*n_batches
163 print("Number␣of␣total␣histories␣:␣", n_total)
164 print("k␣value␣␣␣␣␣␣␣␣␣␣␣␣␣␣:␣{:.5f}".format(k_average))
165 print("standard␣deviation␣␣␣␣:␣{:.5f}".format(k_stdev))
166 print("fractional␣std.␣dev.␣:␣{:.5f}".format(k_fsd))
```

上記のプログラムもほとんどこれまで説明済みのコードが再利用されており、固有値計算のアルゴリズムの説明とプログラム内のコメント文を照らし合わせれば、何をしているのかを容易に理解できると思う。固有値計算で新たに追加したコードについて説明しておく。44 行目と 45 行目は核分裂バンクを空のリストとして定義している。bank はランダムウォークの途中で発生した核分裂の情報を保存しておくバンクで、bank_init は、粒子の追跡をスタートするときの情報を取り出す核分裂バンクである。bank のデータが規格化されて、bank_init に保存される。

48 行目の tally_k は各バッチで計算された増倍率 k_i を保存しておくコンテナである。空のリストとして定義し、すべての k_i を保存し、世代ループがすべて終了するとこのデータを用いて統計処理を行う。

52 行目から 54 行目では、球内の一様分布から 1 点をサンプリングにより決定している。極座標の 1 点 (r, θ, ϕ) をする。(θ, ϕ) については、飛行方向のサンプリングと同様に決定することができる。径方向の位置 r については、以下の確率密度関数からサンプリングを行う。

$$p(r) = \left(\frac{3}{4\pi R^3}\right) 4\pi r^2 dr \quad \text{for} \quad 0 \leq r < R \tag{64}$$

半径 R の球の体積 $4\pi R^3/3$ で、原点から距離 r 離れた微小球殻の体積が $4\pi r^2 dr$ であることから計算される。これを累積分布関数に変換し、逆関数を求めれば、以下の式でサンプリングできることが分かる。

$$r = R\sqrt[3]{\xi} \tag{65}$$

52 行目は (65) 式を実装している。

65 行目の counter_k は、各バッチで計算された k_i と k_i^2 を numpy の 1 次元配列で保存するためのコンテナである。array([k_1, k_1^2]), array([k_2, k_2^2]), $\cdots$ として各バッチ毎に作成された counter_k オブジェクトを tally_k に追加していく。

68 行目から 85 行目は核分裂バンクの規格化 (核分裂源分布の規格化) を行っている。核分裂源分布の規格化については様々な方法があるが、ここでは参考文献 [7, p.230] の例題 8.1 で用いられている手法を採用している。69 行目から 76 行目のループで、bank の中身を p_info として取り出し、核分裂発生数 p_info[3] が 1.0 となるようにバンクデータを複製し、bank_init へ保存する。この例題では p_info[3]$= \nu = 2.5$ であるので、p_info[3]≥ 1 の間は、複製した分だけ p_info を 1 ずつ減らしていく。p_info[3]< 1 となれば、乱数を 1 つ生成し、端数の値と比較して、複製するかどうかを確率的に決定する。78 行目は、すべての核分裂バンクの情報に対して規格化が終了すれば bank の情報は必要ないので、新たに核分裂バンクの情報を保存するためにすべ

ての情報を消去している。80 行目から 82 行目は核分裂バンクが 0 になったときの例外処理である。83 行目から 85 行目は、核分裂バンクに n_particles 個分のデータがない場合の処理を行っている。itertools モジュールの cycle 関数を用い、bank_init に保存されているデータを無限に繰り返し追加していく。n_particles 個以上のデータが追加されると終了する。

102 行目の grp は中性子のエネルギー群数を表している。1 群の問題では常に 0(1 群のインデックス) であり、値が変わることはないが、多群データを格納できる断面積データのコンテナからデータを取り出すために指定している。

105 行目から 134 行目がランダムウォークの処理である。114 行目で体系から漏れたと判断された場合と 118 行目で吸収されたと判断された場合にループを抜ける。

139 行目から 145 行目で現在のバッチのおける増倍率 k_i と k_i^2 を計算し、コンテナに保存している。140 行目のループで bank コンテナから核分裂中性子数 p_info[3] を取り出し、和をとることによって現在のバッチの総核分裂中性子数を計算している。142 行目と 143 行目で counter_k に k_i と k_i^2 を保存し、145 行目でそれらを 1 セットとして tally_k に保存する。

153 行目から 158 行は、各バッチ計算された増倍率データの統計処理を行っている。153 行目で前から n_skip 個分のデータを削除している。155 行目は増倍率の平均値と 2 乗平均値を同時に計算している。tally_k の要素を numpy の配列として定義しているので、このようにまとめて記述することができる。158 行目の k_fsd は相対標準偏差を計算している。単に、標準偏差を平均値で割ったものであり、100 倍すればパーセンテージでの表記となる。

上記のスクリプトを mc_sphere_1g.py という名前で保存し、非対話モードで実行してみよう。

```
> python mc_sphere_1g.py
```

計算結果は表 1 の Python の行で示されている値となる。プログラムが正しく実装されているかどうかを確かめるために、多群モンテカルロコード GMVP[1] の結果も表に示している。参照解を与えるために、GMVP の計算は 1 バッチあたり 10,000 ヒストリー、全バッチ数 1,000、スキップバッチ数 100 で計算を行った。GMVP の実効増倍率の結果は、4 種類のエスティメータの複合エスティメータから計算されたものである。

表 1: 1 領域裸の球体系 1 群問題に対する実効増倍率の計算結果

コード	実効増倍率	標準偏差	相対標準偏差 (%1σ)
Python	1.10185	0.00391	0.355
GMVP	1.10662	0.00011	0.001

Python プログラムで計算した結果は参照解と比較して、2 標準偏差程度の範囲内で一致しており、正しく実装できていると思われる。ここで示した Python プログラムの結果はヒストリー数が十分ではない。また、スキップバッチ数も 20 と小さめである。読者には、1 ヒストリーあたりのヒストリ数、バッチ数、スキップバッチ数を変更し、GMVP の計算結果に近づくかどうか確かめてもらいたい。特にスキップバッチ数は 20 では不十分だと思われる。ここで示した Python プログラムは計算速度が遅いので、ヒストリー数を大きくしすぎると計算時間がかかってしまうので注意されたい。

次に、上で示した 1 群問題に対する固有値計算プログラムを 2 群問題に対するプログラムへ拡張してみよう。1 群の固有値計算プログラムで世代反復法のアルゴリズムは実装できているので、基本的には、核分裂中性子のエネルギー群数を決める処理と、散乱後のエネルギー群を決める処理を追加すればよい。あと、それらの処理で必要となる確率データを準備しなければならない。これらの処理を追加した、2 群の固有値計算プログラムを以下に示す。

```
1 import numpy as np
2 import math
3 import random
4 import itertools
5 from scipy.sparse import lil_matrix
6
```

```
7
8  # ... set calculation conditions.
9  random.seed(1)
10 n_batches = 120
11 n_particles = 2000
12 n_skip = 20
13
14 # ... set sphere radius.
15 radius = 15.6
16
17 # ... set group constants.
18 NG   = 2    # number of energy groups
19 NMat = 1    # number of materials.
20 SigT = np.zeros((NMat, NG)) # total xsec
21 SigA = np.zeros((NMat, NG)) # absorption xsec
22 SigP = np.zeros((NMat, NG)) # production xsec
23 SigF = np.zeros((NMat, NG)) # fission xsec
24 SigS = [ lil_matrix((NG, NG))  for i in range(NMat) ] # scattering xsec
25 Chi  = np.zeros((NMat, NG)) # fission spectrum
26 D    = np.zeros((NMat, NG)) # diffusion coef
27 SigA[0,:] = [0.00286,  0.0850]
28 SigP[0,:] = [0.0034189, 0.149 ]
29 NuTot = np.array([[2.5, 2.5]])
30 SigF = SigP/NuTot
31 SigS[0][0,0] = 0.1924
32 SigS[0][0,1] = 0.0212
33 SigS[0][1,1] = 1.32145
34 D[0,:] = [1.54, 0.237]
35 Chi[0,:] = [1.0, 0.0]
36 SigT[0,:] = SigS[0].toarray().sum(axis=1) + SigA[0]
37
38 # ... prepare probabilities.
39 prob_fis = SigF/SigA
40 prob_abs = SigA/SigT
41
42 # ... prepare cummulative probabilities.
43 sig_gg = SigS[0].toarray()
44 sig_g = sig_gg.sum(axis=1)
45 prob_gg = sig_gg/sig_g.repeat(sig_g.size).reshape(sig_g.size, sig_g.size)
46 cum_prob_gg = np.array([np.tril(np.tile(v, (len(prob_gg) ,1))).sum(axis=1) for v in prob_gg])
47 cum_prob_chi = np.tril(np.tile(Chi, (Chi.size,1))).sum(axis=1)
48
49 # ... prepare containers for bank.
50 bank = []        # unnormalized bank
51 bank_init = []   # normalized bank (initial bank to start each batch)
52
53 # ... prepare containers for tally.
54 tally_k = []
55
56 # ... prepare initial source.
57 for i in range(n_particles):
58     rrr = radius*math.pow(random.random(), 1.0/3.0)
59     costh = 2.0*random.random() - 1.0
60     phi = 2.0*math.pi*random.random()
61     sinth = math.sqrt(1.0 - costh**2)
62     x0 = rrr*sinth*math.cos(phi)
63     y0 = rrr*sinth*math.sin(phi)
64     z0 = rrr*costh
65     nu_value_init = 1.0
66     bank_init.append(np.array([x0, y0, z0, nu_value_init]))
67
68 # ... start loop for batch.
69 for j in range(n_batches):
70     print("batch␣{:>4}".format(j), end="")
71     counter_k = np.zeros(2) # ... [score sum, square score sum]
72
```

```
# ... normalize fission source.
    if j != 0:
        for p_info in bank:
            while p_info[3] >= 1.0:
                bank_init.append(np.array(
                    [p_info[0], p_info[1], p_info[2], 1.0]))
                p_info[3] -= 1.0
            if p_info[3] >= random.random():
                bank_init.append(np.array(
                    [p_info[0], p_info[1], p_info[2], 1.0]))

        bank.clear()

        if not bank_init:
            print("XXX␣No␣fission␣neutrons.")
            exit()
        for v in itertools.cycle(bank_init):
            bank_init.append(v)
            if len(bank_init) >= n_particles: break

# ... start each particle.
    for i in range(n_particles):
        x0 = bank_init[i][0]
        y0 = bank_init[i][1]
        z0 = bank_init[i][2]

# ... determine initial flight direction.
        costh = 2.0*random.random() - 1.0
        phi = 2.0*math.pi*random.random()
        sinth = math.sqrt(1.0 - costh**2)
        u = sinth*math.cos(phi)
        v = sinth*math.sin(phi)
        w = costh

# ... determine initial energy group.
        rn = random.random()
        for g, val in enumerate(cum_prob_chi):
            if rn < val:
                grp = g
                break

# ... loop for collisions.
        while True:
# ... determine flight distance.
            xstot = SigT[0][grp]
            dist = -math.log(random.random())/xstot
# ... calculate position after flight & airline distance.
            x1 = x0 + dist*u; y1 = y0 + dist*v; z1 = z0 + dist*w
            airl_dist = math.sqrt(x1**2 + y1**2 + z1**2)

# ... check whether leaks or not.
            if airl_dist > radius:
                break  # ... leaked.

# ... analyze collision.
            if random.random() < prob_abs[0][grp]:
                if random.random() < prob_fis[0][grp]:
# ... tally k-value.
                    particle_info = np.array([x1, y1, z1, NuTot[0][grp]])
                    bank.append(particle_info)
                break  # ... absorbed.

# ... find new energy group.
            rn = random.random()
            for g, val in enumerate(cum_prob_gg[grp]):
                if rn < val:
```

```
                    grp_new = g
                    break
            grp = grp_new

# ... determine flight direction.
            costh = 2.0*random.random() - 1.0
            phi = 2.0*math.pi*random.random()
            sinth = math.sqrt(1.0 - costh**2)
            u = sinth*math.cos(phi)
            v = sinth*math.sin(phi)
            w = costh

# ... update position.
            x0 = x1; y0 = y1; z0 = z1

# ... end of loop for particle.

# ... tally k-score.
    sum_fis_neu = 0.0
    for p_info in bank:
        sum_fis_neu += p_info[3]
    counter_k[0] = sum_fis_neu/float(n_particles)
    counter_k[1] = counter_k[0]**2
    print("  k = {0:.5f}".format(counter_k[0]))
    tally_k.append(counter_k)

# ... clear fission bank.
    bank_init.clear()

# ... end of loop for batch.

# ... process statistics.
del tally_k[:n_skip]
n_active = float(n_batches) - float(n_skip)
k_average, k2_average = sum(tally_k)/n_active
k_variance = (k2_average - k_average**2)/n_active
k_stdev = math.sqrt(k_variance)
k_fsd = k_stdev/k_average

# ... output results.
print("***** final results *****")
n_total = n_particles*n_batches
print("Number of total histories : ", n_total)
print("k value                   : {:.5f}".format(k_average))
print("standard deviation        : {:.5f}".format(k_stdev))
print("fractional std. dev.      : {:.5f}".format(k_fsd))
```

このプログラムで取り扱っている問題は、「Python を利用した核計算 (1) — 決定論的手法 —」の 1 領域裸の球体系 2 群問題である。体系の半径は半径 15.6 cm であり、物質に対する巨視的断面積はこの問題と同じ値を用いている。表 2 に巨視的断面積の値を示す。決定論的手法の計算では、1 回の核分裂あたりに放出される核分裂中性子数である ν 値は各群とも 2.5 と設定した。1 バッチあたりの粒子数 (ヒストリー数) は 1,000、統計処理から除くバッチ数 (スキップバッチ数) は 20、スキップバッチ数を含む全バッチ数は 120 で計算を行う設定である。

表 2: 2 群固有値問題に対する巨視的断面積データ

エネルギー群	1 群	2 群
生成断面積 $\nu\Sigma_{\mathrm{f},g}$	0.0034189	0.149
吸収断面積 $\Sigma_{\mathrm{a},g}$	0.00286	0.085
核分裂スペクトル χ_g	1.0	0.0

散乱後の群数 (g')	1 群	2 群
散乱断面積 $\Sigma_{\mathrm{s},1\to g'}$	0.1924	0.0212
散乱断面積 $\Sigma_{\mathrm{s},2\to g'}$	0.0	1.32145

43 行目から 45 行目において、g 群から g' 群へ散乱する確率を計算する。43 行目は $\Sigma_{\mathrm{s},g\to g'}$ を sig_gg 配列へ格納し、44 行目で各行に対する和をとる。つまり、$\Sigma_{\mathrm{s},g} = \sum_{g'} \Sigma_{\mathrm{s},g\to g'}$ を計算する。45 行目で $\Sigma_{\mathrm{s},g\to g'}/\Sigma_{\mathrm{s},g}$ を計算し、g 群から g' 群へ散乱する確率を prob_gg に格納する。46 行目は前に示したのと同じ方法で、累積確率へ変換する。

47 行目は核分裂に対する累積確率を計算し、cum_prob_chi へ格納する。Chi はすでに和が 1.0 であるので、そのままで確率となっている。これも前に示したのと同じ方法で累積確率へ変換する。

108 行目から 112 行目は核分裂中性子のエネルギー群のサンプリングを行っている。乱数を 1 つ発生させて、核分裂スペクトルの節点と順番に比較していき、節点が乱数より大きくなればその時点でのインデックスを中性子エネルギーのインデックスとして決定する。この 2 群問題では、$\chi_1 = 1.0, \chi_2 = 0.0$ であるので、grp = 0 としておけばよいが、ここでは一般的に利用できるように実装している。

117 行目の xstot は飛行距離を計算するときの全断面積の値である。SigT 配列からインデックス grp のデータを参照し、エネルギー群に応じて全断面積の値を変更するようにしている。

136 行目から 141 行目は散乱反応を起こすことが決定した後、散乱後のエネルギー群数を決める処理である。grp 群の累積確率データを用いて、発生させた乱数 rn とその累積確率データの節点を比べて、散乱後のエネルギー群のインデックスを決定している。新しく決定した散乱後のエネルギー群は grp_new として保存し、探索が終了したのちに grp を grp_new で更新している。

上記のスクリプトを mc_sphere_2g.py という名前で保存し、非対話モードで実行してみよう。

```
> python mc_sphere_2g.py
```

計算結果を表 3 に示す。GMVP コードの結果も表に示している。参照解を与えるために、1 バッチあたり 10,000 ヒストリー、全バッチ数 1,000、スキップバッチ数 100 で行った GMVP コードの結果も表に示す。GMVP の実効増倍率の結果は、4 種類のエスティメータの複合エスティメータから計算されたものである。

表 3: 1 領域裸の球体系 2 群問題に対する実効増倍率の計算結果

コード	実効増倍率	標準偏差	相対標準偏差 (%1σ)
Python	0.58566	0.00287	0.490
GMVP	0.58208	0.00024	0.042

Python プログラムで計算した結果は参照解と比較して、2 標準偏差の範囲内で一致しており (ほぼ 1 標準偏差ぐらいの差異)、正しく実装できていると思われる。2 群計算の方もヒストリー数やスキップバッチ数が小さいので、読者の方でパラメータを変更して計算し、GMVP の結果に近づくかどうか確認していただきたい。

4 むすびに

粒子輸送モンテカルロ計算の基礎手法 (乱数の発生方法とサンプリング方法) について解説し、Python 言語を用いてこれらの方法を実装したサンプルコードを示した。また、これらのサンプルコードを再利用して、固定源問題を解くプログラム、1 群と 2 群固有値問題を解くプログラムを示した。読者はこれらのサンプルコードを対話モードで実行し確認したり、サンプルプログラムを実際に自分で実行したりすれば、モンテカルロ計算の仕組みがよく理解できるだろう。

本文では言及しなかったが、ここで示したモンテカルロ計算はアナログ・モンテカルロ法に基づいている。ここで示したプログラムを非アナログ・モンテカルロ法へ拡張することはそれほど難しくはない。読者がもし拡張するのであれば、山本俊弘氏が作成した過去の炉物理夏期セミナーの資料 [8] が参考になるだろう。

また、ここで示したプログラムは裸の球体系に対する問題しか解くことができない。「Python を利用した核計算 (1) — 決定論的手法 —」で示された 1 次元平板状体系、1 次元無限円筒体系へと拡張するのもよい練習問題となるだろう。さらに反射体付きの 2 領域体系へと拡張しても面白いだろう。

本稿で示した Python スクリプトは、読者がモンテカルロ計算手法を容易に理解できるようにするため、クラスを用いずに 1 つのファイルにバッチ処理する形式で記述した。断面積データなどは、「Python を利用した

核計算 (1) — 決定論的手法 —」と共有化して再利用した方が効率的である。また、モンテカルロ計算ルーチンをそのままクラス化して、複数回のモンテカルロ計算ができるようにしておけば、臨界サーチの計算などもできるようになるだろう。

読者がここで学んだモンテカルロ・プログラムを足がかりに、さらに高度なモンテカルロ計算手法やモンテカルロコード開発に興味をもってもらえれば幸いである。

参考文献

[1] Y. Nagaya, K. Okumura, T. Sakurai and T. Mori, "MVP/GMVP version 3; General purpose Monte Carlo codes for neutron and photon transport calculations based on continuous energy and multigroup methods," JAEA-Data/Code 2016-018 (2017).

[2] X-5 Monte Carlo Team, "MCNP – A General Monte Carlo N-Particle Transport Code, Version 5," LA-UR-03-1987 (2003).

[3] D. H. Lehmer, "Mathematical methods in large-scale computing units," Proceedings of the Second Symposium on Large Scale Digital Computing Machinery, Harvard University Press, Cambridge, Massachusetts, pp.141-146 (1951).

[4] `https://docs.python.jp/3/library/random.html`

[5] C. J. Everett and E. D. Cashwell, "A Third Monte Carlo Sampler," LA-9721-MS (1983).

[6] L. L. Carter and E. D. Cashwell, "Particle Transport Simulation with the Monte Carlo Method," TID-26607 (1975).

[7] S.A. Dupree and S.K. Fraley, "A Monte Carlo Primer," Kluwer Academic/Plemium Publishers (2002).

[8] 山本 俊弘, "Boltzmann 方程式の解法 — 確率論的手法による解法 —," 第 34 回炉物理夏期セミナーテキスト (2002).