モンテカルロ法

モンテカルロ法というと、乱数を用いたアルゴリズムのことを指す。化学の世界でモンテカルロ法と言った場合は、モンテカルロ積分かマルコフ連鎖モンテカルロ法のことを指すことが多い。今回はより理解の簡単なモンテカルロ積分について学習しよう。

1 モンテカルロ積分

モンテカルロ積分は積分アルゴリズムの一種である。 モンテカルロ積分とは、乱数を使った数値積分のことである。以前の講義で数値積分について扱った時は等間隔グリッドを用いて積分を行ったのに対し、モンテカルロ積分では乱数で生成した点における関数値を用いて積分を行う。 一次元の積分では等間隔グリッドもかなり精度が良いが、高次元の積分ではモンテカルロ積分に軍配が上がることが多い。 まずは今回用いるライブラリをインポートしよう。

```
import numpy as np
from scipy.special import gamma
import matplotlib.pyplot as plt
```

まずは導入として、簡単な二次元の関数の積分をやってみよう。 二次元の積分ではまだ次元が少ないので、等間隔グリッドと比べてそれほどモンテカルロ積分が有用になるわけではないが、まずは参考になるだろう。 最初は非常に簡単な関数、

$$f(x,y) = x^2 + y^2$$

を$-1 \leq x \leq 1,\ -1 \leq \mathrm{y} \leq 1$の範囲で積分してみよう。参考として、この関数の積分の解析的な答えは

$$\int_{-1}^{1}\int_{-1}^{1}(x^2+y^2)\mathrm{d}x\,\mathrm{d}y = \frac{8}{3}$$

となる。

まずは等間隔グリッドを用いてこの関数値を計算してみよう。x_iやy_iを$-1 \leq x \leq 1, -1 \leq \mathrm{y} \leq 1$の範囲を等間隔に切った点として、$n$が十分大きい時、区分求積より、

$$\int_{-1}^{1}\int_{-1}^{1}f(x,y)\mathrm{d}x\,\mathrm{d}y \approx \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\frac{1-(-1)}{n}\frac{1-(-1)}{n}f(x_i,y_i) = \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\frac{4}{n^2}f(x_i,y_i)$$

Created by Akihide Hayashi @ Osaka University. https://akhdhys.github.io/

が成立するので、以下のコードで積分を実行できる。

```
def x2y2(xy):
    x, y = xy
    return x * x + y * y
```

```
n = 500
x = np.linspace(-1, 1, n, endpoint=False)
y = np.linspace(-1, 1, n, endpoint=False)
answer = 0
for xi in x:
    for yj in y:
        answer += x2y2([xi, yj]) * 4 / (n * n)
print("numerical: {}".format(answer))
print("exact    : {}".format(8 / 3))
print("error    : {}".format(8 / 3 - answer))
```

それなりの精度で積分できたと思う。次に乱数で得た点に関してプログラムを書いてみよう。考え方はあまり変わらない。各点が乱数で得られるという所だけが異なる。

$$\int_{-1}^{1}\int_{-1}^{1} f(x,y)\mathrm{d}x\,\mathrm{d}y \approx \sum_{i=0}^{n-1}\frac{4}{n}f(x_i,y_i)$$

先の等間隔グリッドの例では$500\times500=250000$点を使って積分したので、今回のモンテカルロ積分でも公平に 250000 点を使って積分しよう。

```
n = 500 * 500
answer = 0
for _ in range(n):
    xi = np.random.random() * 2 - 1
    yi = np.random.random() * 2 - 1
    answer += x2y2([xi, yi]) * 4 / n
print("numerical: {}".format(answer))
print("exact    : {}".format(8 / 3))
print("error    : {}".format(8 / 3 - answer))
```

どうだろう？まだまだ等間隔グリッドの方が精度が高いが、乱数を使ったモンテカルロ積分でもそこそこの精度が出るのではないだろうか？

次に、また二次元の例であるが、円の面積を数値計算で求めてみよう。円の面積を求めるには、以下のような関数を積分すれば良い。

$$f(x,y) = \begin{cases} 1 & x^2 + y^2 \leq 1 \\ 0 & \text{else} \end{cases}$$

まずは等間隔グリッドを用いてやってみよう。

```
def circle_area(xy):
    x, y = xy
    if x * x + y * y <= 1:
        return 1
    else:
        return 0
```

```
n = 500
x = np.linspace(-1, 1, n, endpoint=False)
y = np.linspace(-1, 1, n, endpoint=False)
answer = 0.0
for xi in x:
    for yj in y:
        answer += circle_area([xi, yj]) * 4 / (n * n)
print("numerical: {}".format(answer))
print("exact    : {}".format(np.pi))
print("error    : {}".format(np.pi - answer))
```

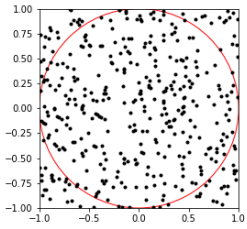
モンテカルロ積分も実装してみよう。

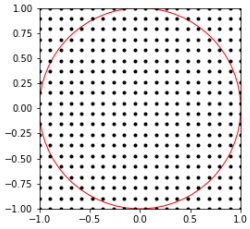
```
n = 500 * 500
answer = 0
for _ in range(n):
    xi = np.random.random() * 2 - 1
    yi = np.random.random() * 2 - 1
```

```
    answer += circle_area([xi, yi]) * 4 / n
print("numerical: {}".format(answer))
print("exact    : {}".format(np.pi))
print("error    : {}".format(np.pi - answer))
```

どうだろう？まあ悪くない精度でπに近づくのではないだろうか？

ここで、等間隔グリッドによる積分とモンテカルロ積分のイメージを再考しよう。





左がモンテカルロ、右が等間隔グリッドだ。どちらも計 400 点を用いている。一辺の長さが 2 で面積が 4 の正方形にたくさん点を打ち、赤い円の中に入った点の割合を 4 に掛けることで円の面積が求められている。

さあ、ここで各積分を関数にしておこう。任意の長方形の領域で任意の関数 f に対してモンテカルロ積分を実装することを目標に、上で使った積分のコードを改良していこう。まずは一旦、上の内容を関数にする。まずは等間隔グリッドから。

```
def two_dimension_integral(f, n):
    x = np.linspace(-1, 1, n, endpoint=False)
    y = np.linspace(-1, 1, n, endpoint=False)
    answer = 0
    for xi in x:
        for yj in y:
            answer += f([xi, yj])
    return answer * 4 / (n * n)
print(two_dimension_integral(circle_area, 500))
```

モンテカルロ積分も実装しよう。

```
def montecarlo(f, n):
```

```
    answer = 0
    for _ in range(n):
        xi = np.random.random() * 2 - 1
        yi = np.random.random() * 2 - 1
        answer += f([xi, yi]) * 4 / n
    return answer

print(montecarlo(circle_area, 250000))
```

さて、この実装でも良いのだが、積分に時間がかかるのは嫌なので、もう少しパフォーマンスをチューニングしよう。np.random は同時に複数のランダム値を返せることを思い出すと、上のコードはちょっと短縮できる。

```
def montecarlo(f, n):
    answer = 0
    for _ in range(n):
        xi, yi = np.random.random(2) * 2 - 1
        answer += f([xi, yi]) * 4 / n
    return answer

print(montecarlo(circle_area, 250000))
```

まだ$-1 \leq x,y \leq 1$の範囲しか積分できていないので、これを$s_x \leq x \leq e_x,\ s_y \leq y \leq e_y$の範囲で積分できるように拡張しよう。

```
def montecarlo(f, n, s, e):
    answer = 0
    s, e = np.array(s), np.array(e)
    for _ in range(n):
        xi, yi = np.random.random(2) * (e - s) + s
        answer += f([xi, yi]) * np.prod(e - s) / n
    return answer

print(montecarlo(circle_area, 250000, [-1, -1], [1, 1]))
```

次に任意の次元に対応させよう。

```
def montecarlo(f, n, d, s, e):
```

```
    answer = 0
    s, e = np.array(s), np.array(e)
    for _ in range(n):
        ri = np.random.random(d) * (e - s) + s
        answer += f(ri) * np.prod(e - s) / n
    return answer

print(montecarlo(circle_area, 250000, 2, [-1, -1], [1, 1]))
```

普通の状況では重みの掛け算は最後にまとめて行うほうが速い。

```
def montecarlo(f, n, d, s, e):
    answer = 0
    s, e = np.array(s), np.array(e)
    for _ in range(n):
        ri = np.random.random(d) * (e - s) + s
        answer += f(ri)
    return answer * np.prod(e - s) / n

print(montecarlo(circle_area, 250000, 2, [-1, -1], [1, 1]))
```

numpy の機能をフルに使うと以下のようにも書ける。

```
def montecarlo(f, n, d, s, e):
    s, e = np.array(s), np.array(e)
    r = np.random.random((n, d)) * (e - s) + s
    return sum(f(ri) for ri in r) * np.prod(e - s) / n

print(montecarlo(circle_area, 250000, 2, [-1, -1], [1, 1]))
```

さて、これでモンテカルロ積分のコードは完了だ。等間隔グリッドとの精度比較をしてみよう。

```
n = list(range(10, 500, 5))
errors_grid = [np.pi - two_dimension_integral(circle_area, ni) for ni in
n]
errors_mont = [np.pi - montecarlo(circle_area, ni*ni, 2, [-1]*2, [1]*2)
for ni in n]
```

```
plt.plot(n, errors_grid, label="grid")
plt.plot(n, errors_mont, label="montecarlo")
plt.axhline(0, c='k')
plt.legend()
```

次は半径 1 の球の体積を求めてみよう。モンテカルロ積分のコードはそのまま使いまわすことが出来る。

```
def sphere_volume(r):
    if r @ r <= 1:
        return 1
    else:
        return 0

numerical = montecarlo(sphere_volume, 250000, 3, [-1, -1, -1], [1, 1, 1])
print("numerical: {}".format(numerical))
print("exact    : {}".format(4 / 3 * np.pi))
print("error    : {}".format(4 / 3 * np.pi - numerical))
```

もっと高次元の超球の体積も求めてみよう。n 次元の半径 R の超球の体積は

$$V_n(R) = \frac{\pi^{n/2}}{\Gamma(n/2+1)} R^n$$

であることが知られている。

```
n = 6
numerical = montecarlo(sphere_volume, 250000, n, [-1] * n, [1] * n)
exact = np.pi ** (n / 2) / gamma(n / 2 + 1)
print("numerical: {}".format(numerical))
print("exact    : {}".format(exact))
print("error    : {}".format(exact - numerical))
```

2 課題

3 次元の球に対して等間隔グリッドの積分を実装し、同じ点の数になるように等間隔グリッドとモンテカルロ法での積分を行い、精度を比較せよ。 ただし、これまでのように一次元あたり 500 点を取っていると高次元の積分は時間がかかりすぎるので、適宜節約せよ。

もし余裕があれば 4 次元以上もやってみると良い。次元が上がれば上がるほどモンテカルロ積分が有利になることが分かるだろう。

3 最後に

今回紹介出来なかったが、マルコフ連鎖モンテカルロ法(MCMC)は確率分布に従うデータ列を生成することが出来るアルゴリズムであり、磁性やタンパク質、高分子等に関するシミュレーションで活躍する。

非常に有用なアルゴリズムであるため、もし興味があれば MCMC の一種であるメトロポリス・ヘイスティングやギブズ・サンプラーについて調べてみて欲しい。