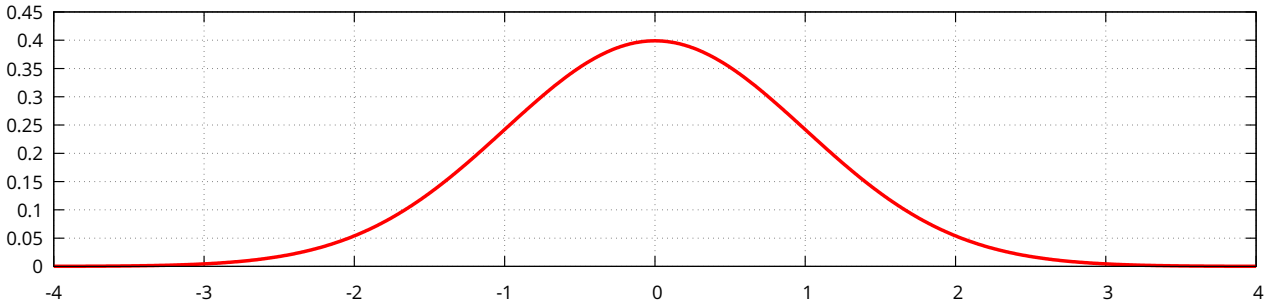
正規分布に従う疑似乱数を生成する方法は？速度は？調べてみました！

Kumpei IKUTA

2020-12-09



はじめに

こんにちは。応用情報 Advent Calender 2020 も 9 日目に突入しましたが、みなさまいかがお過ごしでしょうか。知的情報処理研究室 M1 の生田です。

正規分布は自然現象を良く表現する[*1]ことから、数値シミュレーションをはじめとした工学上の応用が数多くあります。また、正規分布からのサンプリングは、他の確率分布（t 分布やベータ分布など）に従う乱数の生成にも用いられるため、直接的に正規分布を用いないシミュレーションでも、間接的に正規分布が必要になることがあります。私たちの研究室でも、ニューラルネットワークの重みの初期値など、正規分布は至るところで活躍しています。

しかし、現在広く使われている疑似乱数生成アルゴリズム（線形合同法やメルセンヌ・ツイスターなど）やハードウェア乱数生成器（x86 の `RDRAND` 命令や Linux の `/dev/random` など）は、どれも一様乱数しか生成することができません。そのため、一様分布からサンプルした乱数を、何らかの工夫によって正規分布に変換する必要があります。本記事では、 6 つのアルゴリズムの概要を紹介したのち、それぞれについてパフォーマンスの比較を行っていきます。数学的な正当性の証明については、各アルゴリズムの文献を参照してください。

本記事を書くにあたり、紹介するアルゴリズムの選定や実装上のテクニックについて、四辻哲章『計算機シミュレーションのための確率分布乱数生成法』（プレアデス出版、2010）[1] を参考にしました。

また、実験に用いたコード（および本記事の TEX ソースコードや図表の生成スクリプト）は GitHub のレポジトリ から閲覧できるので、実装について気になる点があった方はぜひご覧下さい。Issue や Pull Request によるコントリビューションもお待ちしております。

[*1] ただし、20 世紀以降の複雑系科学の発展に伴い、この考え方は過去のものとなりつつあるようです。

アルゴリズムの紹介

CLT による近似

確率変数の和の分布は、サンプル数を増やしていくと正規分布に収束することが知られています（CLT; 中心極限定理）。とくに、区間 $(0, 1)$ から一様にサンプルした確率変数を 12 個足すと、平均 6、分散 1 の分布となるので、これを正規分布の近似として利用する方法があります。下式では、この性質を用い、一様乱数 U_i から正規分布（の近似）に従う確率変数 Z をサンプルしています。

$$Z = \sum_{i=1}^{12} U_i - 6$$

単純な加減算のみで正規分布を近似できる一方、1 つの確率変数をサンプルするのに 12 個の一様乱数をサンプルする必要があるため、実用されるケースは限定的です。

Box-Muller 法

Box-Muller 法 [2] は、2 次元正規分布の極座標変換に基づく以下の関数を用い、区間 $(0, 1]$ から一様にサンプルした 2 つの乱数（下式の U_1 と U_2）を変換します。

$$Z_1 = \sqrt{-2\log(U_1)}\cos(2\pi U_2)$$
$$Z_2 = \sqrt{-2\log(U_1)}\sin(2\pi U_2)$$

一様分布から 2 つの変数をサンプルし、 2 つの正規分布に従う（独立な）変数をサンプルすることができます。そのため、

多くの実装では状態を保持することで計算回数を減らす工夫をしています。CPython の `random.gauss` はこのアルゴリズムで実装されています。

Polar 法

Box-Muller 法は、一度のサンプリングに平方根・対数・三角関数という 3 つの数学関数の演算が必要でした。1960 年代当時、これらの計算は非常に遅かったため、少しでも呼び出し回数を減らすために考えられたのが Polar 法 [3] です。Polar 法は、以下の式で変換を行います。

$$Z_1 = U_1\sqrt{\frac{-2\log s}{s}}$$
$$Z_2 = U_2\sqrt{\frac{-2\log s}{s}}$$
$$\text{ただし } s = U_1^2 + U_2^2$$

ここで U_1 と U_2 は 0 を除く単位円板上 $(0 < s < 1)$ から一様にサンプルする必要がありますが、これを直接実装するのは困難です。そこで、 U_1, U_2 それぞれを $(-1, 1)$ の区間から一様にサンプルし、単位円板から外れていたらサンプルしなおすという処理を行います（棄却サンプリング）。この場合、U_1, U_2 は約 21.5 % の確率[*2]で再サンプルが必要になってしまいますが、それでも三角関数の計算よりは速いだろうということが期待されます。GCC や Clang の C++ コンパイラでは、標準ライブラリの `std::normal_distribution` をこのアルゴリズムで実装しています。

Kinderman 法

まず、任意の確率密度関数 $f(x)$ について、$\mathbb{R}^2$ 上の集合 C_f を以下のように定義します。

$$C_f = \left\{ (u_1, u_2) \,\middle|\, 0 \le u_2 \le \sqrt{f\left(\frac{u_2}{u_1}\right)} \right\}$$

この集合上から一様にサンプルした点 (U_1, U_2) について、U_2/U_1 は確率密度 $f(x)$ に従って分布します。Kinderman 法 [4] では、この性質を用いて正規分布からのサンプリングを行います。標準正規分布に対する C_f は、下式のようになります[*3]。

$$C_f = \left\{ (u_1, u_2) \,\middle|\, \frac{u_2^2}{u_1^2} + \log(2\pi) \le -4\log u_1 \right\}$$

当然、この集合から要素を直接サンプルすることはできないので、Polar 法と同様に棄却サンプリングを行います。2 つの一様乱数を用いて長方形領域からサンプルする場合、棄却確率を最小化するためには、U_1 を区間 $\left(0, (2\pi)^{-\frac{1}{4}}\right)$ から、U_2 を区間 $\left(-\sqrt{\frac{\sqrt{2/\pi}}{e}}, \sqrt{\frac{\sqrt{2/\pi}}{e}}\right)$ から一様にサンプルする必要があります。この場合の棄却確率 は約 26.9 % です。

[*2] $(4-\pi)/4 \approx 0.215$

[*3] 元論文では分散が 2π の正規分布に対する C_f しか示されていないので、注意が必要です。

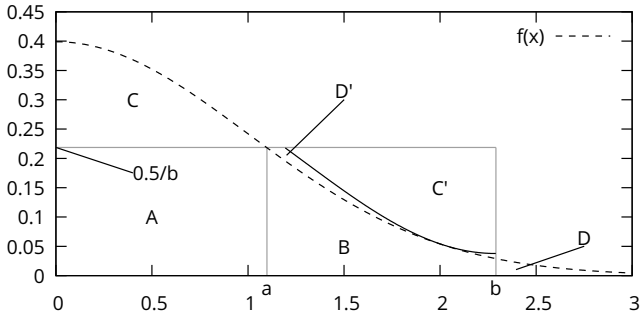


図 1　Monty-Python 法に基づく変形

また、CPython の `random.normalvariate` はこのアルゴリズムで実装されています。

Monty-Python 法

Monty-Python 法 [5] は、確率密度関数のグラフを変形して長方形に詰めこむことで、棄却サンプリングを効率化するというアイデアに基づくアルゴリズムです。

正規分布に対して Monty-Python 法を適用する場合、確率密度関数を図 1 で示す A, B, C, D の 4 つの領域に分割し、C を C' に回転・移動させることで、幅 b 、高さ $0.5/b$ の長方形に詰め込みます。この際、$a = \sqrt{\log 4}$, $b = \sqrt{2\pi}$ とし、さらに C' を適切にスケーリングすることで、棄却確率を約 1.2% まで下げることができます。また、B と C' の間の領域 D' は D と同じ面積なので、サンプルした点が D' 上にあった場合は D からのサンプリングを行います。

Ziggurat 法

Ziggurat 法 [6, 7] は、数学関数の呼び出しを減らすだけでなく、浮動小数点数の算術・比較演算の回数までも極限まで削減することを目的として考案されたアルゴリズムです。事前計算したルックアップテーブルに基づいて確率密度関数を n 個（ここで n は 256 または 128 が用いられることが多い）の面積が等しい長方形で覆い、きわめて効率良く棄却サンプリングを行います。高効率なサンプリングとビット演算を多用した計算のテクニックにより、**99.3 % の確率で浮動小数演算を乗算 1 回のみに抑えることができます。**

Go の `rand.NormFloat64` や Julia の `randn` では Ziggurat 法が使われています。また、Boost や GNU Scientific Library などの著名なライブラリでも採用されているようです。

実装・ベンチマーク

紹介したアルゴリズムを C++ で実装し、速度の比較を行いました。実験に用いた CPU は AMD Ryzen 7 1700 (3GHz)、コンパイラは GCC (g++) です。また、コンパイル時には `-O2` オプションによる最適化を付けました。

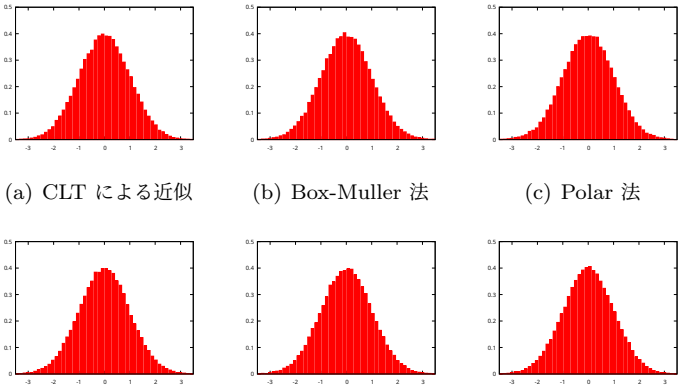
以上の環境のもと、各アルゴリズムで 10^9 個のサンプリングを行いました。表 1 に計測結果をまとめます。また、図 2 はそれぞれのアルゴリズムで生成した 10^5 個の乱数列のヒストグラ

表 1　ベンチマーク結果 ($N = 10^9$)

アルゴリズム	実行時間 (ms)	速度 [a]	N_U [b]
CLT による近似	214998	4651208	12.0000
Box-Muller 法	39672	25206459	1.0000
Polar 法	39033	25619200	1.2732
Kinderman 法	72048	13879701	2.7376
Monty-Python 法	65234	15329456	1.5578
Ziggurat 法	13153	76026520	1.0220

[a] 1 秒あたりのサンプル数

[b] 正規乱数を 1 つサンプルするごとに必要な一様乱数の個数（平均値）



(a) CLT による近似　(b) Box-Muller 法　(c) Polar 法

(d) Kinderman 法　(e) Monty-Python 法　(f) Ziggurat 法

図 2　生成した乱数列のヒストグラム ($N = 10^5$)

ムです。少なくともグラフの形状的には、いずれのアルゴリズムも正しく正規分布を生成できているようです。

考察

CLT による近似は、計算がきわめて単純であるにもかかわらず、圧倒的に遅いという結果になりました。このことから、正規乱数を発生させるプロセスにおいて、一様乱数のサンプリングがかなりの割合を占めていることが推察できます。

Box-Muller 法を高速化させる目的で考えられた Polar 法は、狙いどおり Box-Muller 法よりも（わずかながら）速くなりました。Polar 法が考案された 1960 年代に比べ、現代の CPU は浮動小数点演算を遥かに高速に行えるため、高速化の効果がかなり小さくなったものと考えられます。

Kinderman 法は他の方法に比べてかなり遅くなりました。これは i) 正規分布から 1 回サンプルするのに一様乱数からのサンプルが最低 2 回必要な点や、ii) Kinderman 法はそもそもあまりスピードを重視しておらず、任意の分布を簡単にサンプルできる一般的なアルゴリズムとしての側面が強い（元論文でも、アルゴリズムが “short” であることが強調されています）点を考慮すると、妥当な結果と言えます。

きわめて高効率な Ziggurat 法は、当然ながら他のアルゴリズムよりも遥かに高速になりました。ですが、LUT を保持しておくのに多く（今回の実装では約 5KiB）のメモリ領域を要する点には注意が必要です。まさに space-time トレードオフの典型例と言えるでしょう。

また、CUDA 等の GPGPU 環境では、分岐やメモリアクセスが少なく、決め打ちで計算できる Box-Muller 法が有利になるかもしれません。機会があれば、こちらについても実験してみたいと思います。

おわりに

いかがでしたか？がんばって調べてみましたが…よくわかりませんでした！自分が産まれてもいないような時代の論文を解読して実装するという体験は、なかなか新鮮なものがありました。浮動小数点演算を減らすための工夫が FPU の高速化・内蔵化によって微妙になったり、メモリをたくさん使って LUT を参照する Ziggurat が高速だったりと、計算機の進化の潮流を肌で感じることができ、とても楽しかったです。

さてさて、明日の応用情報 Advent Calender 2020 でも、D1 のしゅんけーパイセンが何か面白いことを書いてくれるそうです。お楽しみに！

参考文献

[1] 四辻哲章. 計算機シミュレーションのための確率分布乱数生成法. プレアデス出版, 2010.

[2] George EP Box. A note on the generation of random normal deviates. *Ann. Math. Stat.*, Vol. 29, pp. 610–611, 1958.

[3] G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *SIAM Review*, Vol. 6, No. 3, pp. 260–264, 1964.

[4] Albert J Kinderman and John F Monahan. Computer generation of random variables using the ratio of uniform deviates. *ACM Transactions on Mathematical Software (TOMS)*, Vol. 3, No. 3, pp. 257–260, 1977.

[5] George Marsaglia and Wai Wan Tsang. The monty python method for generating random variables. *ACM Transactions on Mathematical Software (TOMS)*, Vol. 24, No. 3, pp. 341–350, 1998.

[6] George Marsaglia and Wai Wan Tsang. A fast, easily implemented method for sampling from decreasing or symmetric unimodal density functions. *SIAM Journal on scientific and statistical computing*, Vol. 5, No. 2, pp. 349–359, 1984.

[7] George Marsaglia, Wai Wan Tsang, et al. The ziggurat method for generating random variables. *Journal of statistical software*, Vol. 5, No. 8, pp. 1–7, 2000.