計算物理演習レポート第 1 回 : 超球体積

理学部理学科物理学コース 学籍番号 20S2035Y
山本　凜

2023/1/10

1　要旨

超球体積を計算する手法の候補としてモンテカルロ法 (MC) が挙げられる. しかし, MC を用いて実際にシミュレーションをすると, 高次元では次元の呪いによりカウント数が増えず, 体積を求めることができない. また, その課題を克服するためにサンプル点の打ち込み数を増やしたとしても, 今度はプログラムの実行時間が膨大に長くなってしまうという問題が発生する. さらに, サンプル点を増やすという対策も高次元になるほど効果が薄まる. これらの問題点に対する解決策としては, 実行時間を短縮するようにプログラムコードの高速化を行う, アルゴリズムの見直しをする等が考えられる.

2　序論

${x_1}^2 + {x_2}^2 + \cdots + {x_d}^2 \leq r^d$ 領域の d 次元単位超球体積である V_d は, 一般に (2.1) 式のようにガンマ関数を用いて表すことができる.

$$V_d = \frac{\pi^{d/2} r^d}{\Gamma(d/2+1)} \tag{2.1}$$

$$\Gamma(x) = \int_0^\infty t^{x-1} e^{-t} \mathrm{d}t \tag{2.2}$$

超球体積は MC を用いて数値的に求めることもできるが, MC では高次元の超球体積に対応できないという問題点がある. これは次元の呪いとも呼ばれ, 高次元特有の振る舞いに起因する.

そこで, 本レポートの目標としては, ガンマ関数を用いた単位超球体積の解析解と, MC を用いて算出した数値解とを比較して, 単位超球体積を数値的に求める際の問題点について探り, その解決法を模索することをおく.

3　本論

3.1　超球体積の公式を導出

まずは, 超球体積を解析的に求める公式を簡単に導出する. d 次元球の体積は半径の d 乗に比例する. つまり, 半径 r の超球の体積は, ある係数 c_d を用いて $c_d\ r^d$ と表すことができる. また, d 次元球の表面積は, 体積を r 微分して $dc_d\ r^{d-1}$ と書くことができる.

$$V_d = c_d\ r^d \tag{3.1}$$

$$S_d = dc_d r^{d-1} \tag{3.2}$$

次の積分を考える.

$$I_d = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} e^{-({x_1}^2 + {x_1}^2 + \cdots {x_d}^2)} \mathrm{d}x_1 \mathrm{d}x_2 \cdots \mathrm{d}x_d \tag{3.3}$$

(3.3) を 2 通りの方法で計算する. まずは, ガウス積分を用いて次のように計算する.

$$I_d = \left(\int_{-\infty}^{\infty} e^{-{x_1}^2} \mathrm{d}x_1\right)\left(\int_{-\infty}^{\infty} e^{-{x_2}^2} \mathrm{d}x_2\right) \cdots \left(\int_{-\infty}^{\infty} e^{-{x_d}^2} \mathrm{d}x_d\right) \tag{3.4}$$

$$= \left(\int_{-\infty}^{\infty} e^{-x^2} \mathrm{d}x\right)^d \tag{3.5}$$

$$= \pi^{d/2} \tag{3.6}$$

次に, 極座標を用いて積分変数 r 以外について計算すると, 表面積が出てくるので (3.2) を用いて, 次のように計算を進めることができる.

$$I_d = \int_0^{\infty} e^{-r^2} S_d \, \mathrm{d}r \tag{3.7}$$

$$= d \, c_d \int_0^{\infty} e^{-r^2} r^{d-1} \mathrm{d}r \tag{3.8}$$

ここで, $t = r^2$ と置換すると

$$d \, c_d \int_0^{\infty} e^{-r^2} r^{d-1} \mathrm{d}r = d \, c_d \int_0^{\infty} e^{-t} \, t^{\frac{d-1}{2}} \, \frac{1}{2} \, t^{-\frac{1}{2}} \mathrm{d}t \tag{3.9}$$

$$= \frac{d}{2} \, c_d \int_0^{\infty} t^{\frac{d}{2}-1} e^{-t} \mathrm{d}t \tag{3.10}$$

(2.2) より

$$\frac{d}{2} \, c_d \int_0^{\infty} t^{\frac{d}{2}-1} e^{-t} \mathrm{d}t = c_d \, \frac{d}{2} \, \Gamma\left(\frac{d}{2}\right) \tag{3.11}$$

$$= c_d \, \Gamma\left(\frac{d}{2} + 1\right) \tag{3.12}$$

(3.6) 式, (3.12) 式は等価なので, 係数 c_d を求めることができる.

$$c_d \, \Gamma\left(\frac{d}{2} + 1\right) = \pi^{d/2} \tag{3.13}$$

$$c_d = \frac{\pi^{d/2}}{\Gamma(d/2 + 1)} \tag{3.14}$$

最後に (3.1) 式を用いると

$$V_d = \frac{\pi^{d/2} r^d}{\Gamma(d/2 + 1)} \tag{2.1}$$

超球体の体積は以上のようにして導出できる.

3.2　モンテカルロ法を用いて超球体積を求める手法

上記の公式を既知としなかった場合, 我々はどのようにして超球の体積を求めればよいか. まずは, 2 次元の単位円について考えるとする. この円の面積を S として, x, y に 0 から 1 の範囲の一様乱数を振ると, 点 $(x,\ y)$ は面積 1 の正方形の中に一様に分布するはずである. その中で, 角度 $\pi/4$ の扇形に入っているものの割合は, 正方形と扇形の面積の比率, すなわち $S/4$ となる. ここで, 十分な量 (NTRY 個) のサンプル点を打ち込み, そのうちで扇形の部分に入っているものをカウントした結果を ncnt 個とすれば, 単位円の面積 S は以下のようにして求めることができる.

$$\frac{\mathrm{ncnt}}{\mathrm{NTRY}} \simeq \frac{S}{4} \tag{3.15}$$

$$S \simeq \frac{\mathrm{ncnt}}{\mathrm{NTRY}} \times 4 \tag{3.16}$$

同じようにして d 次元の球の体積も求める. 辺の長さが 1 の d 次元の立方体 (超立方体) を考えると, その体積は 1 である. その中に占める, d 次元球の $1/2^d$ の区画の割合から, 超球の体積を数値的に求めることができる. これより対象とする試行は, 「1. サンプル点が超球に入るか入らないかの 2 択であること, 2. サンプル点が超球に入る確率と入らない確率がそれぞれ決まっていること, 3. 各試行が独立していること.」を満たすので, ベルヌーイ試行と呼ばれ, この確率は二項分布に従うことが知られている.

まずは, 2 次元球で, 打ち込むサンプル点の総数を 2^{20} 個にする. 100 回のシミュレーションを行うが, サンプル点が超球の内部にある確率は前述のように二項分布に従う. サンプル点の打ち込み回数が十分に多いといえる場合, 二項分布は正規分布に近似できる. 以下はシミュレーション 100 回, 1000 回分の数値解の分布である.

実行結果 1,2

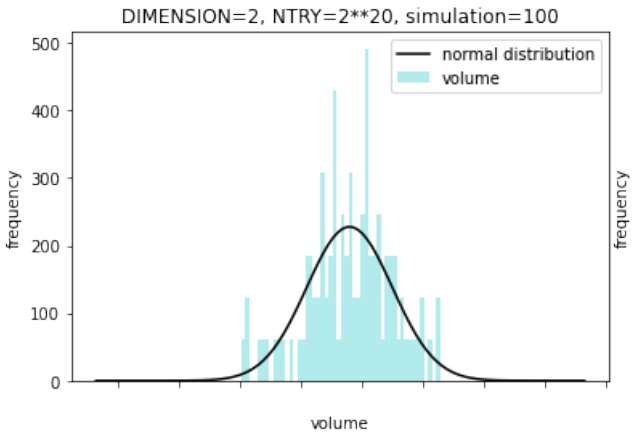

図 1

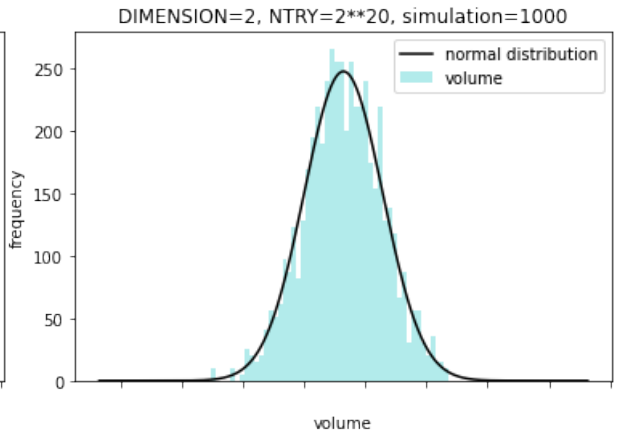

図 2

シミュレーション 1000 回分のデータは図のように正規分布と重なる. シミュレーション 100 回分のデータは尖度は高いものの, 歪度はほとんど 0 であると見れるので, 以降はシミュレーション 100 回分のデータの平均値をとった値を超球体積の数値解として計算する. 2 次元球体積の数値解は

実行結果 3

d= 2 ,V= 3.1419740295410157 pi= 3.141592653589793

この結果は 2 次元の単位球の体積 (単位円の面積)π に対応している. また, NTRY の増加に対しての体積, 相対誤差の推移は以下のようになる.

実行結果 4, 5

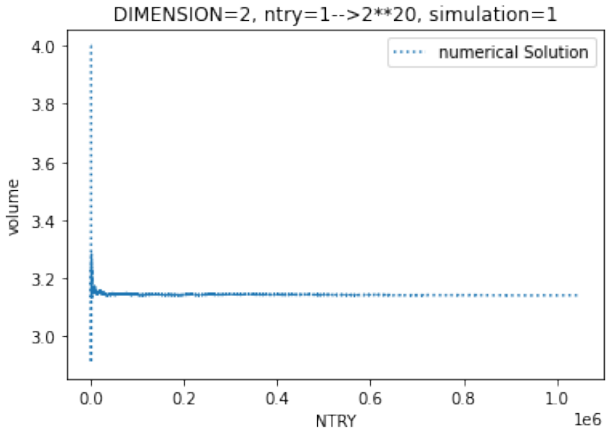

図 3

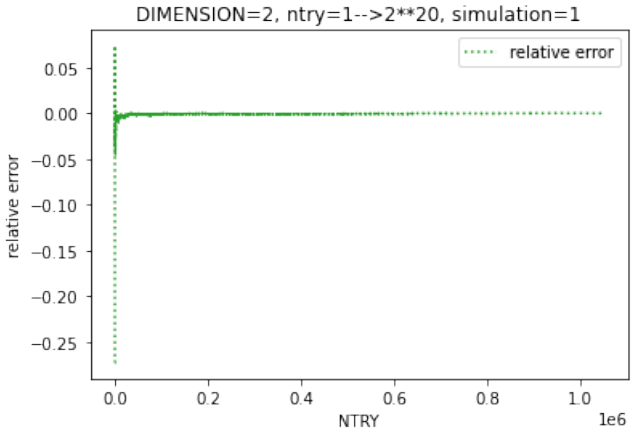

図 4

サンプル点の総数を増やすほど, 単位 2 次元球体積 (単位円の面積) の数値解が π に収束していること, 相対誤差が 0 に限りなく近づいていることが分かる. MC による数値解がどの程度まで解析解に一致しているのか, また, 何次元まで適応に耐えられるのかを調べていく.

3.3 数値解と解析解の比較

2〜20 次元までの数値解をグラフ化して, これらが各次元の解析解とどこまで一致しているかを調べることとする.

実行結果 6

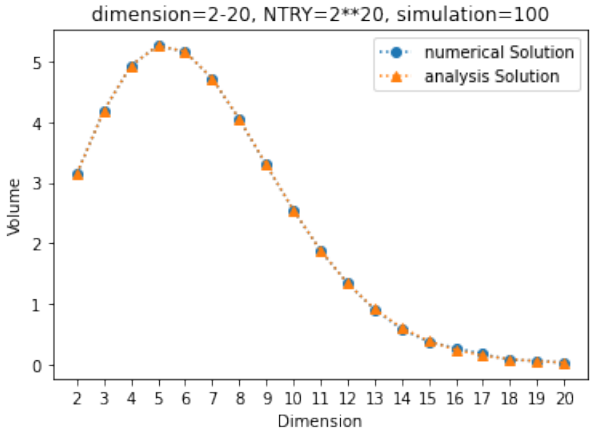

図 5

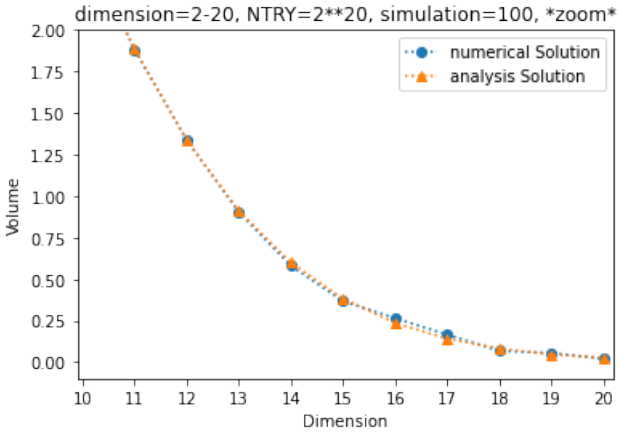

図 6

図 5, 6 を見るに, MC で求めた数値解は次元を変えても, おおよそ解析解と一致していることが見て分かる. そこで, 試しに 100〜110 次元についても同じ処理をしてみる.

実行結果 7

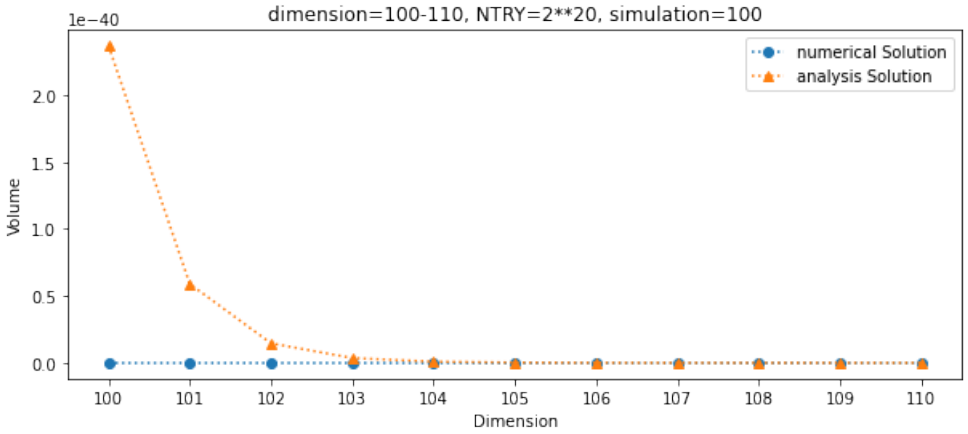

図 7

図 7 から, ある程度高次元になると, 超球体積の解析解はまだ値を持っているのに対して, 数値解は 0 になってしまうという問題点が発見された. MC で超球体積を求める際には, 超立方体にサンプル点を打ち込み, その中にある超球体にサンプル点が当たった回数と打ち込んだ総数との比が必要なので, サンプル点が 1 回は超球体に当たらないと数値解は算出できない. そこで, MC において, どれほど高次元になったらサンプル点がカウントされなくなるのかを調べるために各次元における相対誤差を算出してみる.

実行結果 8, 9

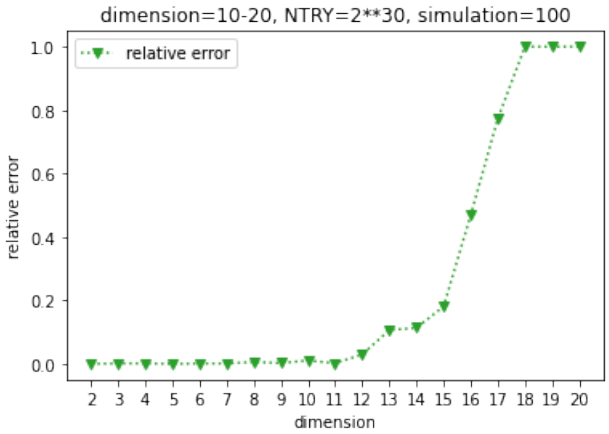

図 8

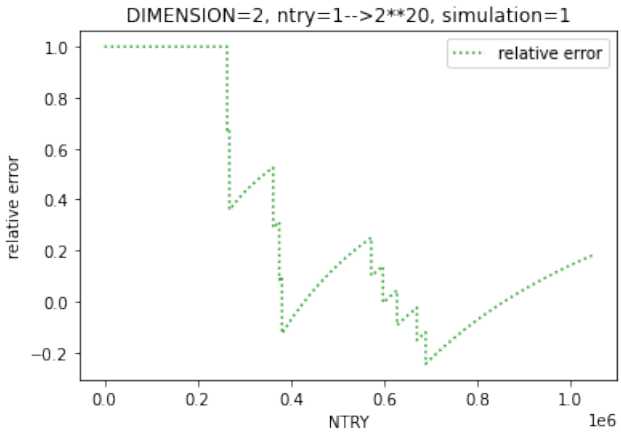

図 9

図 8 を見ると, 段階的に高次元化したときに, 12 次元あたりからだんだんと相対誤差は大きくなり, ついに 18 次元からは 1 になってしまっていることが分かる. 先ほどの図 5, 6 を見ても分かりづらかったが, 実は高次元化するにつれて解析解とのずれが大きくなっており, 18 次元からは体積を 0 として算出してしまっていたことが分かる. 図 9 のように, 具体的に 15 次元で, 1 個ずつサンプル点を増やすことを想定して相対誤差の推移を見ると, NTRY $= 2^{20} \simeq 1,000,000$ 個のサンプル点を打ち込んでも, ばらつきが大きいことが分かり, より多くのサンプル点が必要であると考えられる. しかし, 高次元で MC におけるサンプル点を増やすと, プログラムの処理時間がかなり長くなってしまう.

3.4 MC の精度

3.4.1 ゆらぎ

こうして, 乱数の限界が垣間見えたところで, MC の精度について評価していきたい.

(引用 [1]:早川美徳『MC による積分』)
『超球の体積を V とすれば、ランダムに生成された点が超球の内部に位置する確率は $p = V$ である。N 回のサンプリングを行った場合、点が超球の内部にある回数は二項分布で与えられるので、その平均は、N_p、分散は $Np(1-p)$ である。 すると、体積の推定値のゆらぎは

$$\sqrt{\frac{p(1-p)}{N}}$$

となる。ここで、ゆらぎ（誤差）の N 依存性は d には依存しない。

高次元での積分を、各次元ごとに m 個のデータ点を使って計算しようとすると、データ点の総数は $N \sim m^d$ 程度で増大することになる。 例えば、台形法を使った場合の誤差は σ〜$1/m^2$ 程度となるので、

$$\sigma \sim 1/m^2 \sim N^{-2/d}$$

であり、次元数 d が大きくなると、同じ程度の計算精度を担保するに必要となる N は膨大となる。 言い換えると、高次元の問題になればなるほど、データ点数の観点からは、次元に関係なく $1/\sqrt{N}$ 程度で誤差（ゆらぎ）を減らすことができるモンテカルロ法が有利となってくる』

このように, MC は次元によっての誤差は生じないという特性を持っている. これは, いくら高次元を対象にしようとも, それの影響でゆらぐことはないという確かな MC の強味である. だが, 実際に高次元の超球体積を算出しようとしても, 今まで見てきたように上手くいかない.

3.4.2 次元の呪い

3.2 節で見たように, MC で超球体積を求める際には, サンプル点が 1 回は超球体に当たらないと数値解は算出できない. そこで, 超立方の体積と内部にある超球の体積 (解析解) の比に着目してみる. 超立方の体積は $= 2^d$ で指数関数的に増加するのに対して, 超球体積は図 2 等からも分かるように $d = 5$ をピークに, $N \to \infty$ で 0 に収束する. 従って, 高次元になればなるほど, サンプル点を打ち込む範囲は大きくなり, 当てる対象はどんどん小さくなっていくというイメージが掴める. 確かに, 図 6 から, 18 次元以上の超球体積を算出しようとすると, $2^{20} \simeq 1,000,000$ 個ものサンプル点を打っても, 超球に当たらずにカウントが進まず, 超球体積の数値解が 0 になってしまっているということが確認できている. こういった現象は次元の呪いと呼ばれる.

3.4.3 サンプル点の総数と実行時間

サンプル点の総数の量が, 精度と実行時間にどれほどの影響を与えるのかを見ていく. コード 6 から, デコレータ njit をとり, NTRY=$2^{10,\ 15,\ldots,30}$ における実行結果と実行時間を以下に表す.

実行結果 10, 11, 12, 13

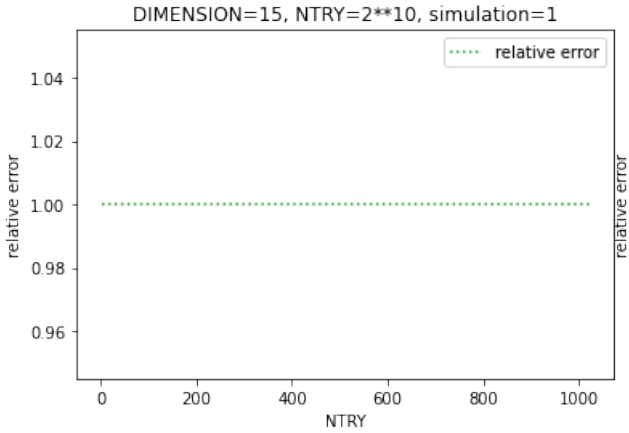

図 10

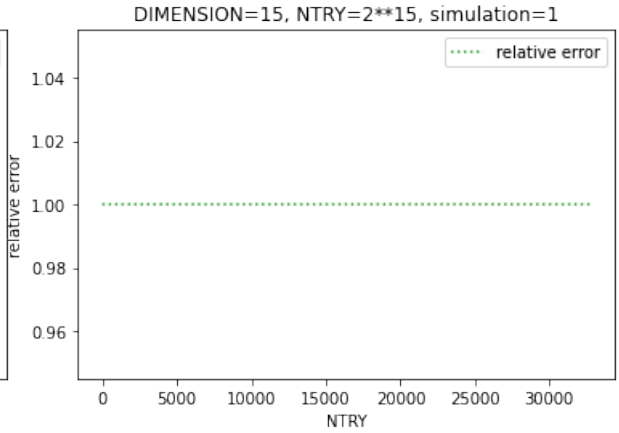

図 11

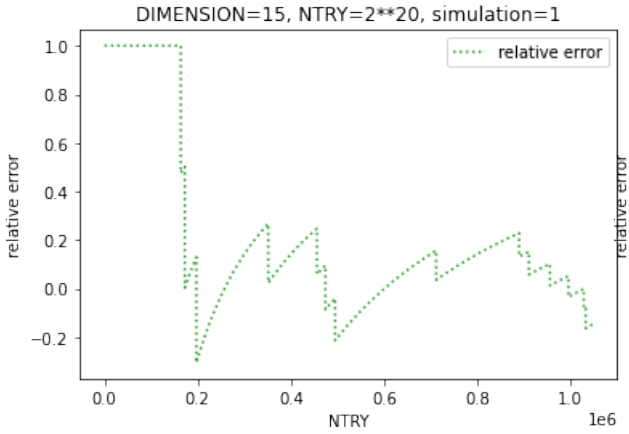

図 12

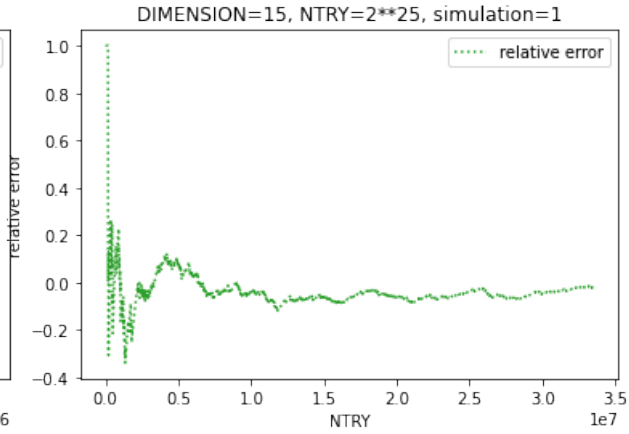

図 13

まず, サンプル点 $2^{10} \simeq 1,000$, $2^{15} \simeq 30,000$ 個程度では, 15 次元球には当たらなかったということが確認できる. また, サンプル点 2^{20} 個ほどから. 徐々に精度が高まり, サンプル点 2^{25} 個ほどでようやく, 相対誤差が 0 に収束していく様子が確認できた. さらに, サンプル点が 2^{30} 個の場合の相対誤差の推移のグラフに関しては処理時間があまりにも長く, セッションがクラッシュしてしまうので, for 文の tqdm 関数を一番負荷が大きい, サンプル点の総数分の処理をする部分に用いることで計測した.[CodeA] 表 1 は打ち込んだサンプル数の総数に対しての, プログラムの実行時間を表したものである.

表 1　サンプル点の総数と実行時間 [s] の関係

NTRY [個]	2^{10}	2^{15}	2^{20}	2^{25}	2^{30}
time [s]	0.494	0.961	8.537	180.021	およそ 4,800

実行結果 14

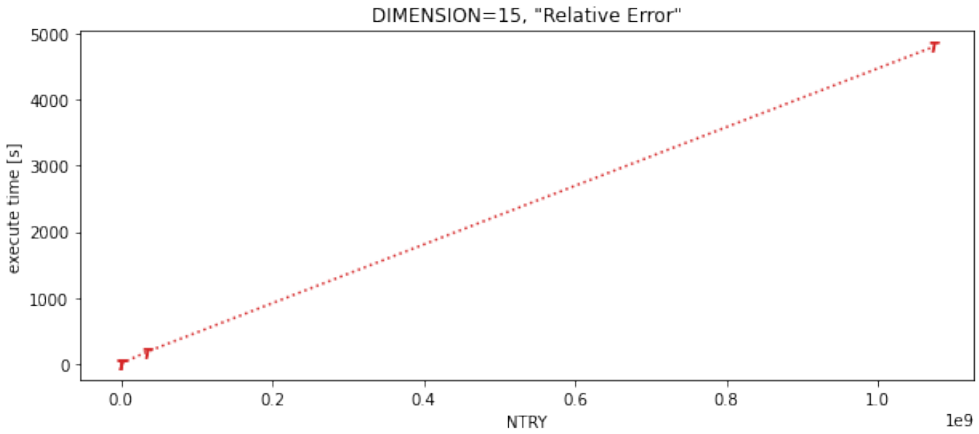

図 14

MC でのコードでは, d 次元回分の繰り返し処理を, NTRY 回分繰り返し処理にかけるので, 高次元になればなるほど, サンプル点を増やしたときの実行時間の伸びは大きくなってしまい, それは線形的に増加するということが図 12 からも分かる. また, 本レポートを構成するプログラムコードの実行時間を短縮する工夫として, 100 万回程度以上繰り返し用いる処理は関数化した上でデコレータ njit をつけ, JIT コンパイラによってランタイム処理を高速化することを図る. ということを行った. 例えばコード 1 の定数を変更して, NTRY$= 2^{20}$, 20 次元の球体積を計算しようとしたときに 69 秒ほどかかる [CodeB1] のだが, 上記の高速化を行うと 1.9 秒ほど [CodeB2] と約 1/30 倍に短縮した上で全く同様の数値解を得ることができる.

3.5 他のアルゴリズムを検討

超球体積を計算する際, MC 以外のアルゴリズムが使えないかを考えてみる. MC の問題点が, 対象が高次元になると計算に必要なサンプル点がかなり多くなってしまうこと, つまり, 次元の呪いにあると考えるならば, 確実に超球体に当たるサンプル点を用いるアルゴリズムを使えばよいのではないか. この意味で, 確実にサンプル点が超球にプロットされ, 超立方との比を得ることのできる格子点法は一見有用に思える.

しかし, プログラムの実行時間のことを考慮に入れると看過できない問題点が浮上する. MC では

$$\text{実行時間} \sim (\text{次元}) \times (\text{サンプル点の総数})$$

のように線形的増加であったものが, 格子点法の場合は, 分割数の分, 処理が必要になるので

$$\text{実行時間} \sim (\text{分割数})^{(\text{次元})}$$

と, 高次元を扱おうとすると, 指数関数的に実行時間が急増する. これでは 3 次元の球の体積さえも満足に計算することができないので, 高次元の超球体積を計算する手法としては格子点法は不向きであると考えられる.

他に候補として挙げられそうなアルゴリズムは, MC のサンプル点打ち込みの結果から乱数判定を行い, より効率的にサンプル点を打ち込むことのできるマルコフ連鎖モンテカルロ法 (MCMC) である.

4　結果と考察

今までの実験から, MC を用いた超球体積の計算の精度は, 次元の大きさとサンプル点の総数に依存することが確認できた. そこで, コード 6 の数値を変えて得たデータ [CodeC1, CodeC2, CodeC3, CodeC4, CodeC5] を有効数字 3 桁で丸めた値をまとめた. (相対誤差 10% 未満を青, 100% 以上を赤で表している.) 以下の表 [CodeC6], 図から結果の整理と考察をする.

表 2　dimension(横: d) と NTRY(縦) の違いによる相対誤差

index	d =10	d =11	d =12	d =13	d =14	d =15	d =16	d =17	d =18	d =19	d =20
2^{10}	0.216	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
2^{15}	0.152	0.095	0.345	0.647	0.166	1.0	1.0	1.0	1.0	1.0	1.0
2^{20}	0.011	0.002	0.03	0.107	0.113	0.181	0.469	0.773	1.0	1.0	1.0
2^{25}	0.006	0.001	0.004	0.018	0.009	0.014	0.079	0.141	0.046	0.341	1.422
2^{30}	0.001	0.002	0.001	0.001	0.002	0.018	0.049	0.082	0.046	0.078	0.097

実行結果 15

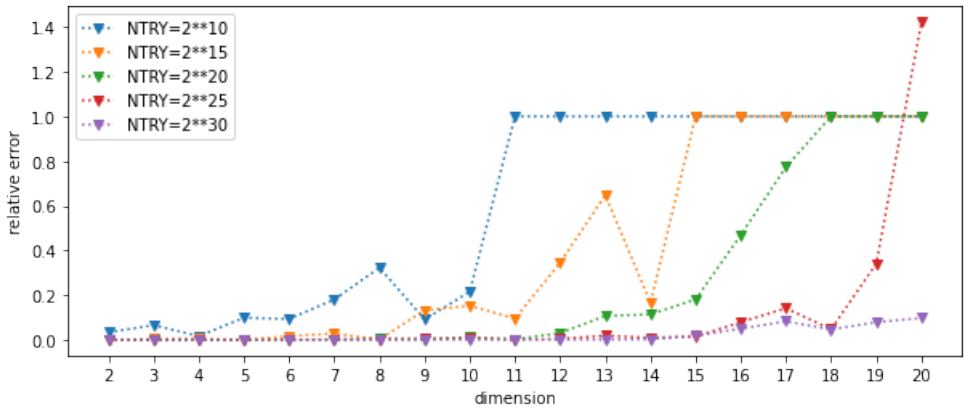

図 15

NTRY= $2^{10,\ 15,\ 20}$ については, それぞれ 11,15,18 次元から相対誤差が 1.0 で打ち止めになっている. これは, 各次元に対応する超球にサンプル点が 1 つも入らなかったことを示し, 次元の呪いに起因する.

また, NTRY=2^{25} については, 20 次元で相対誤差が 1.0 を超えている, これは実際の解析回よりも大きな値が出てしまった, つまり, サンプル点が超球に入る確率と入らない確率が収束する前に実験が終わってしまったことを示し, 対象とする次元に対してのサンプル点が少なかったことに起因する.

さらに, NTRY= 2^{30} での相対誤差は 20 次元でも 0.097 程度に収まっているので, 20 次元の超球体積を求める際には, おおよそ 2^{30} 個程度のサンプル点を打ち込む必要があるということが分かる.

5　結論

3.4 で見たように, MC で求めた数値解は, 高次元においては無視できない誤差が生じてしまう問題がある. それは, 次元の呪いといい, 高次元においては, サンプル点が対象とする超立方に比べて, 超球がはるかに小さくなってしまう現象があり, その場合には相当量のサンプル数を必要とする点. また一方で, 高次元化するに従って, サンプル数を増加すると, MC は $1/\sqrt{N}$ でゆらぎを減らしていくので, サンプル数を増やせば増やすほど, サンプル点の総数が与える, 数値解の収束に対する影響が小さくなってしまうという点. この MC アルゴリズムの特徴 2 点が原因となっていることが分かった. MC で, 高次元の超球体積を精度よく求めるためには相当のサンプル数を用意しなくてはならないので, 一定以上の高次元超球体積を計算する必要がある場合には, 本レポートで実施した以上の高速化のための技術である Cython 等 (引用 [2]:p215) を用いるか, 他のアルゴリズム, 例えば MCMC 等を用いるとよいのではないかと考える.

6　あとがき

格子点法, MCMC を用いて超球体積を求めるプログラムも作成したかったのだが, それが成しえなかったのが心残りである. 特に格子点法については, いろいろと試行錯誤をしたのだが, 結局任意の次元に対応したプログラムを作ることはできなかった. 実際にコードを書こうとすると, 当アルゴリズムについての見識をかなり明瞭にしなくてはならない. これを目指すことで, MC との共通点や違いを感じることができたので, 遠回りであるが MC についても理解を深めることができたので, 今後も 1 つの問題に対しての解決策, アルゴリズムは複数考える癖をつけるようにしていきたい.

付録 A　コード集

- 再現性を得るために, 乱数シードは 100, もしくは 100-199, 1000 に固定している. CodeA 以外のコードは Google Colaboratory 環境で実行確認済みである. (2023.1.9 現在)
- 実行結果 n とコード n は対応している.
- 本レポートに載せたコードは全て GitHub 内の VolumeD_ballipynb にアップロードしている.

Code1

```
# Code1*. 次元球の体積についてのヒストグラム2, 正規分布. def, njit, DIMENSION=2, NTRY=2**20, simulation=100. execution time_s=2
import random as rd
import math as m
import numpy as np
from numba import njit


@njit(cache=True)
def CalculateNumericalVolume_f(si,DIMENSION):
    rd.seed(si)
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
```

```
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** DIMENSION
    return numerical_volume


@njit(cache=True)
def Simulate_f():
    DIMENSION: int = 2
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    SIMULATE: int = LAST_SEED - FIRST_SEED
    for si in (range(FIRST_SEED, LAST_SEED)):
        numerical_volume = CalculateNumericalVolume_f(si, DIMENSION)
        numerical_volume_num.append(numerical_volume)
    return SIMULATE, numerical_volume_num


SIMULATE, data = Simulate_f()

data_mean = np.mean(data)
data_std = np.std(data)

pi=m.pi
x = np.linspace(pi-0.01,pi+0.01,SIMULATE)
y = np.exp(- (x - data_mean) ** 2 / (2 * data_std ** 2)) / (np.sqrt(2 * np.pi) *
  data_std)

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.hist(data, label="volume", bins=50, histtype="bar", density=True, alpha=0.3, color=
  "c")
ax.plot(x,y,label="normal␣distribution",color="k")
ax.legend()
plt.xticks(color="None")
plt.title("DIMENSION=2,␣NTRY=2**20,␣simulation=100")
plt.xlabel("volume")
plt.ylabel("frequency")
plt.show()
```

Code2

```
# Code2. 次元球の体積についてのヒストグラム2, 正規分
  布. def, njit, DIMENSION=2, NTRY=2**20, simulation=1000. execution time_s=20
import random as rd
import math as m
import numpy as np
from numba import njit


```

```
@njit(cache=True)
def CalculateNumericalVolume_f(si,DIMENSION):
    rd.seed(si)
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** DIMENSION
    return numerical_volume


@njit(cache=True)
def Simulate_f():
    DIMENSION: int = 2
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 1100
    SIMULATE: int = LAST_SEED - FIRST_SEED
    for si in (range(FIRST_SEED, LAST_SEED)):
        numerical_volume = CalculateNumericalVolume_f(si, DIMENSION)
        numerical_volume_num.append(numerical_volume)
    return SIMULATE, numerical_volume_num


SIMULATE, data = Simulate_f()

data_mean = np.mean(data)
data_std = np.std(data)

pi=m.pi
x = np.linspace(pi-0.01,pi+0.01,SIMULATE)
y = np.exp(- (x - data_mean) ** 2 / (2 * data_std ** 2)) / (np.sqrt(2 * np.pi) *
  data_std)

import matplotlib.pyplot as plt
fig, ax = plt.subplots()
ax.hist(data, label="volume", bins=50, histtype="bar", density=True, alpha=0.3, color=
  "c")
ax.plot(x,y,label="normal␣distribution",color="k")
ax.legend()
plt.xticks(color="None")
plt.title("DIMENSION=2,␣NTRY=2**20,␣simulation=1000")
plt.xlabel("volume")
plt.ylabel("frequency")
plt.show()
```

Code3

```
# Code3. 次元球の体積を計
  算2. def, njit, DIMENSION=2, NTRY=2**20, simulation=100. execution time_s=1.3
import random as rd
```

```
import numpy as np
from numba import njit
from tqdm.auto import tqdm

@njit(cache=True)
def CalculateNumericalVolume_f(DIMENSION, si):
    rd.seed(si)
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** DIMENSION
    return numerical_volume


@njit(cache=True)
def main():
    DIMENSION: int = 2
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED, LAST_SEED):
        numerical_volume = CalculateNumericalVolume_f(DIMENSION, si)
        numerical_volume_num.append(numerical_volume)
    average_numerical_volume: float = np.average(numerical_volume_num)
    return DIMENSION, average_numerical_volume


DIMENSION, average_numerical_volume = main()

print("d=", DIMENSION, ",V=", average_numerical_volume, "pi=", m.pi)
```

Code4

```
# Code4. 次元のときの体積の推移を求める2. def, njit, NTRY=2**20, simulation=1. execution time_s=1
import random as rd
from numba import njit


@njit(cache=True)
def main():
    rd.seed(100)
    DIMENSION: int = 2
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    ni_num = [ ]
    numerical_volume_num = [ ]
    delimiter: int = 1
    for ni in range(1,NTRY):
        sum: float = 0.0
```

```
            for di in range(DIMENSION):
                x = rd.random()
                sum += x * x
            if sum < 1.0:
                ncnt += 1.0
            if ni % delimiter == 0:
                ni_num.append(ni)
                numerical_volume: float = ncnt / ni * 2.0 ** DIMENSION
                numerical_volume_num.append(numerical_volume)
        return ni_num, numerical_volume_num


ni_num, numerical_volume_num = main()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(ni_num, numerical_volume_num, ":", label="numerical Solution", color="tab:
  blue")
plt.title("DIMENSION=2, ntry=1-->2**20, simulation=1")
plt.xlabel("NTRY")
plt.ylabel("volume")
plt.legend()
plt.show()
```

Code5

```
# Code5. 次元のときの相対誤差の推移をグラフ
  化2. def, njit, NTRY=2**20, simulation=1. execution time_s=1
import random as rd
import math as m
from numba import njit


@njit(cache=True)
def main():
    rd.seed(100)
    DIMENSION: int = 2
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    dimension_num = [ ]
    relative_error_num = [ ]
    delimiter: int = 1
    for ni in range(1, NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1:
            ncnt += 1.0
        if ni % delimiter == 0:
            dimension_num.append(ni)
            numerical_volume = ncnt / ni * 2 ** DIMENSION
            analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION
              / 2.0) + 1.0)
            relative_error: float = (analysis_volume - numerical_volume) /
              analysis_volume
```

```
            relative_error_num.append(relative_error)
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num, relative_error_num, ":", label="relative error", color="tab:green")
plt.title("DIMENSION=2, ntry=1-->2**20, simulation=1")
plt.xlabel("NTRY")
plt.ylabel("relative error")
plt.legend()
plt.show()
```

Code6

```
# Code6. 各次元ごとの超球体積における数値解と解析解をグラフ化. NTRY=2**20, simulation=100. execution time_s=161s
import random as rd
import math as m
import numpy as np
from numba import njit

FIRST_DIMENSION: int = 2
LAST_DIMENSION: int = 20


@njit(cache=True)
def CalculateNumericalVolume_f(si, dimension):
    rd.seed(si)
    numerical_volume: float = 0.0
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume


def NumericalSimulations_f(dimension):
    numerical_volume_num=[ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED,LAST_SEED):
        numerical_volume=CalculateNumericalVolume_f(si, dimension)
        numerical_volume_num.append(numerical_volume)
    return numerical_volume_num


```

```
def AverageNumericalSolution_f():
    average_numerical_volume_num = [ ]
    dimension_num=[i for i in range(FIRST_DIMENSION, LAST_DIMENSION+1)]
    for dimension in range(FIRST_DIMENSION, LAST_DIMENSION+1):
        numerical_volume_num = NumericalSimulations_f(dimension)
        average_numerical_volume_num.append(np.average(numerical_volume_num))
    return dimension_num, average_numerical_volume_num


def AnalysisSolution_f():
    analysis_volume_num = [ ]
    for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
        analysis_volume = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2) + 1)
        analysis_volume_num.append(analysis_volume)
    return analysis_volume_num


dimension_num, average_numerical_volume_num = AverageNumericalSolution_f()
analysis_volume_num = AnalysisSolution_f()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num,average_numerical_volume_num, ":", label="numerical␣Solution",
  marker="o", color="tab:blue")
plt.plot(dimension_num,analysis_volume_num, ":", label="analysis␣Solution", marker="^"
  , color="tab:orange")
plt.title("dimension=2-20,␣NTRY=2**20,␣simulation=100")
plt.xlabel("Dimension")
plt.ylabel("Volume")
plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
plt.legend()
plt.show()

plt.figure(figsize=(6,4))
plt.plot(dimension_num, average_numerical_volume_num, ":", label="numerical␣Solution",
   marker="o", color="tab:blue")
plt.plot(dimension_num, analysis_volume_num, ":", label="analysis␣Solution", marker="^
  ", color="tab:orange")
plt.title("dimension=2-20,␣NTRY=2**20,␣simulation=100,␣*zoom*")
plt.xlabel("Dimension")
plt.ylabel("Volume")
plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
plt.xlim([9.9,20.2])
plt.ylim([-0.1,2])
plt.legend()
plt.show()
```

Code7

```
# Code7. 各次元ごとの超球体積における数値解と解析解をグラフ
  化. def, njit, dimension=100-110, NTRY=2**20,simulation=100. execution time_s=885s
import random as rd
import math as m
import numpy as np
from numba import njit

```

```
FIRST_DIMENSION: int = 100
LAST_DIMENSION: int = 110


@njit(cache=True)
def CalculateNumericalVolume_f(si, dimension):
    rd.seed(si)
    numerical_volume: float = 0.0
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume


def NumericalSimulations_f(dimension):
    numerical_volume_num=[ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED,LAST_SEED):
        numerical_volume=CalculateNumericalVolume_f(si, dimension)
        numerical_volume_num.append(numerical_volume)
    return numerical_volume_num


def AverageNumericalSolution_f():
    average_numerical_volume_num = [ ]
    inum=[i for i in range(FIRST_DIMENSION, LAST_DIMENSION+1)]
    for dimension in range(FIRST_DIMENSION, LAST_DIMENSION+1):
        numerical_volume_num = NumericalSimulations_f(dimension)
        average_numerical_volume_num.append(np.average(numerical_volume_num))
    return inum, average_numerical_volume_num


def AnalysisSolution_f():
    analysis_volume_num = [ ]
    for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
        analysis_volume = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2) + 1)
        analysis_volume_num.append(analysis_volume)
    return analysis_volume_num


inum, average_numerical_volume_num = AverageNumericalSolution_f()
analysis_volume_num = AnalysisSolution_f()

import matplotlib.pyplot as plt
plt.figure(figsize=(10,4))
plt.plot(inum,average_numerical_volume_num, ":", label="numerical␣Solution", marker="o
  ", color="tab:blue")
plt.plot(inum,analysis_volume_num, ":", label="analysis␣Solution", marker="^", color="
  tab:orange")
```

```
  plt.title("dimension=100-110, NTRY=2**20, simulation=100")
  plt.xlabel("Dimension")
  plt.ylabel("Volume")
  plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
  plt.legend()
  plt.show()
```

Code8

```
  # Code8*. 各次元ごとの解析解と数値解との相対誤差をグラフ化. dimension=2-20, NTRY=2**20, simulation=100. execution time_s=2.5s
import random as rd
import math as m
import numpy as np
from numba import njit

FIRST_DIMENSION: int = 2
LAST_DIMENSION: int = 20


@njit(cache=True)
def CalculateNumericalVolume_f(si, dimension):
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    rd.seed(si)
    for i in range(NTRY):
        sum: float = 0.0
        for _ in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume

@njit(cache=True)
def NumericalSimulations_f(dimension):
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED,LAST_SEED):
        numerical_volume = CalculateNumericalVolume_f(si, dimension)
        numerical_volume_num.append(numerical_volume)
        return numerical_volume_num

@njit(cache=True)
def CalculateRelativeError_f():
    dimension_num = [ ]
    relative_error_num = [ ]
    for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
        dimension_num.append(dimension)
        numerical_volume_num = NumericalSimulations_f(dimension)
        average_numerical_volume: float = np.average(numerical_volume_num)
        analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2) +
           1)
```

```
        # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
          average_numerical_volume = ", average_numerical_volume)
        relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
          analysis_volume)
        relative_error_num.append(relative_error)
        # print("dimension :", dimension, ", RelativeError = ", relative_error)
    return dimension_num, relative_error_num


def main():
    dimension_num, relative_error_num = CalculateRelativeError_f()
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

# print(relative_error_num)

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num,relative_error_num, ":", label="relative␣error", marker="v",
  color="tab:green")
plt.title("dimension=10-20,␣NTRY=2**30,␣simulation=100")
plt.xlabel("dimension")
plt.ylabel("relative␣error")
plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
plt.legend()
plt.show()
```

Code9

```
  # Code8*. 各次元ごとの解析解と数値解との相対誤差をグラフ
    化. dimension=2-20, NTRY=2**20, simulation=100. execution time_s=2.5s
  import random as rd
  import math as m
  import numpy as np
  from numba import njit

  FIRST_DIMENSION: int = 2
  LAST_DIMENSION: int = 20


  @njit(cache=True)
  def CalculateNumericalVolume_f(si, dimension):
      NTRY: int = 2 ** 20
      ncnt: float = 0.0
      rd.seed(si)
      for i in range(NTRY):
          sum: float = 0.0
          for _ in range(dimension):
              x: float = rd.random()
              sum += x * x
          if sum < 1.0:
              ncnt += 1.0
      numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
      return numerical_volume
```

```
25
26  @njit(cache=True)
27  def NumericalSimulations_f(dimension):
28      numerical_volume_num = [ ]
29      FIRST_SEED: int = 100
30      LAST_SEED: int = 200
31      for si in range(FIRST_SEED,LAST_SEED):
32          numerical_volume = CalculateNumericalVolume_f(si, dimension)
33          numerical_volume_num.append(numerical_volume)
34          return numerical_volume_num
35
36  @njit(cache=True)
37  def CalculateRelativeError_f():
38      dimension_num = [ ]
39      relative_error_num = [ ]
40      for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
41          dimension_num.append(dimension)
42          numerical_volume_num = NumericalSimulations_f(dimension)
43          average_numerical_volume: float = np.average(numerical_volume_num)
44          analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
                + 1)
45          # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
              average_numerical_volume = ", average_numerical_volume)
46          relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
              analysis_volume)
47          relative_error_num.append(relative_error)
48          # print("dimension :", dimension, ", RelativeError = ", relative_error)
49      return dimension_num, relative_error_num
50
51
52  def main():
53      dimension_num, relative_error_num = CalculateRelativeError_f()
54      return dimension_num, relative_error_num
55
56
57  dimension_num, relative_error_num = main()
58
59  # print(relative_error_num)
60
61  import matplotlib.pyplot as plt
62  plt.figure(figsize=(6,4))
63  plt.plot(dimension_num,relative_error_num, ":", label="relative␣error", marker="v",
      color="tab:green")
64  plt.title("dimension=10-20,␣NTRY=2**30,␣simulation=100")
65  plt.xlabel("dimension")
66  plt.ylabel("relative␣error")
67  plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
68  plt.legend()
69  plt.show()
```

Code10

```
1  # Code10. 次元のときの相対誤差の推移をグラフ化実行時間の確
     認15[], def, NTRY=2**10, simulation=1. execution time_s=0.494
2  import random as rd
3  import math as m
```

```
4
5
6   def main():
7       rd.seed(100)
8       DIMENSION: int = 15
9       NTRY: int = 2 ** 10
10      ncnt: float = 0.0
11      dimension_num = [ ]
12      relative_error_num = [ ]
13      delimiter: int = 1
14      for ni in range(1, NTRY):
15          sum: float = 0.0
16          for di in range(DIMENSION):
17              x: float = rd.random()
18              sum += x * x
19          if sum < 1:
20              ncnt += 1.0
21          if ni % delimiter == 0:
22              dimension_num.append(ni)
23              numerical_volume = ncnt / ni * 2 ** DIMENSION
24              analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION
                  / 2.0) + 1.0)
25              relative_error: float = (analysis_volume - numerical_volume) /
                  analysis_volume
26              relative_error_num.append(relative_error)
27      return dimension_num, relative_error_num
28
29
30  dimension_num, relative_error_num = main()
31
32  import matplotlib.pyplot as plt
33  plt.figure(figsize=(6,4))
34  plt.plot(dimension_num, relative_error_num, ":", label="relative␣error", color="tab:
      green")
35  plt.title("DIMENSION=15,␣NTRY=2**10,␣simulation=1")
36  plt.xlabel("NTRY")
37  plt.ylabel("relative␣error")
38  plt.legend()
39  plt.show()
```

Code11

```
1   # Code11. 次元のときの相対誤差の推移をグラフ化実行時間の確
      認15[], def, NTRY=2**15, simulation=1. execution time_s=0.961
2   import random as rd
3   import math as m
4
5
6   def main():
7       rd.seed(100)
8       DIMENSION: int = 15
9       NTRY: int = 2 ** 15
10      ncnt: float = 0.0
11      dimension_num = [ ]
12      relative_error_num = [ ]
13      delimiter: int = 1
```

```
    for ni in range(1, NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1:
            ncnt += 1.0
        if ni % delimiter == 0:
            dimension_num.append(ni)
            numerical_volume = ncnt / ni * 2 ** DIMENSION
            analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION
              / 2.0) + 1.0)
            relative_error: float = (analysis_volume - numerical_volume) /
              analysis_volume
            relative_error_num.append(relative_error)
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num, relative_error_num, ":", label="relative␣error", color="tab:
  green")
plt.title("DIMENSION=15,␣NTRY=2**15,␣simulation=1")
plt.xlabel("NTRY")
plt.ylabel("relative␣error")
plt.legend()
plt.show()
```

Code12

```
# Code12. 次元のときの相対誤差の推移をグラフ化実行時間の確
  認15[], def, NTRY=2**20, simulation=1. execution time_s=8.537
import random as rd
import math as m


def main():
    rd.seed(100)
    DIMENSION: int = 15
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    dimension_num = [ ]
    relative_error_num = [ ]
    delimiter: int = 1
    for ni in range(1, NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1:
            ncnt += 1.0
        if ni % delimiter == 0:
            dimension_num.append(ni)
            numerical_volume = ncnt / ni * 2 ** DIMENSION
```

```
            analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION / 2.0) + 1.0)
            relative_error: float = (analysis_volume - numerical_volume) / analysis_volume
            relative_error_num.append(relative_error)
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num, relative_error_num, ":", label="relative error", color="tab:green")
plt.title("DIMENSION=15, NTRY=2**20, simulation=1")
plt.xlabel("NTRY")
plt.ylabel("relative error")
plt.legend()
plt.show()
```

Code13

```
# Code13. 次元のときの相対誤差の推移をグラフ化実行時間の確認15[], def, NTRY=2**25, simulation=1. execution time_s=180.021
import random as rd
import math as m


def main():
    rd.seed(100)
    DIMENSION: int = 15
    NTRY: int = 2 ** 25
    ncnt: float = 0.0
    dimension_num = [ ]
    relative_error_num = [ ]
    delimiter: int = 1
    for ni in range(1, NTRY):
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1:
            ncnt += 1.0
        if ni % delimiter == 0:
            dimension_num.append(ni)
            numerical_volume = ncnt / ni * 2 ** DIMENSION
            analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION / 2.0) + 1.0)
            relative_error: float = (analysis_volume - numerical_volume) / analysis_volume
            relative_error_num.append(relative_error)
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num, relative_error_num, ":", label="relative error", color="tab:green")
plt.title("DIMENSION=15, NTRY=2**25, simulation=1")
plt.xlabel("NTRY")
plt.ylabel("relative error")
plt.legend()
plt.show()
```

CodeA

```
# CodeA. 次元のときの相対誤差の推移をグラフ化実行時間の確認15[], def, NTRY=2**30, simulation=1. execution time_s=?=4800
import random as rd
import math as m
from tqdm.auto import tqdm


def main():
    rd.seed(100)
    DIMENSION: int = 15
    NTRY: int = 2 ** 30
    ncnt: float = 0.0
    dimension_num = [ ]
    relative_error_num = [ ]
    delimiter: int = 1
    for ni in tqdm(range(1, NTRY)): # 関数で実行時間を推測tqdm
        sum: float = 0.0
        for di in range(DIMENSION):
            x: float = rd.random()
            sum += x * x
        if sum < 1:
            ncnt += 1.0
        if ni % delimiter == 0:
            dimension_num.append(ni)
            numerical_volume = ncnt / ni * 2 ** DIMENSION
            analysis_volume: float = (m.pi ** (DIMENSION / 2.0)) / m.gamma((DIMENSION / 2.0) + 1.0)
            relative_error: float = (analysis_volume - numerical_volume) / analysis_volume
            relative_error_num.append(relative_error)
    return dimension_num, relative_error_num


dimension_num, relative_error_num = main()

import matplotlib.pyplot as plt
plt.figure(figsize=(6,4))
plt.plot(dimension_num, relative_error_num, ":", label="relative error", color="tab:green")
plt.title("DIMENSION=15, NTRY=2**30, simulation=1")
plt.xlabel("NTRY")
plt.ylabel("relative error")
plt.legend()
plt.show()
```

Code14

```
# Code14. 次元の相対誤差の実行時間推移のグラフ15
NTRY_num=[2**i for i in range(10,31,5)]
execution_time_num=[0.494, 0.961, 8.537, 180.021, 4800]
import matplotlib.pyplot as plt
plt.figure(figsize=(10,4))
plt.plot(NTRY_num, execution_time_num, ":", marker="$T$", color="tab:red")
plt.title("DIMENSION=15, \"Relative Error\"")
plt.xlabel("NTRY")
plt.ylabel("execute time [s]")
plt.show()
```

CodeB1

```
# CodeB1. 次元球の体積を計
  算20. def, dimension=2, NTRY=2**20, simulation=100. execution time_s=69
import random as rd
import numpy as np


def CalculateNumericalVolume_f(dimension, si):
    rd.seed(si)
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume


def main():
    dimension: int = 2
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED, LAST_SEED):
        numerical_volume = CalculateNumericalVolume_f(dimension, si)
        numerical_volume_num.append(numerical_volume)
    average_numerical_volume: float = np.average(numerical_volume_num)
    return dimension, average_numerical_volume


dimension, average_numerical_volume = main()

print("d=", dimension, ",V=", average_numerical_volume, "pi=", np.pi)
```

CodeB2

```
# CodeB2. 次元球の体積を計算20. def, njit, dimension=2, NTRY=2**20, simulation=100. execution time_s=1.9
import random as rd
import numpy as np
from numba import njit


@njit(cache=True)
def CalculateNumericalVolume_f(dimension, si):
    rd.seed(si)
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    for ni in range(NTRY):
        sum: float = 0.0
        for di in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume


@njit(cache=True)
def main():
    dimension: int = 2
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED, LAST_SEED):
        numerical_volume = CalculateNumericalVolume_f(dimension, si)
        numerical_volume_num.append(numerical_volume)
    average_numerical_volume: float = np.average(numerical_volume_num)
    return dimension, average_numerical_volume


dimension, average_numerical_volume = main()

print("d=", dimension, ",V=", average_numerical_volume, "pi=", np.pi)
```

CodeC1

```
# CodeC1. 各次元ごとの解析解と数値解との相対誤差をグラフ化. NTRY=2**10, simulation=100. execution time_s=1.3
import random as rd
import math as m
import numpy as np
from numba import njit

FIRST_DIMENSION: int = 10
LAST_DIMENSION: int = 20

```

```
10
11  @njit(cache=True)
12  def CalculateNumericalVolume_f(si, dimension):
13      NTRY: int = 2 ** 10
14      ncnt: float = 0.0
15      rd.seed(si)
16      for i in range(NTRY):
17          sum: float = 0.0
18          for _ in range(dimension):
19              x: float = rd.random()
20              sum += x * x
21          if sum < 1.0:
22              ncnt += 1.0
23      numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
24      return numerical_volume
25
26  @njit(cache=True)
27  def NumericalSimulations_f(dimension):
28      numerical_volume_num = [ ]
29      FIRST_SEED: int = 100
30      LAST_SEED: int = 200
31      for si in range(FIRST_SEED,LAST_SEED):
32          numerical_volume = CalculateNumericalVolume_f(si, dimension)
33          numerical_volume_num.append(numerical_volume)
34          return numerical_volume_num
35
36  @njit(cache=True)
37  def CalculateRelativeError_f():
38      dimension_num = [ ]
39      relative_error_num = [ ]
40      for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
41          dimension_num.append(dimension)
42          numerical_volume_num = NumericalSimulations_f(dimension)
43          average_numerical_volume: float = np.average(numerical_volume_num)
44          analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
                + 1)
45          # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
              average_numerical_volume = ", average_numerical_volume)
46          relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
              analysis_volume)
47          relative_error_num.append(relative_error)
48          # print("dimension :", dimension, ", RelativeError = ", relative_error)
49      return dimension_num, relative_error_num
50
51
52  def main():
53      dimension_num, relative_error_num = CalculateRelativeError_f()
54      return dimension_num, relative_error_num
55
56
57  dimension_num, relative_error_num = main()
58
59  # print(relative_error_num)
60
61  import matplotlib.pyplot as plt
62  plt.figure(figsize=(6,4))
```

```
63  plt.plot(dimension_num,relative_error_num, ":", label="relative␣Error", marker="v",
      color="tab:green")
64  plt.title("dimension=10-20,␣NTRY=2**30,␣simulation=100")
65  plt.xlabel("dimension")
66  plt.ylabel("relative␣error")
67  plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
68  plt.legend()
69  plt.show()
```

CodeC2

```
1   # CodeC2. 各次元ごとの解析解と数値解との相対誤差をグラフ
      化. dimesnsion=10-20, NTRY=2**15, simulation=100. execution time_s=1.3
2   import random as rd
3   import math as m
4   import numpy as np
5   from numba import njit
6
7   FIRST_DIMENSION: int = 10
8   LAST_DIMENSION: int = 20
9
10
11  @njit(cache=True)
12  def CalculateNumericalVolume_f(si, dimension):
13      NTRY: int = 2 ** 15
14      ncnt: float = 0.0
15      rd.seed(si)
16      for i in range(NTRY):
17          sum: float = 0.0
18          for _ in range(dimension):
19              x: float = rd.random()
20              sum += x * x
21          if sum < 1.0:
22              ncnt += 1.0
23      numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
24      return numerical_volume
25
26  @njit(cache=True)
27  def NumericalSimulations_f(dimension):
28      numerical_volume_num = [ ]
29      FIRST_SEED: int = 100
30      LAST_SEED: int = 200
31      for si in range(FIRST_SEED,LAST_SEED):
32          numerical_volume = CalculateNumericalVolume_f(si, dimension)
33          numerical_volume_num.append(numerical_volume)
34          return numerical_volume_num
35
36  @njit(cache=True)
37  def CalculateRelativeError_f():
38      dimension_num = [ ]
39      relative_error_num = [ ]
40      for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
41          dimension_num.append(dimension)
42          numerical_volume_num = NumericalSimulations_f(dimension)
43          average_numerical_volume: float = np.average(numerical_volume_num)
```

```
            analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
                + 1)
            # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
              average_numerical_volume = ", average_numerical_volume)
            relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
              analysis_volume)
            relative_error_num.append(relative_error)
            # print("dimension :", dimension, ", RelativeError = ", relative_error)
        return dimension_num, relative_error_num


    def main():
        dimension_num, relative_error_num = CalculateRelativeError_f()
        return dimension_num, relative_error_num


    dimension_num, relative_error_num = main()

    # print(relative_error_num)

    import matplotlib.pyplot as plt
    plt.figure(figsize=(6,4))
    plt.plot(dimension_num,relative_error_num, ":", label="relative␣error", marker="v",
      color="tab:green")
    plt.title("dimension=10-20,␣NTRY=2**30,␣simulation=100")
    plt.xlabel("dimension")
    plt.ylabel("relative␣Error")
    plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
    plt.legend()
    plt.show()
```

CodeC3

```
# CodeC3. 各次元ごとの解析解と数値解との相対誤差をグラフ化. dimesnsion=10-20, NTRY=2**20, simulation=100. execution time_s=2.4
import random as rd
import math as m
import numpy as np
from numba import njit

FIRST_DIMENSION: int = 10
LAST_DIMENSION: int = 20


@njit(cache=True)
def CalculateNumericalVolume_f(si, dimension):
    NTRY: int = 2 ** 20
    ncnt: float = 0.0
    rd.seed(si)
    for i in range(NTRY):
        sum: float = 0.0
        for _ in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
```

```
        numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
        return numerical_volume

    @njit(cache=True)
    def NumericalSimulations_f(dimension):
        numerical_volume_num = [ ]
        FIRST_SEED: int = 100
        LAST_SEED: int = 200
        for si in range(FIRST_SEED,LAST_SEED):
            numerical_volume = CalculateNumericalVolume_f(si, dimension)
            numerical_volume_num.append(numerical_volume)
            return numerical_volume_num

    @njit(cache=True)
    def CalculateRelativeError_f():
        dimension_num = [ ]
        relative_error_num = [ ]
        for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
            dimension_num.append(dimension)
            numerical_volume_num = NumericalSimulations_f(dimension)
            average_numerical_volume: float = np.average(numerical_volume_num)
            analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
               + 1)
            # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
              average_numerical_volume = ", average_numerical_volume)
            relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
              analysis_volume)
            relative_error_num.append(relative_error)
            # print("dimension :", dimension, ", RelativeError = ", relative_error)
        return dimension_num, relative_error_num


    def main():
        dimension_num, relative_error_num = CalculateRelativeError_f()
        return dimension_num, relative_error_num


    dimension_num, relative_error_num = main()

    # print(relative_error_num)

    import matplotlib.pyplot as plt
    plt.figure(figsize=(6,4))
    plt.plot(dimension_num,relative_error_num, ":", label="relative␣Error", marker="v",
      color="tab:green")
    plt.title("dimension=10-20,␣NTRY=2**15,␣simulation=100")
    plt.xlabel("dimension")
    plt.ylabel("relative␣error")
    plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
    plt.legend()
    plt.show()
```

CodeC4

```
# CodeC4. 各次元ごとの解析解と数値解との相対誤差をグラフ
  化. dimesnsion=10-20, NTRY=2**25, simulation=100. execution time_s=2.7
```

```
import random as rd
import math as m
import numpy as np
from numba import njit

FIRST_DIMENSION: int = 10
LAST_DIMENSION: int = 20


@njit(cache=True)
def CalculateNumericalVolume_f(si, dimension):
    NTRY: int = 2 ** 25
    ncnt: float = 0.0
    rd.seed(si)
    for i in range(NTRY):
        sum: float = 0.0
        for _ in range(dimension):
            x: float = rd.random()
            sum += x * x
        if sum < 1.0:
            ncnt += 1.0
    numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
    return numerical_volume

@njit(cache=True)
def NumericalSimulations_f(dimension):
    numerical_volume_num = [ ]
    FIRST_SEED: int = 100
    LAST_SEED: int = 200
    for si in range(FIRST_SEED,LAST_SEED):
        numerical_volume = CalculateNumericalVolume_f(si, dimension)
        numerical_volume_num.append(numerical_volume)
        return numerical_volume_num

@njit(cache=True)
def CalculateRelativeError_f():
    dimension_num = [ ]
    relative_error_num = [ ]
    for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
        dimension_num.append(dimension)
        numerical_volume_num = NumericalSimulations_f(dimension)
        average_numerical_volume: float = np.average(numerical_volume_num)
        analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
           + 1)
        # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
          average_numerical_volume = ", average_numerical_volume)
        relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
          analysis_volume)
        relative_error_num.append(relative_error)
        # print("dimension :", dimension, ", RelativeError = ", relative_error)
    return dimension_num, relative_error_num


def main():
    dimension_num, relative_error_num = CalculateRelativeError_f()
    return dimension_num, relative_error_num

```

```
56
57  dimension_num, relative_error_num = main()
58
59  # print(relative_error_num)
60
61  import matplotlib.pyplot as plt
62  plt.figure(figsize=(6,4))
63  plt.plot(dimension_num,relative_error_num, ":", label="relative␣Error", marker="v",
      color="tab:green")
64  plt.title("dimension=10-20,␣NTRY=2**25,␣simulation=100")
65  plt.xlabel("dimension")
66  plt.ylabel("relative␣error")
67  plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
68  plt.legend()
69  plt.show()
```

CodeC5

```
1   # CodeC5. 各次元ごとの解析解と数値解との相対誤差をグラフ
      化. dimesion=10-20, NTRY=2**30, simulation=100. execution time_s=1195
2   import random as rd
3   import math as m
4   import numpy as np
5   from numba import njit
6
7   FIRST_DIMENSION: int = 10
8   LAST_DIMENSION: int = 20
9
10
11  @njit(cache=True)
12  def CalculateNumericalVolume_f(si, dimension):
13      NTRY: int = 2 ** 30
14      ncnt: float = 0.0
15      rd.seed(si)
16      for i in range(NTRY):
17          sum: float = 0.0
18          for _ in range(dimension):
19              x: float = rd.random()
20              sum += x * x
21          if sum < 1.0:
22              ncnt += 1.0
23      numerical_volume: float = ncnt / NTRY * 2.0 ** dimension
24      return numerical_volume
25
26  @njit(cache=True)
27  def NumericalSimulations_f(dimension):
28      numerical_volume_num = [ ]
29      FIRST_SEED: int = 100
30      LAST_SEED: int = 200
31      for si in range(FIRST_SEED,LAST_SEED):
32          numerical_volume = CalculateNumericalVolume_f(si, dimension)
33          numerical_volume_num.append(numerical_volume)
34          return numerical_volume_num
35
36  @njit(cache=True)
37  def CalculateRelativeError_f():
```

```
38      dimension_num = [ ]
39      relative_error_num = [ ]
40      for dimension in range(FIRST_DIMENSION,LAST_DIMENSION+1):
41          dimension_num.append(dimension)
42          numerical_volume_num = NumericalSimulations_f(dimension)
43          average_numerical_volume: float = np.average(numerical_volume_num)
44          analysis_volume: float = ((m.pi) ** (dimension / 2)) / m.gamma((dimension / 2)
                + 1)
45          # print("dimension :", dimension, ", analysis_volume = ", analysis_volume, ",
              average_numerical_volume = ", average_numerical_volume)
46          relative_error: float = m.fabs((analysis_volume - average_numerical_volume) /
              analysis_volume)
47          relative_error_num.append(relative_error)
48          # print("dimension :", dimension, ", RelativeError = ", relative_error)
49      return dimension_num, relative_error_num
50
51
52  def main():
53      dimension_num, relative_error_num = CalculateRelativeError_f()
54      return dimension_num, relative_error_num
55
56
57  dimension_num, relative_error_num = main()
58
59  # print(relative_error_num)
60
61  import matplotlib.pyplot as plt
62  plt.figure(figsize=(6,4))
63  plt.plot(dimension_num,relative_error_num, ":", label="relative␣error", marker="v",
      color="tab:green")
64  plt.title("dimension=10-20,␣NTRY=2**30,␣simulation=100")
65  plt.xlabel("dimension")
66  plt.ylabel("relative␣error")
67  plt.xticks([i for i in range(FIRST_DIMENSION,LAST_DIMENSION+1)])
68  plt.legend()
69  plt.show()
```

CodeC6

```
1   # CodeC6. 各次元各サンプル点の総数ごとに解析解と数値解との相対誤差
      を,化tabel. NTRY=2*10-2**30, simulation=100. execution time_s=0.3
2   import pandas as pd
3
4   i10_num = [0.033263732104414294, 0.0649703809235029, 0.016378123412200834,
      0.09830579924799727, 0.09292559406626424, 0.17985355796824742, 0.32244517119083377,
      0.09049813462405956, 0.21573672566718738, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0]
5   i15_num = [0.0006633909779249858, 0.0033387963117393043, 0.00529611895132014,
      0.0007643522382241023, 0.015923334645784038, 0.027663334118415768,
      0.0009332698317228344, 0.1321403428377592, 0.15188668417631854, 0.09468486454572125,
       0.34469827182361296, 0.6472135237558793, 0.1656439259525315, 1.0, 1.0, 1.0, 1.0,
      1.0, 1.0]
6   i20_num = [2.9549422947802508e-05, 0.0015246980681839174, 0.001526876920680194,
      0.0001208136839772289 2, 0.0010652669027548783, 0.0012845060780169728,
      0.005805601253379658, 0.0024197475962788734, 0.010863262889816889,
      0.0017220411387030559, 0.029759858562893943, 0.10672158627348134,
```

```
    0.1134966713245647, 0.18074320427010196, 0.46883242606639175, 0.7732872543477504,
    1.0, 1.0, 1.0]
i25_num = [0.00012052501131553322, 0.00016623036051146455, 0.00018945693129115843,
    0.000189903833064828 5, 0.0006537320872372532, 0.0017471367240486976,
    0.002277296439162892, 0.004033474562531739, 0.006369084446597451,
    0.010417504735911907, 0.0039742845310076795, 0.01798010249040907 6,
    0.008723456709732444, 0.01383028471574883, 0.07875623895889816, 0.14106398617530844,
     0.046157069395586316, 0.3405803022436725, 1.421833728652197]
i30_num = [1.951267208688593e-06, 2.2111590329099127e-05, 3.9482131301577615e-05,
    0.0001037831028682182, 2.767637215597141e-05, 0.0001384313425900941,
    0.0002752380089617943, 5.11085937674066e-05, 0.0005769576416525477,
    0.0019791748127185524, 0.0005745868445457106, 0.0006696844149080387,
    0.0022254297600066758, 0.018091928320992313, 0.04944865309244419,
    0.08218530780829328, 0.04597608160232043, 0.07835104220747517, 0.09739340829552663]

del i10_num[:8]
del i15_num[:8]
del i20_num[:8]
del i25_num[:8]
del i30_num[:8]

i10_num = [round(i,3) for i in i10_num]
i15_num = [round(i,3) for i in i15_num]
i20_num = [round(i,3) for i in i20_num]
i25_num = [round(i,3) for i in i25_num]
i30_num = [round(i,3) for i in i30_num]

NTRY_num=[i10_num, i15_num, i20_num, i25_num, i30_num]
df = pd.DataFrame(NTRY_num)
df
```

Code15

```
# Code15. 各次元各サンプル点の総数ごとに解析解と数値解との相対誤差をグラフ
  化,. NTRY=2*10-2**30, simulation=100. execution time_s=0.5

i10_num = [0.033263732104414294, 0.0649703809235029, 0.016378123412200834,
    0.09830579924799727, 0.09292559406626424, 0.17985355796824742, 0.32244517119083377,
    0.09049813462405956, 0.21573672566718738, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0]
i15_num = [0.0006633909779249858, 0.0033387963117393043, 0.00529611895132014,
    0.0007643522382241023, 0.015923334645784038, 0.027663334118415768,
    0.0009332698317228344, 0.1321403428377592, 0.15188668417631854, 0.09468486454572125,
     0.34469827182361296, 0.6472135237558793, 0.1656439259525315, 1.0, 1.0, 1.0, 1.0,
    1.0, 1.0]
i20_num = [2.9549422947802508e-05, 0.0015246980681839174, 0.001526876920680194,
    0.00012081368397722892, 0.0010652669027548783, 0.0012845060780169728,
    0.005805601253379658, 0.0024197475962788734, 0.010863262889816889,
    0.0017220411387030559, 0.029759858562893943, 0.10672158627348134,
    0.1134966713245647, 0.18074320427010196, 0.46883242606639175, 0.7732872543477504,
    1.0, 1.0, 1.0]
i25_num = [0.00012052501131553322, 0.00016623036051146455, 0.00018945693129115843,
    0.000189903833064828 5, 0.0006537320872372532, 0.0017471367240486976,
    0.002277296439162892, 0.004033474562531739, 0.006369084446597451,
    0.010417504735911907, 0.0039742845310076795, 0.01798010249040907 6,
    0.008723456709732444, 0.01383028471574883, 0.07875623895889816, 0.14106398617530844,
```

```
    0.046157069395586316, 0.3405803022436725, 1.421833728652197]
i30_num = [1.951267208688593e-06, 2.2111590329099127e-05, 3.9482131301577615e-05,
  0.0001037831028682182, 2.767637215597141e-05, 0.0001384313425900941,
  0.0002752380089617943, 5.11085937674066e-05, 0.0005769576416525477,
  0.0019791748127185524, 0.0005745868445457106, 0.0006696844149080387,
  0.0022254297600066758, 0.018091928320992313, 0.04944865309244419,
  0.08218530780829328, 0.04597608160232043, 0.07835104220747517, 0.09739340829552663]
i_num = [i for i in range(2,21)]

import matplotlib.pyplot as plt
plt.figure(figsize=(10,4))
plt.plot(i_num, i10_num, ":", label="NTRY=2**10", marker="v")
plt.plot(i_num, i15_num, ":", label="NTRY=2**15", marker="v")
plt.plot(i_num, i20_num, ":", label="NTRY=2**20", marker="v")
plt.plot(i_num, i25_num, ":", label="NTRY=2**25", marker="v")
plt.plot(i_num, i30_num, ":", label="NTRY=2**30", marker="v")
plt.xlabel("dimension")
plt.ylabel("relative␣error")
plt.xticks([i for i in range(2,21)])
plt.legend()
plt.show()
```

参考文献

[1] 早川美徳, 東北大学, 『MC による積分』.
https://wagtail.cds.tohoku.ac.jp/coda/python/stochastic-methods/mc-simu-int.html

[2] Mark Summerfield, オライリー・ジャパン, 実践 Python 3.

[3] 津田孝夫, 培風館, モンテカルロ法とシミュレーション [三訂版].