DxMT 東京科学大学拠点 D²MatE 2026年度チュートリアルシリーズ
東京科学大学 総合研究院 フロンティア材料研究所学術講演会

生成AIとToyプログラムで理解するバンド理論・データ解析

第２回第４部　pythonプログラムの高速化

神谷利夫

東京科学大学
総合研究院
元素戦略MDX研究センター

Institute of SCIENCE TOKYO

本チュートリアルには、
Microsoft365 Copilot / Chat GPT / Gemini / Claude等
を使用した結果が含まれています

プログラムに要求される性能

1. 正確性
2. 速度 (pythonでは優先順位は低い)
   - 最適化：無駄な処理を減らし、パフォーマンスを向上。
   - リソース管理：CPUやメモリの使用量を抑える。
3. 保守性（メンテナンス性）
   - コードの可読性：他の開発者が理解しやすいコードを書くこと。
   - 適切なコメント：意図が明確になるように記述。
   - モジュール化：変更が容易になるように、機能を分割。
4. 汎用性
   - 設定ファイル、起動時引数の活用：環境に依存せず動作できるようにする。
   - 柔軟な設計：さまざまなユースケースに対応できる構造を持たせる。
5. 例外耐性 (堅牢性)
   - 適切なエラーハンドリング：エラーを捕捉し、適切に処理する。
   - フェイルセーフ設計：予期しない障害に強い構成にする。
6. 再利用性
   - ライブラリ化：共通機能をモジュールとして分け、再利用を促進。
   - DRY（Don't Repeat Yourself）原則：同じコードを何度も書かない。

高速化の手順

1. 現在の計算結果と実行時間を記録し、基準にする。
2. **生成AIと相談して高速化する**
   不要な計算・コピーの削減、NumPyの利用など。
3. **比較テストとベンチマークを行う**
   結果が許容誤差内で一致し、実際に速くなったことを確認する。
4. **まだ遅ければプロファイリングする**
   時間を使っている関数を特定する。
5. **重い関数を改善し、３に戻る**

プロファイリング

時間計測とボトルネック検出

cProfile: 時間計測とボトルネック

cProfile: 時間を使う関数と呼び出し回数を確認する

cProfileの実行方法: > python -m cProfile -s cumulative Ne-T_sol.py

sort指定	並べる基準	用途
"cumulative" または "cumtime"	呼出先を含む合計時間	どの処理全体が重いかを探す
"tottime" または "time"	呼出先を除く、関数自身の時間	どの関数内部が重いかを探す
"calls"	呼出回数	呼び過ぎている関数を探す
"name"	関数名	名前順で確認する
"filename"	ファイル名	ファイルごとに確認する

プロファイリング例: ChatGPT-6.1 Sol版 Ne-T_sol.py

> python -m cProfile -s cumulative Ne-T_sol.py

Ordered by: cumulative time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
85	0.004	0.000	15.530	0.183	__init__.py:1(<module>)
602/9	0.001	0.000	7.387	0.821	text.py:65(_get_text_metrics_with_cache)
13/5	0.089	0.007	7.387	1.477	text.py:73(_get_text_metrics_with_cache_impl)
910/10	0.008	0.000	6.129	0.613	<frozen importlib._bootstrap>:1349(_find_and_load)
903/10	0.004	0.000	6.128	0.613	<frozen importlib._bootstrap>:1304(_find_and_load_unlocked)

> python -m cProfile -s tottime Ne-T_sol.py

Ordered by: internal time

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	1.947	1.947	1.973	1.973	{built-in method exec}
454356	0.523	0.000	0.561	0.000	Ne-T_sol.py:70(integrand)
753	0.308	0.000	0.308	0.000	{built-in method _io.open_code}
2	0.153	0.076	0.153	0.076	{built-in method _wmi.exec_query}

プロファイリング例: ChatGPT-6.1 Sol版 Ne-T_sol.py

CLI出力を grep して実行プログラムの関数だけを表示する

PowerShell: **> python -m cProfile -s cumulative Ne-T_sol.py | Select-String -SimpleMatch "Ne-T_sol.py:"**

Linux : **% python -m cProfile -s cumulative Ne-T_sol.py | grep -F 'Ne-T_sol.py:'**

(Ordered by: internal time)

(ncalls	tottime	percall	cumtime	percall	filename:lineno(function))
4148	0.013	0.000	0.680	0.000	Ne-T_sol.py:55(Fj)
112	0.001	0.000	0.665	0.006	Ne-T_sol.py:106(solve)
1962	0.001	0.000	0.659	0.000	Ne-T_sol.py:110(q)
8296	0.003	0.000	0.659	0.000	Ne-T_sol.py:79(<genexpr>)
1962	0.002	0.000	0.658	0.000	Ne-T_sol.py:102(deltaQ)
454356	0.504	0.000	0.540	0.000	Ne-T_sol.py:70(integrand)
2074	0.002	0.000	0.343	0.000	Ne-T_sol.py:88(Nh)
2074	0.002	0.000	0.343	0.000	Ne-T_sol.py:84(Ne)

プログラムの例: Ne-T_sol.pyの計算部分をcProfileで測定する。元コードの変更不要

> python profile_Ne-T.py Ne-T_sol.py

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
4148	0.013	0.000	0.679	0.000	Ne-T_sol.py:55(Fj)
112	0.001	0.000	0.663	0.006	Ne-T_sol.py:106(solve)
8296	0.003	0.000	0.658	0.000	Ne-T_sol.py:79(<genexpr>)
1962	0.001	0.000	0.658	0.000	Ne-T_sol.py:110(q)

perf_counter(): 経過時間計測

cProfile　: プログラム全体から、時間のかかる関数を見つける
Timeit　　: 特定の処理について、書き方や実装による速度差を測る

```
from time import perf_counter

start = perf_counter()

rows = []
for T in temperatures:
    Ef = solve(T, model)
    n = Ne(Ef, T, model)
    p = Nh(Ef, T, model)
    rows.append((T, Ef, n, p))

elapsed = perf_counter() - start
print(f"計算時間: {elapsed:.6f} 秒")
```

```
import timeit

t1 = timeit.repeat(
    "sum([i*i for i in range(1000)])",
    repeat=5,
    number=1000,
)
t2 = timeit.repeat(
    "sum(i*i for i in range(1000))",
    repeat=5,
    number=1000,
)

print("リスト内包表記:", min(t1) / 1000, "秒/回")
print("ジェネレータ式:", min(t2) / 1000, "秒/回")
```

高速化

高速化の方法

計算の見直し	高速アルゴリズムへの置き換え 重複計算の削減 (キャッシュ化) 大きなデータのコピー、複雑なオブジェクトの作成を削減
実装の改善	NumPy: ndarray、ベクトル・行列計算 キャッシュ: 同じ引数の計算結果を再利用
コンパイル	Numba: Just-in-time コンパイル C関数との連携 Nuitkaによるコンパイル (実行可能ファイルで配布)

高速化の方法: 優先順

1. アルゴリズム、必要精度の確認・選択
2. 線形代数・FFT・最適化などは、numpy, scipyを使う。
   その他、最適化・コンパイルされたライブラリを使う。
3. 純pythonのforループは非常に遅い。
   できる限りnp.sum()などの関数を使う。
   ufunc (universal function:引数として ndarray を受け取り、一括処理する関数) 化
   numbaの@vectorize()を使うとコンパイルによる高速化も可
4. 事前計算結果を再利用 (キャッシュ)
5. Numbaの@jitを使って実行中コンパイルを行う
6. GPUを使う
7. コードの一部をC言語で書き直す (C DLL)。
8. Nuitkaをつかってコンパイル・実行可能ファイルを作成

このうち、2~5は大きな修正が不要

Python科学技術計算の高速化：8つの手順

1 測定する

cProfile / line_profiler / timeit

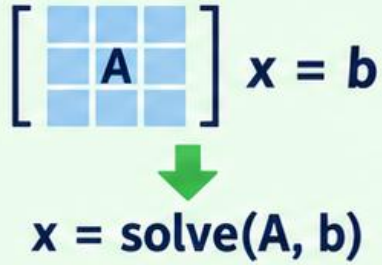
実行時間・メモリ・ボトルネックを確認



2 計算量を減らす

不変量をループ外へ／対称性／早期終了

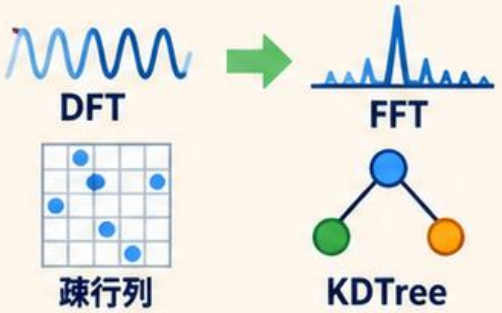
inv(A) @ b より solve(A, b)



3 アルゴリズムとデータ構造

DFTよりFFT／密行列より疎行列

set・dict・KDTreeで検索を高速化



4 Pythonループを減らす

sum()・map()・NumPy ufunc

np.sum / np.mean / @ でまとめて処理

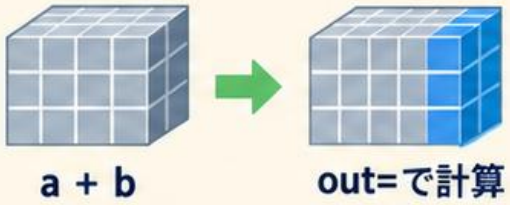
```
s = 0
for x in a:
    s += x
```



5 メモリ移動を減らす

out=・インプレース演算・view

一時配列・コピー・巨大broadcastを避ける



6 dtypeと精度を調整

float64 → float32／int64 → int32

許容誤差と混合精度を検討



7 I/Oと表示を減らす

頻繁・大きなファイル読み書きを避ける

大量のconsole出力・毎回のplot更新を減らす



8 並列化・GPU・コンパイル

multiprocessing / joblib / BLAS

CuPy・GPU・Numba・Cythonを必要な箇所へ



基本方針： まず測定し、計算とデータ移動を減らす。並列化・GPU化はその後。

不要なコピーとメモリ確保の削減

オーバーヘッドの大きい操作

- 大きな配列やオブジェクトのコピー
  - 配列や文字列の連結の際はコピーが発生する
- 型変換
- 複雑なオブジェクトの作成

オーバーヘッド削減の方法

- 型固定の場合はndarrayを使う
- 大きな作業領域が必要な場合、作業配列を再利用する

文字列連結

strings_sum_benchmark.py

```
def concat_plus(n: int) -> str:
    result = ""
    for _ in range(n):
        result = result + CHUNK
    return result

def concat_iadd(n: int) -> str:
    result = ""
    for _ in range(n):
        result += CHUNK
    return result
```

```
def list_join(n: int) -> str:
    parts: list[str] = []
    for _ in range(n):
        parts.append(CHUNK)
    return "".join(parts)

def string_io(n: int) -> str:
    out = StringIO()
    for _ in range(n):
        out.write(CHUNK)
    return out.getvalue()
```

> Python strings_sum_benchmark.py

chunk = '#' * 80, count = 100,000, output = 7.63 MiB

method	median	minimum
s = s + chunk	0.931520 s	min= 0.926120 s
s += chunk	0.943046 s	min= 0.937524 s
list.append + join	0.003262 s	min= 0.003142 s
StringIO.write	0.007793 s	min= 0.007774 s

ベクトル化: 普通の関数をufunc化 & コンパイル

```
import math
import numpy as np
from numba import vectorize

@vectorize(["float64(float64)"])
def f(x):
    return math.sin(x) + x*x + 1.0

x = np.array([0.1, 0.2, 0.3])
y = f(x)
```

Numba @vectorizeはコンパイルもするため、高速化される可能性がある
np.frompyfunc(pyfunc, 1, 1), np.vectorize(pyfunc, otypes=[np.float64])も
スカラー関数pyfuncを配列の各要素に適用できる。
ただし、高速化が保証されるわけではない

ufunc化: ufunc_benchmark.py

> python ufunc_benchmark.py -n 1000000 --repeat 5

Startup/warmup (separate from trials; Numba includes JIT):

Python for	0.150598 s
list comprehension	0.137998 s
NumPy ufunc	0.008650 s
np.vectorize	0.127658 s
Numba @vectorize	0.912394 s
Numba @njit	0.155935 s

method	median(s)	minimum(s)	speedup	max.abs.err	match
Python for	0.143478	0.142806	1.00x	0.000e+00	True
list comprehension	0.136096	0.134761	1.05x	0.000e+00	True
NumPy ufunc	0.008925	0.008817	16.08x	0.000e+00	True
np.vectorize	0.127787	0.126511	1.12x	0.000e+00	True
Numba @vectorize	0.004517	0.004415	31.77x	0.000e+00	True
Numba @njit	0.004734	0.004703	30.31x	0.000e+00	True

事前計算のキャッシュ: @lru_cache

@lru_cache: 同じ引数の呼出結果を一定件数まで保持して再利用する

- 結果が引数だけで決まり、同じ入力が再登場する関数に向く
  - 引数はハッシュ可能である必要がある
  - 実数引数は、丸目誤差を許容できる場合のみ、適当な桁で丸めてキャッシュ化できる
- cache_infoでヒット率を確認し、時間とメモリの効果を測る

注意:

- 外部状態や入力ファイルが変わる計算では、古い結果の再利用に注意
- 返された配列などを変更すると、保存済みの結果にも影響し得る
- 件数上限を設け、条件変更時は必要に応じてcache_clearを使う

@lru_cacheの例

```
from functools import lru_cache
from math import exp
from time import perf_counter
from scipy.integrate import quad

def integral_raw(a: float) -> float:
    """∫[0, 1] exp(-a*x²) dx"""
    value, _ = quad(lambda x: exp(-a * x * x), 0.0, 1.0)
    return value

@lru_cache(maxsize=128)
def integral_cached(a: float) -> float:
    return integral_raw(a)

def integral(a: float, use_cache: bool = True) -> float:
#    a = float(a)
    a = round(float(a), 10)        # 実数引数を丸めても問題ない場合
    if use_cache:
        return integral_cached(a)
    return integral_raw(a)
```

```
integral_cached.cache_clear()
integral(a, use_cache=True)

integral(a, use_cache=False)
```

@lru_cacheの例: lru_cache_benchmark.py

exp(x)を100回計算する関数の実行時間を計測。cacheする場合は小数点以下第3位で丸め

> python lru_cache_benchmark.py

N=100,000, repeat=3, exp calls per evaluation=100
cache key: round(x, 3)

case	requests	actual evaluations	median time
no cache	300,000	300,000	1.900788 s
lru_cache + rounded key	300,000	3,003	0.097058 s

lru_cache info (one run): hits=98,999, misses=1,001, currsize=1,001
speedup (median): 19.58 x

Numbaによる数値ループのコンパイル

@njitで、型の安定した数値計算関数をコンパイルする

- 分岐や多重ループなど、配列演算にしにくい処理にも使える
- 入出力・GUI・複雑なPythonオブジェクト操作はループの外に出す

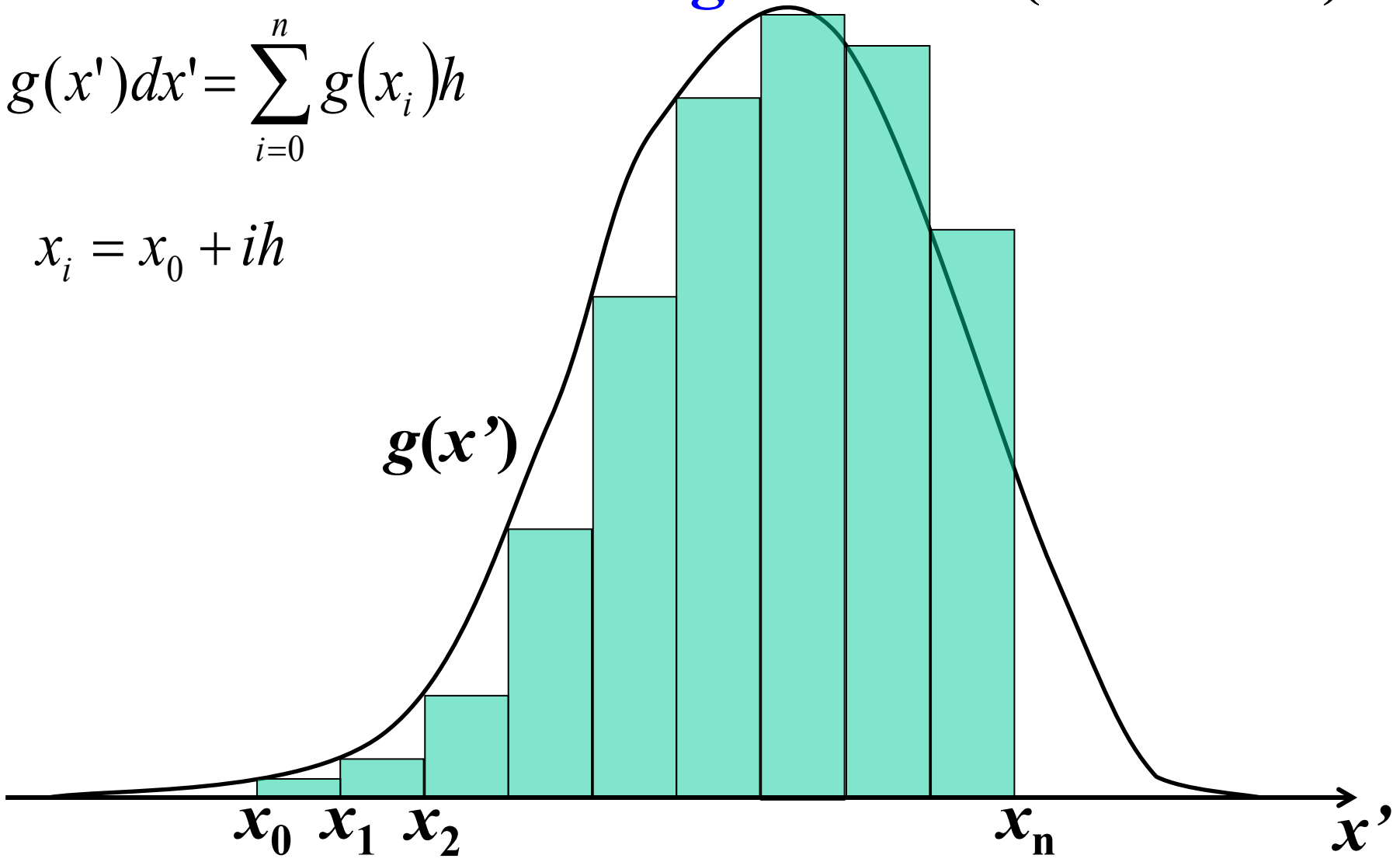
初回実行時はコンパイルの時間がかかる

- @njitの効果を調べる際は、初回実行時と、2回目以降の実行時間を分けて計測

Riemann sum / rectangular rule (Riemann和)

$$\int_{x_0}^{x} g(x')dx' = \sum_{i=0}^{n} g(x_i)h$$

$$x_i = x_0 + ih$$



Asymmetric formula:

monotone increasing $g(x)$ => Underestimation (過小評価)

monotone decreasing $g(x)$ => Overestimation (過大評価)

Riemann積分の比較

```
> Python pi_riemann_benchmark.py
Numba JIT warm-up: 0.323677 s
Midpoint Riemann sum: nmax=1,000,000, repeat=3
for         pi=3.14159265393  abs.error=3.445e-10  median=0.069430 s  min=0.069118 s
generator   pi=3.14159265393  abs.error=3.444e-10  median=0.108990 s  min=0.108287 s
listcomp    pi=3.14159265393  abs.error=3.444e-10  median=0.124552 s  min=0.123914 s
numpy       pi=3.14159265393  abs.error=3.444e-10  median=0.007117 s  min=0.007014 s
numba       pi=3.14159265393  abs.error=3.445e-10  median=0.002242 s  min=0.001868 s
cupy        pi=3.14159265393  abs.error=3.444e-10  median=0.000190 s  min=0.000182 s

C exe version:
> cl /O2 /std:c11 pi_riemann.c /Fe:pi_riemann.exe
> pi_Riemann.exe
C           pi=3.14159265393  abs.error=3.445e-10  median=0.001916 s  min=0.001911 s
```

C言語化: 動的ライブラリをC/C++で作り、動的ロードする

確認項目	要点
Windows	DLL、関数のexport、呼出規約
Linux	共有ライブラリ.so
共通	整数・浮動小数点型、ポインタ、32/64 bitの整合
配布・保守	ライブラリ探索パス、依存関係、OS別の動作確認が必要

惑星シミュレータ

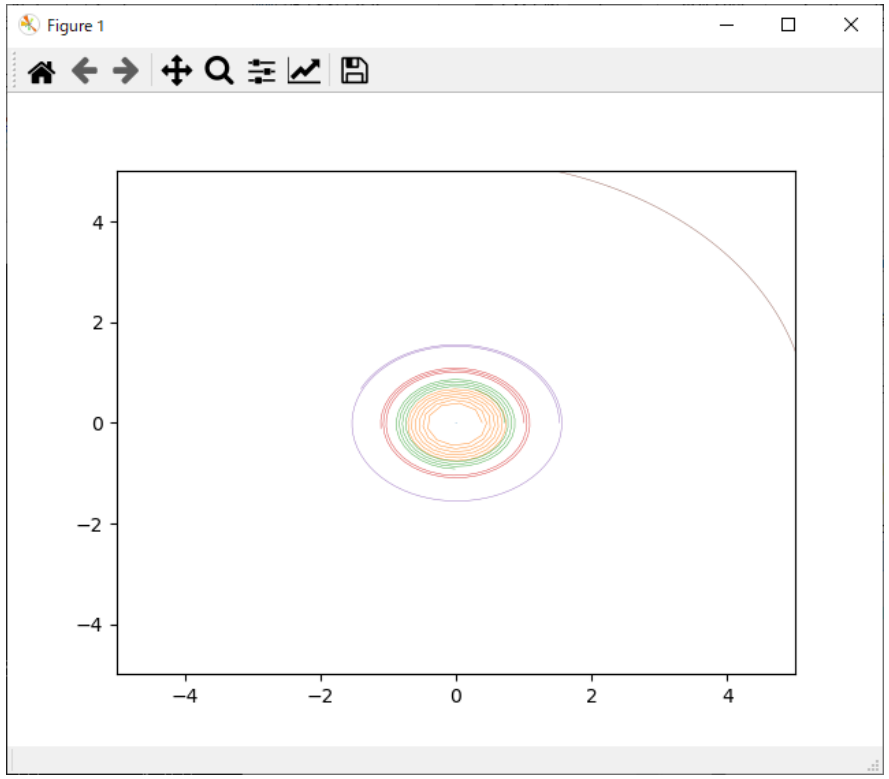
Usage: python diffeq2nd_planet_animation.py solver dt nt

solver: ‘Euler’ or ‘Verlet’

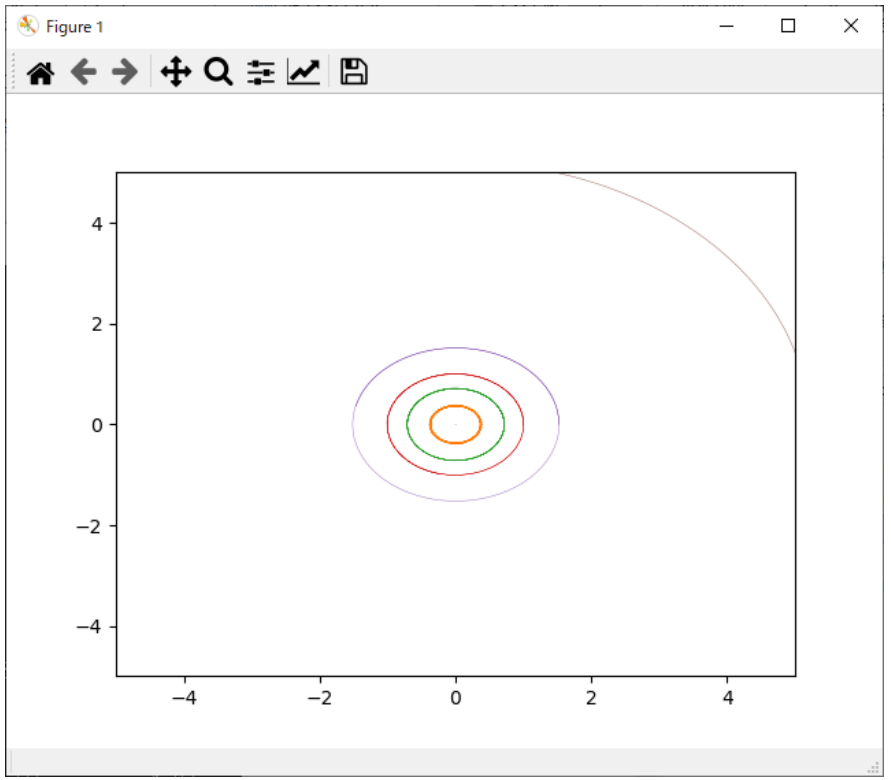
dt: time step in day (time is normalized by a day)

nt: number of steps

python diffeq2nd_planet_animation.py **Euler** 0.2 5000



python diffeq2nd_planet_animation.py **Verlet** 0.2 5000



惑星シミュレータの高速化例

1. リスト型 (python) : diffeq2nd_planet.py
2. numpy.ndarray : diffeq2nd_planet_generator.py
3. numba (parallel=False) : diffeq2nd_planet_generator_numba.py
4. C DLL : diffeq2nd_planet_c.py

> python run_benchmark.py --db planet_db.csv --nt 20000 --repeat 3

backend	median(s)	speedup	match
python	1.506046	1.00	True
numpy	0.261673	5.76	True
numba	0.022866	65.86	True
C	0.021953	68.60	True

注: Numbaは初回実行してコンパイル済みの値

- 数値計算に使う配列は、基本的にndarrayにそろえる
- ループは可能な限りnumpyや組み込み関数に置き換える
- Numba just-in-time (JIT) コンパイルも選択肢

FD積分のベンチマーク: C DLLとの比較

Ne-T¥tkfermi_c_benchmark

Compile: > cl /nologo /O2 /LD /MD tkfermi_core.c

>python benchmark.py

Original cache: DISABLED (__wrapped__); C version: NO CACHE

Windows-11-10.0.26200-SP0 | Python 3.13.14 | NumPy 2.4.6 | SciPy 1.18.0

order=0.5, count=1000 per eta per trial, repeat=3

eta	region	Python us	C us	speedup	C/Py rel.diff
-10	series	0.974	0.951	1.02	0.000e+00
-3	series	1.401	1.012	1.39	0.000e+00
-1	series	2.496	1.317	1.90	0.000e+00
-0.1001	series	14.600	2.647	5.52	0.000e+00
-0.1	quad	41.378	9.642	4.29	0.000e+00
0	quad	40.649	9.902	4.11	0.000e+00
1	quad	57.056	23.309	2.45	0.000e+00
5	quad	61.083	16.476	3.71	0.000e+00
10	quad	68.394	16.432	4.16	0.000e+00
19.9	quad	89.190	17.474	5.10	0.000e+00
20	asymptotic	1.954	1.059	1.85	0.000e+00
50	asymptotic	1.937	1.284	1.51	0.000e+00
100	asymptotic	2.035	1.178	1.73	0.000e+00

Sum of per-eta median times: Python=0.383149s, C=0.102682s; speedup=3.73x

Nuitkaによるコンパイルと実行ファイル化

pythonコードを.exe化するプログラムはいくつかあるが、
コンパイルのしやすさなどから Nuitka を推奨

Nuitka: Pythonプログラムをcコードに変換してビルドする

- 純pythonコードは高速化される場合がある
  NumPyなどはすでにCでコンパイルされており、
  さらに大幅に速くなるとは限らない
- 実行可能ファイルのみでの配布が可能
  配布先のOS・CPU構成に合わせたビルドと動作確認が必要

Nuitkaの使い方

インストール: **python -m pip install -U nuitka**

Cコンパイラ: **WindowsではVisual Studio 2022以降のBuild Toolsで**
「C++によるデスクトップ開発」を導入するなど
https://learn.microsoft.com/en-us/cpp/overview/acquire-msvc?view=msvc-170

通常のコンパイル: python -m nuitka program.py
python -m nuitka --msvc=latest program.py

実行: program.exe

配布: python -m nuitka --mode=standalone program.py
python -m nuitka --msvc=latest --mode=standalone program.py
program.distフォルダ全体を配布

1ファイルにまとめる: python -m nuitka --mode=onefile program.py

Riemann積分の比較: nuiktaの効果

```
> python pi_riemann_benchmark.py
Midpoint Riemann sum: nmax=1,000,000, repeat=3
for          pi=3.141592653589764  abs.error=2.887e-14  median=0.058062 s  min=0.057455 s
generator    pi=3.141592653589877  abs.error=8.349e-14  median=0.079835 s  min=0.079459 s
listcomp     pi=3.141592653589877  abs.error=8.349e-14  median=0.091448 s  min=0.088638 s
numpy        pi=3.141592653589877  abs.error=8.349e-14  median=0.005920 s  min=0.005908 s
```

Nuitkaでコンパイル

```
> Nuitka¥pi_riemann_benchmark.bat
Midpoint Riemann sum: nmax=1,000,000, repeat=3
for          pi=3.141592653589764  abs.error=2.887e-14  median=0.049343 s  min=0.048183 s
generator    pi=3.141592653589877  abs.error=8.349e-14  median=0.124777 s  min=0.118142 s
listcomp     pi=3.141592653589877  abs.error=8.349e-14  median=0.092806 s  min=0.090791 s
numpy        pi=3.141592653589877  abs.error=8.349e-14  median=0.005976 s  min=0.005963 s
```

FD積分のベンチマーク: nuitkaの効果

> python benchmark_fd.py -n 10000 --repeat 3

Normalized F_1/2; count = calls per implementation per trial

implementation	median(s)	us/call	max rel.err
tkfermi	0.410278	41.028	1.790e-11
sonnet5.5	0.667121	66.712	2.136e-16
sol	0.753457	75.346	2.836e-16
Gemini3.8Flash	1.017146	101.715	7.863e-08
copilot	1.046361	104.636	4.389e-07
Gemini3Flash	1.538233	153.823	3.241e-08

> nuitka¥benchmark_fd.bat -n 10000 --repeat 3 **# nuitkaでコンパイル**

Normalized F_1/2; count = calls per implementation per trial

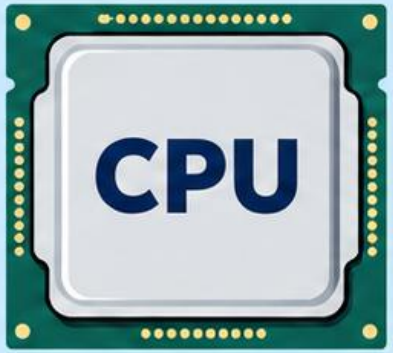
implementation	median(s)	us/call	max rel.err
tkfermi	0.559050	55.905	1.790e-11
sonnet5.5	0.775310	77.531	2.136e-16
sol	0.779119	77.912	2.836e-16
Gemini3.8Flash	1.080404	108.040	7.863e-08
copilot	1.108947	110.895	4.389e-07
Gemini3Flash	1.762363	176.236	3.241e-08

GPUを使う例

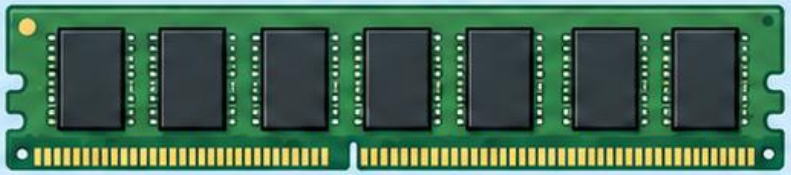
ライブラリ	GPU利用	補足
PyTorch	◎	GPU対応。モデルやデータをGPUに配置して計算
TensorFlow	◎	GPU対応。対応する演算をGPUで実行
GPyTorch	◎	PyTorchを使ってガウス過程回帰をGPUで計算
BoTorch	○	PyTorch／GPyTorchを通してGPUを利用
scikit-learn	基本はCPU	一部に例外はあるが、通常はCPU用と考えてよい
GPy	基本はCPU	通常の利用ではCPU
PHYSBO	基本はCPU	主にCPUでの計算・MPI並列化を利用

やりたいこと	よく使う選択肢
自作のNumPy数値計算をGPU化	CuPy
ニューラルネットワークの構築・学習	PyTorch／TensorFlow
ガウス過程回帰・ベイズ最適化	GPyTorch／BoTorch

CPU



RAM

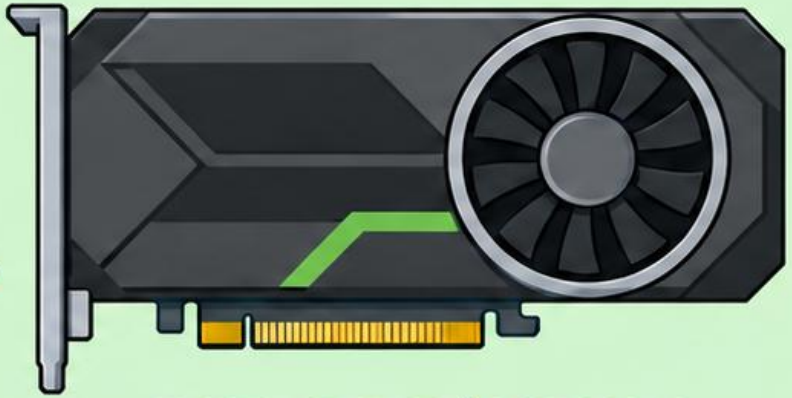


通常のプログラム・測定データ

PCIe 転送

転送を繰り返さない

GPU



VRAM/HBM



大量の配列を保持して超並列処理

CPUで準備 → GPUへ転送 → VRAM上で連続計算 → 結果だけCPUへ

GPUに一度送る → GPU内でまとめて処理 → 最後に結果を戻す

GPUが有効な場合

重いベクトル・行列計算、FFTなど

- CPUからGPUへのデータの転送、計算結果の逆転送の
  オーバーヘッドを考慮
- GPU内にデータを保持したまま、できるだけ一連の計算を行う

ベンチマーク

- GPUの処理完了を待ってから時間を測る。初回の初期化時間は分けて評価する
- データ転送時間を含めた全計算時間、GPU計算のみの時間、
  CPU計算の全計算時間を比較する
- 同じ計算でも、データサイズによって優劣が変わる。
  一般に、データサイズが大きいとGPUが有利になる
- 一般に、並列に処理できる計算量が多いほどGPUが有利になる

FFT/DFTの比較

> Python dft_comparison.py

注: CPU FFT: GPUを使わない; GPU kernel: GPU内の計算時間; GPU total: データ転送などを含む

Numba first call (JIT/cache startup): 0.462589 s

N	Python DFT	Numba DFT	CPU FFT	GPU kernel	GPU total
64	0.0012506	6.68e-05	4.40003e-06	2.37001e-05	0.0001064
128	0.0048591	0.0002606	5.09992e-06	3.18e-05	0.0001114
256	0.0196942	0.0009308	7.49994e-06	3.11001e-05	0.0001103
512	0.0773484	0.0035724	1.19e-05	3.21001e-05	0.0001241
1024	0.304044	0.0141	1.48001e-05	3.02e-05	0.000118
2048	skip	0.0556043	2.64999e-05	3.71002e-05	0.0001483
--cut—					
131072	skip	skip	0.0026426	8.98e-05	0.0006279
262144	skip	skip	0.0054587	0.000202	0.00126
524288	skip	skip	0.0121744	0.0002495	0.0023286
1048576	skip	skip	0.0270214	0.0004679	0.0046714

N ≧ 131072でGPUが勝つ

バンド計算: 平面波法

一次結合の基底関数として平面波を使う

$$\phi_{\mathbf{k}}(\mathbf{r}) = \exp(i\mathbf{k}\cdot\mathbf{r})\sum C_{hkl}u_{hkl}(\mathbf{r}) \qquad u_{hkl}(\mathbf{r}) = \exp[i\mathbf{G}_{\mathbf{hkl}}\cdot\mathbf{r}]$$

$$\begin{vmatrix} H_{11}-ES_{11} & H_{12}-ES_{12} & \cdots & H_{1n}-ES_{1n} \\ H_{21}-ES_{21} & H_{22}-ES_{ss} & & H_{2n}-ES_{2n} \\ \vdots & & \ddots & \vdots \\ H_{n1}-ES_{n1} & H_{n2}-ES_{n2} & \cdots & H_{nn}-ES_{nn} \end{vmatrix} = 0$$

対角化

$$\langle u_{h'k'l'}|H|u_{hkl}\rangle = \delta_{hkl,h'k'l'}\frac{\hbar^2}{2m}(\mathbf{k}+\mathbf{G}_{\mathbf{hkl}})^2 + V^*(\mathbf{G}_{\mathbf{hkl}}-\mathbf{G}_{\mathbf{h'k'l'}})$$

ポテンシャルのフーリエ変換

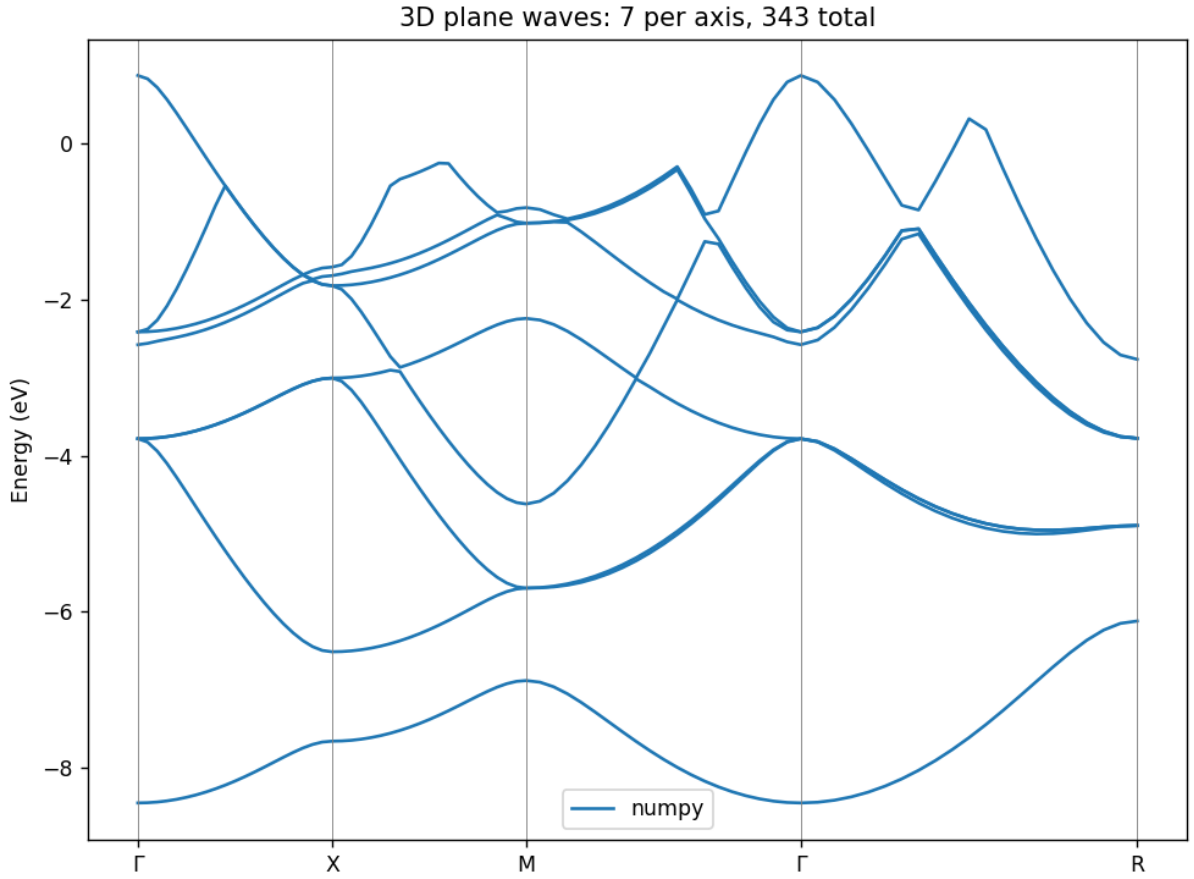
実際の計算時間のほとんどは行列の対角化: GPUで高速化が容易

プログラム: 3次元平面波法

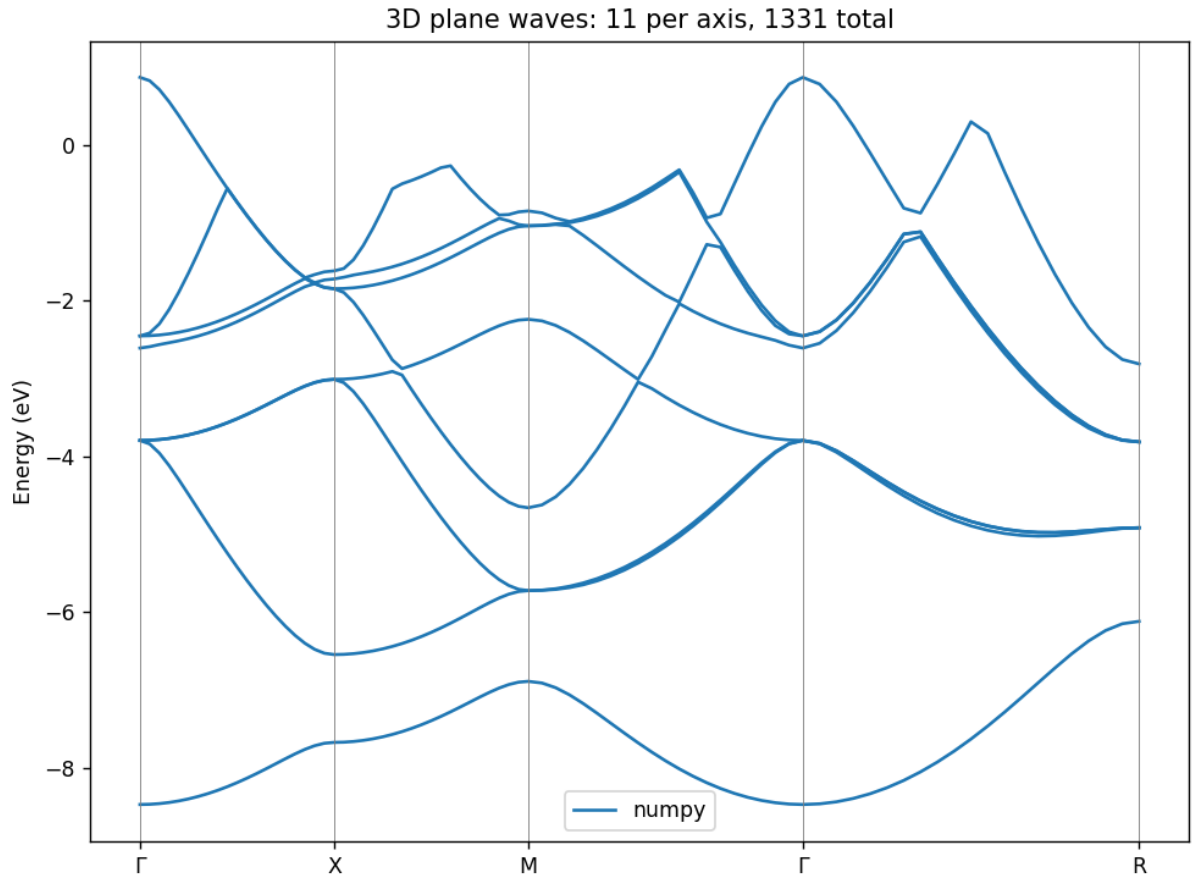
pw3d.py

Si の格子定数　a = 5.4064 Å　　$m^* = 1.0m_e$
ポテンシャル $V(x)$:　障壁幅　0.49064 Å　　障壁高さ 10.0 eV

python pw3d.py --width 5.0 --nk 21 --ng 7
基底平面波数: 7^3



python pw3d.py --width 5.0 --nk 21 --ng 11
基底平面波数: 11^3



pw3d.py: GPUによる行列対角化

NVIDIA RTX PRO 6000 Blackwell Max-Q Workstation Edition

基底数: $17^3 = 4913$

numpy: startup/warmup=0.000057s (separate from trials)

numpy: median total=37.898135s

gpu: startup/warmup=0.264357s (separate from trials)

gpu: median total=13.110283s

CPU/GPU speedup (median totals)=2.891x

基底数: $7^3 = 343$

numpy: startup/warmup=0.000053s (separate from trials)

numpy: median total=0.177483s

gpu: startup/warmup=0.275932s (separate from trials)

gpu: median total=0.334608s

CPU/GPU speedup (median totals)=0.530x

NVIDIA GeForce RTX 5070

基底数: $13^3 = 2197$

numpy: median total=11.829635s

gpu: median total=3.684128s

CPU/GPU speedup (median totals)=3.211x

まとめ

高速化の手順

- 正しさを確認
- 測定。プロファイリングで対象を特定する
- ボトルネックを改善
- 同じ条件で再測定する

代表的な高速化手段

1. アルゴリズム、必要精度の確認・選択
2. 最適化・コンパイルされたライブラリを使う。
3. @numba.vectorize()を使ってufunc化・コンパイル
4. 事前計算結果を再利用 (lru_cache)
5. Numbaの @jitを使って実行中コンパイルを行う

6. Nuitkaによるコンパイルはソースコード依存が大きい。
   上記の方法で高速化している場合は効果が出ないことがある