numpy と線形代数

本資料中では斜体太字でベクトルを表す。また、大文字の太字で行列を表す。

1 numpy

Python で数値計算をする際、numpy というライブラリを用いる。ライブラリを用いる時は、自分で書いたプログラムを使う時と同じように「import」することが出来る。インタプリタを使って試せるいくつかの例を用意した。適宜分からない変数を出力しながら理解を深めて欲しい。

まず、numpy を import することで numpy を利用できるようになる。

```
import numpy as np
```

2 スカラー

numpy には様々な数学関数が用意されている。

```
np.sqrt(3)
np.sqrt(-3)
np.sqrt(-3 + 0j)
np.exp(3)
np.log(3)
np.log2(4)
np.log10(100)
```

log の底はデフォルトでネイピア数 e だが、底が 2 や 10 の関数も用意されている。

また、いくつかの定数が用意されている。

```
np.e
np.pi
np.log(np.e)
np.exp(1j * np.pi)
```

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

最後の例はオイラーの等式であるが、見ての通り、0 のはずの虚部が厳密に 0 になっていない。これは浮動小数点の限界に由来する数値誤差である。小数の計算結果が厳密に 0 になることは期待できない。ゆえに、「==」で小数を判定するのは誤りである。「imag」で虚部を取得できるが、これを 0 との比較の結果は「False」となってしまう。

```
np.exp(1j * np.pi).imag == 0
```

そのかわり、numpy には allclose という近い大きさかどうかを判定する関数が用意されている。

```
np.allclose(np.exp(1j * np.pi).imag, 0)
```

こちらは結果が「True」になる。

3 ベクトル

numpy は「ndarray」という数値計算向けの配列データ構造を用意してくれている。このデータ型が存在するが故に我々は Python を利用するくらいに大事な型である。

「ndarray」は「np.array」関数に「list」を代入することで得られる。また、「ndarray」は「list」よりも数値計算に向いた機能を持っている。

```
x_list = [1.0, 2.0, 3.0]
x = np.array(x_list)
```

ここで、「x_list」は python 標準のリスト型であり、「x」は「np.ndarray」型である。確認して欲しい。

```
print(type(x_list))
print(type(x))
```

上のコードでは「x_list」に一度リストを代入してから変換したが、普通はまとめて行ってしまう。

```
y = np.array([2.0, 5.0, 9.0])
```

また、リストと同じように「np.ndarray」もアクセスや代入を行うことが出来る。

```
x[0]
```

```
x[0:2]
```

```
x[0:3:2]
x[0:2] = np.array([10.0, 20.0])
```

「np.ndarray」では、数学のベクトルのような要素ごとの計算がサポートされる。

足し算や引き算等の演算は全て要素ごとの計算として扱われる。意外に思うかも知れないが、掛け算や割り算等も全て同様に要素ごとの演算となる。他にも「np.sqrt」や「np.log」等の関数も要素ごとの計算となる。

```
x
x + y
x * 2
x * y
x ** y
x / 3
np.sqrt(x)
np.sin(x)
```

ここで、

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} * \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 * y_1 \\ x_2 * y_2 \end{pmatrix}$$

となることに注意。

また、ベクトルに対する演算として、ノルムを求める「np.linalg.norm」が存在する。

$$\mathrm{norm}\,\mathbf{x} = \sqrt{\sum_i x_i^2}$$

```
np.linalg.norm(x)
```

名前が長くて嫌な人は直接インポートしよう。

```
from numpy.linalg import norm
norm(x)
```

また、線形代数で慣れ親しんだ内積は専用の関数が用意されている。同じ内積でも書き方が幾通りかあるが、好きなものを使ってくれて構わない。

$$\mathrm{dot}(\mathbf{x}, \mathbf{y}) = \mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i$$

```
np.dot(x, y)
x.dot(y)
x @ y
```

ただし、@を使った記法は古い Python ではサポートされていないので注意。

同じく、外積も定義されている。

$$\mathrm{cross}(\mathbf{x},\mathbf{y}) = \mathbf{x} \times \mathbf{y} = [x_2y_3 - x_3y_2, x_3y_1 - x_1y_3, x_1y_2 - x_2y_1]^{\mathrm{T}}$$

```
np.cross(x, y)
```

3.1 課題

x と y の外積は x とも y とも内積が 0 になるので確認せよ。

さて、演習として、同じ長さのベクトルに対して「np.linalg.dot」と同等な機能を持つ「dot_vector」関数を実装してみよう。

```
def dot_vector(x, y):
    n = len(x)
    d = 0
    for i in range(n):
        d += x[i] * y[i]
    return d
```

色々なインプットに対して numpy.dot と同じになることを試してみて欲しい。もし結果が違うということであれば、実装を間違えたということである。

4 行列

Numpy では、素晴らしいことに行列を扱うことも出来る。行列を作る時は主に、Python の「list」の「list」を使って構成する。

```
a = np.array([[1, 2], [3, 4]])
```

行列の形状や次元数を調べることが出来る。行列なら「ndim」は「2」である。(ベクトルなら「1」、スカラーなら「0」である。)

```
a.shape
a.ndim
a.size
a.dtype
a.T
```

また、単位行列や零行列のような特殊な行列は専用の生成関数が容易されている。

```
z = np.zeros(4)
o = np.ones(3)
e = np.eye(5)
```

また、その気になればリスト内包表記を使って二重の配列を作り、行列を作ることも出来る。

```
b = np.array([[np.exp(i * j) for j in range(3)] for i in range(3)])
```

もちろん、行列と行列の内積も定義されている。

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_k A_{ik} B_{kj}$$

```
np.dot(e, b)
e @ b
b @ b
b @ z
b @ x
```

行列に対して np.dot と同等の機能を持つ 「dot_matrix」関数を作成し、Jupyter 上で定義 s してみよう。ただ、for 文と shape、zeros、3 重ループ、足し算を用いて実装せよ。

以下のコードを実行して True が表示されれば合格である。

```
def dot_matrix(a, b):
    c = np.zeros((a.shape[0], b.shape[1]))
```

```
    for i in range(a.shape[0]):
        for j in range(b.shape[1]):
            for k in range(a.shape[1]):
                c[i, j] += a[i, k] * b[k, j]
    return c
```

```
a = np.random.random([3, 4])
b = np.random.random([4, 5])
np.allclose(a @ b, dot_matrix(a, b))
```

5 エルミート行列の対角化

線形代数で習った「エルミート行列の対角化」は量子力学において必須である。エルミート行列 H に関して、

$$H_{ij} = H_{ji}^*$$

が成立する。ここで、*は複素共役である。すなわち、H を転置して複素共役を取ると、H に戻る。エルミート行列は常に対角化可能である。

今、適当なエルミート行列を考えよう。

```
H = np.array([[0, 10, 2],
              [10, 4, 3],
              [ 2, 3, 1]])
```

エルミート行列であることを確かめよう。

H == np.conj(H.T)

これを対角化する関数が用意されている。対角化とは

$$\mathrm{H} = \mathrm{C}\epsilon\mathrm{C}^{\dagger}$$

という形に式変形することである。ここで、ϵは対角行列であり、対角要素にしか要素を持たない。（対角要素とは、ϵ_{11}やϵ_{22}等の同じ番号同士のインデックスである。）この操作は numpy に用意されており、以下のコードで完了する。

```
e, C = np.linalg.eigh(H)
```

ただこれだけで対角化は完了し、e にϵの対角要素が、C に C が格納される。

ここで、e[i]のことを i 番目の固有値と呼び、C[:,i]を i 番目の固有ベクトルと呼ぶ。

まず、固有値と固有ベクトルが格納されていることを確認しよう。i 番目の固有ベクトルをc_i、i 番目の固有値をϵ_iとすると、

$$\boldsymbol{H} \cdot \boldsymbol{c}_i = \epsilon_i \boldsymbol{c}_i$$

が成り立つはずである。上式において、i=0 の時、左辺は

```
H @ C[:, 0]
```

で表され、右辺は

```
e[0] * C[:, 0]
```

で表される。右辺と左辺が同じであることを確かめて欲しい。また、0 以外のインデックスに関しても同様に確かめて欲しい。因みに下のコードを使えば一気に全て確かめられる。

```
for i in range(C.shape[1]):
    print(np.allclose(H @ C[:, i], e[i] * C[:,i]))
```

次に、C と e で元の H を再現できることを確認しよう。

固有値を対角に持つ対角行列をϵとすれば、

$$H = C \cdot \epsilon \cdot C^T$$

が成り立つはずである。

```
np.allclose(H, C @ np.diag(e) @ C.T)
```

6 行列式

線形代数で習った行列式も計算可能である。

```
np.linalg.det(H)
```

7 逆行列

逆行列も計算可能である。

```
Hi = np.linalg.inv(H)
```

Hi が H の逆行列であることを示すには、dot 積を取ればよい。

```
Hi @ H
```

```
H @ Hi
```

また、Hi の行列式と H お行列式の積は 1 になる。

```
np.linalg.det(H) * np.linalg.det(Hi)
```

ただし、計算精度の問題で厳密に 1 になるわけではないことに注意。

8 一次方程式の解

行列 H、ベクトル x,a に対して、

$$\mathbf{H}\boldsymbol{x} = \boldsymbol{a}$$

の解は

$$\boldsymbol{x} = \mathbf{H}^{-1}\boldsymbol{a}$$

であるが、実は逆行列を求める操作は遅いので、代わりにこの問題を解く専門の関数が存在する。

```
a = np.array([0.3, 0.1, 0.4])
```

とでもしておき、

```
np.linalg.solve(H, a)
```

である。この結果と H の積が a に戻ることを確認して欲しい。