第 5 章

基盤モジュール: NumPy

大量の実数演算や複素演算を行う科学技術計算においては，高速な計算を行うための工夫を行ったライブラリの利用が必須です。既に見てきたように，Pytnon はネイティブな機能として有限桁の浮動小数点演算をベースにこれらの処理が可能ですが，現代のコンピュータの特性を生かした高速化については別途，C/C++ 等のコンパイラ言語を用いての実装が不可欠で，その有力なモジュールが NumPy，そして NumPy をベースとした SciPy です。本章ではまず NumPy について，これらの機能や高速性を，Python スクリプトを用いた事例を通じて学んでいくことにします。

5.1 NumPy の初等関数

NumPy は math モジュール，cmath モジュールで使用できる基本的な初等関数が使用できます。NumPy には”np”というエイリアスを使用することが多いので，以下，そのように読み込むようにしています。

ソースコード 5.1 np_calc.py

```
1  # np_calc.py: NumPy の初等関数機能
2  import numpy as np # NumPy
3  import math # math モジュール
4  import cmath # cmath モジュール
5
6  a = -3.0
7  z = -1.0 + 2.0j
8
9  print('a␣=␣', a)
10 print('z␣=␣', z)
11
12 # 平方根: NumPy
13 print('np.sqrt(a)␣=␣', np.sqrt(a))
14 print('np.sqrt(z)␣=␣', np.sqrt(z))
15
16 # 平方根: math, cmath
17 # print('math.sqrt(a) = ', math.sqrt(a)) # エラーになる
18 print('cmath.sqrt(z)␣=␣', cmath.sqrt(z))
19
20 # べき乗: NumPy
21 c = np.sqrt(a)
22 w = np.sqrt(z)
23 print('(np.sqrt(a))^2␣=␣', c ** 2)
```

```
print('(np.sqrt(z))^2␣=␣', w ** 2)

# べき乗: math
# c = math.sqrt(a) #エラーになる
w = cmath.sqrt(z)
# print('(math.sqrt(a))^2 = ', c ** 2)
print('(cmath.sqrt(z))^2␣=␣', w ** 2)

# 指数関数,三角関数,対数関数: NumPy
print('np.exp(a)␣␣␣=␣', np.exp(a))
print('np.sin(a)␣␣␣=␣', np.sin(a))
print('np.log(a)␣␣␣=␣', np.log(a))
print('np.log10(a)␣=␣', np.log10(a))
print('np.exp(z)␣␣␣=␣', np.exp(z))
print('np.sin(z)␣␣␣=␣', np.sin(z))
print('np.log(z)␣␣␣=␣', np.log(z))
print('np.log10(z)␣=␣', np.log10(z))

# 指数関数,三角関数,対数関数: math
print('math.exp(a)␣␣␣=␣', math.exp(a))
print('math.sin(a)␣␣␣=␣', math.sin(a))
# print('math.log(a) = ', math.log(a)) # エラーになる
# print('math.log10(a) = ', math.log10(a)) # エラーになる
print('cmath.exp(z)␣␣␣=␣', cmath.exp(z))
print('cmath.sin(z)␣␣␣=␣', cmath.sin(z))
print('cmath.log(z)␣␣␣=␣', cmath.log(z))
print('cmath.log10(z)␣=␣', cmath.log10(z))
```

NumPy の数学関数は引数にリストを与えることで，各要素ごとの関数値を計算することができます。例えば閉区間 $[a, b]$ を n 分割し，端点を含む分割点 $x_i = (b-a)/n$ $(i = 0, 1, ..., n)$ における $\tan x_i$ と $tan^{-1}x_i$ の値を計算するスクリプトは下記のようになります。

ソースコード 5.2　np_calc_vec.py

```
# np_calc_vec.py: リストを引数とする関数計算
import numpy as np # NumPy

# [-π/4, π/4]をn 分割
n = 5 # 分割数
a, b = -np.pi / 4.0, np.pi / 4.0 # 端点
h = (b - a) / n # 区間幅

# x = [a, a + h, ..., a * (n - 1)h = b - h, a * nh = b]
xlist = np.linspace(a, b, n) # [a + h * i for i in range(n + 1)]
print('x␣=␣', xlist)

# tan_list = tan(x), atan_list = atan(x)
tan_list, atan_list = np.tan(xlist), np.arctan(xlist)
print('␣tan(x)␣=␣', tan_list)
print('atan(x)␣=␣', atan_list)
```

NumPy の linspace 関数を用いて区間を n 等分して端点を含む値をリスト化し，それを $\tan x$ と $\tan^{-1} x$ の引数として使用しています。

np_calc_vec.py の実行結果

```
x =  [-0.78539816 -0.39269908  0.          0.39269908  0.78539816]
 tan(x) =  [-1.         -0.41421356  0.          0.41421356  1.        ]
atan(x) =  [-0.66577375 -0.37419668  0.          0.37419668  0.66577375]
```

問題 5.1

1. 適当な x に対して $\log(\exp(x)) = x$ かつ $\log_{10}(10^x) = x$ であることを確認する check_log_exp.py スクリプトを作れ。
2. 適当な x に対して $\cos(\arccos(x)) = x$ かつ $\tan(\arctan(x)) = x$ であることを確認する check_cos_acos.py スクリプトを作れ。

5.2 絶対誤差と相対誤差

正しい値を真値 (true value) と呼び，真値に近いがズレがあるかもしれない値を近似値 (approximation) と呼びます。今，真値 $x \in \mathbb{R}$ に対応する近似値 $\widetilde{x} \in \mathbb{R}$ とする時，誤差 (error) は $E(\widetilde{x})$ と表現し

$$E(\widetilde{x}) = \widetilde{x} - x \tag{5.1}$$

と定義します。真値とのずれを表現したものですが，ズレの大きさだけ見たい時は誤差の絶対値を使います。この時

$$aE(\widetilde{x}) = |E(\widetilde{x})| \tag{5.2}$$

を絶対誤差 (absolute error) と呼びます。これを使用して，近似値に含まれる誤差の大きさを真値との割合で示したものが相対誤差 (relative error) で，

$$rE(\widetilde{x}) = \begin{cases} \frac{aE(\widetilde{x})}{|x|} & (x \neq 0) \\ aE(\widetilde{x}) & (x = 0) \end{cases} \tag{5.3}$$

と定義されます。

更に，真値と一致する m 進桁数を求める際にはこの相対誤差 $rE(\widetilde{x})$ を用いて

$$\lfloor -\log_m(rE(\widetilde{x})) \rfloor \tag{5.4}$$

を求めます。これを m 進有効桁数 (significant digits) と呼びます。$\lfloor \cdot \rfloor$ は小数点以下を切り捨てて整数部のみ取り出す床関数 (floor function) です。

以下は $x = \sqrt{2}$, $\widetilde{x} = 1.4142$ とした時の，$E(\widetilde{x})$, $aE(\widetilde{x})$, $rE(\widetilde{x})$ そして 10 進有効桁数を求める Python スクリプトです。

ソースコード 5.3 errors.py

```
1 # errors.py: 誤差の計算
2 import numpy as np # NumPy
3
4 # 真値
5 x = np.sqrt(2.0)
6 print('True␣x␣=␣', x)
7
```

```
# 近似値
approx_x = 1.4142
print('Approx x = ', approx_x)

# 誤差
Err = approx_x - x
print(f'E(Approx x) = {Err:10.2e}')

# 絶対誤差
aErr = np.abs(Err)
print(f'aE(Approx x) = {aErr:10.2e}')

# 相対誤差
rErr = aErr
if x != 0.0:
    rErr = aErr / np.abs(x)
print(f'rE(Approx x) = {rErr:10.2e}')

# 10進有効桁数
print(f'Sig. digits = {np.floor(-np.log10(rErr)):10.0g}')
```

誤差は大きさを確認するためのものですので，通常 2〜3 桁程度の表示で十分です。これを実行すると下記のようになります。

errors.py の実行結果

```
True x =  1.4142135623730951
Approx x =  1.4142
E(Approx x) =  -1.36e-05
aE(Approx x) =   1.36e-05
rE(Approx x) =   9.59e-06
Sig. digits =          5
```

問題 5.2

適当な区間 $[a, b]$ を n 等分し，全ての点において $\widetilde{x} = \cos(\arccos(x))$ を求め，絶対誤差が非ゼロの点における相対誤差と 10 進有効桁数を求める Python スクリプト `relerr_list.py` を作れ。

5.3 自動微分と数値微分

微分 (differentiation) は，関数 $f(x) \in \mathbb{R}$ の導関数 (derivative) を求める操作の総称です。Python で扱える微分のための手法は，いわゆる手計算と同じ記号操作で求める自動微分 (automatic differentiation) と，導関数の定義に則って極限値の近似値を使う数値微分 (numerical differentiation) の 2 種類があります。

■自動微分　例えば $f(x) = \exp(\cos x) - x^3$ の導関数 $f'(x) = -\exp(\cos x) * \sin x - 3x^2$ を求めるには，augograd パッケージが使えます。

ソースコード 5.4　`autodiff.py`

```
# autograd_diff.py: 自動微分
import autograd.numpy as np
import autograd

# 元の関数
def func(x):
    return np.exp(np.cos(x)) - x ** 3

# 真の導関数
def true_dfunc(x):
    return -np.exp(np.cos(x)) * np.sin(x) - 3 * x ** 2

# x = [-5, 5]
x = np.linspace(-5, 5, 100)

# 自動微分による導関数
dfunc = autograd.elementwise_grad(func)

# 相対誤差チェック
reldiff = np.abs((dfunc(x) - true_dfunc(x)) / true_dfunc(x))
print('reldiff␣=␣', reldiff)
```

■数値微分　Python の数式で記述できる関数の導関数は自動微分で計算できますが，例えば条件分岐を伴うような，一つの数式としては表現できない関数に対しては，そのまま適用できません。そういう時には微分の定義に立ち戻り

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{5.5}$$

という極限値の右辺の値の近似値，すなわち，適度に小さい h を与えて $f'(x) \approx (f(x+h) - f(x)))/h$ と見立てます。これを数値微分と呼びます。導関数の定義から，(5.5) 式の右辺の極限値は $h \to +0$ (数直線の右から 0 に近づく) であっても，$h \to -0$ (数直線の左から 0 に近づく) のどちらでも同じ値に収束する場合のみ考えますので，数値微分としては，小さい正の h に対して

$$f'(x) \approx \begin{cases} \frac{f(x+h)-f(x)}{h} & \text{(前進差分商)} \\ \frac{f(x+h)-f(x-h)}{2h} & \text{(中心差分商)} \\ \frac{f(x)-f(x-h)}{h} & \text{(後退差分商)} \end{cases} \tag{5.6}$$

の 3 種類の近似が考えられます。これを Python スクリプトで計算したものが num_diff.py です。

ソースコード 5.5　num_diff.py

```
# num_diff.py: 数値微分
import numpy as np # NumPy

# 元の関数
def func(x):
  return np.exp(np.cos(x)) - x ** 3

# 真の導関数
def true_dfunc(x):
  return -np.exp(np.cos(x)) * np.sin(x) - 3 * x ** 2

# 前進差分商: (f(x + h) - f(x)) / h
```

```
13 def forward_diff(x, h, f):
14   return (f(x + h) - f(x)) / h
15
16 # 中心差分商: (f(x + h) - f(x - h)) / 2h
17 def central_diff(x, h, f):
18   return (f(x + h) - f(x - h)) / (2.0 * h)
19
20 # 後退差分商: (f(x) - f(x - h)) / h
21 def backward_diff(x, h, f):
22   return (f(x) - f(x - h)) / h
23
24 # x = [-5, 5]
25 x = np.linspace(-5, 5, 4)
26 h = 10.0**(-5)
27
28 # 前進差分商,中心差分商,後退差分商
29 fdiff = forward_diff(x, h, func)
30 cdiff = central_diff(x, h, func)
31 bdiff = backward_diff(x, h, func)
32
33 # 相対誤差チェック
34 reldiff = np.abs((fdiff - true_dfunc(x)) / true_dfunc(x))
35 print('Forward␣diff␣relerr␣=␣', reldiff)
36 reldiff = np.abs((cdiff - true_dfunc(x)) / true_dfunc(x))
37 print('Central␣diff␣relerr␣=␣', reldiff)
38 reldiff = np.abs((bdiff - true_dfunc(x)) / true_dfunc(x))
39 print('Backword␣diff␣relerr=␣', reldiff)
```

これを実行した結果を下記に示します。明らかに中心差分商の精度が良いことが分かります。

num_diff.py の実行結果

```
Forward diff relerr =  [2.02200099e-06 7.39511504e-06 4.87809771e-06 1.97723151e-06]
Central diff relerr =  [4.04379069e-11 1.82339348e-11 1.63852760e-11 4.48259351e-11]
Backword diff relerr=  [2.02192012e-06 7.39515151e-06 4.87806494e-06 1.97732116e-06]
```

問題 5.3

$p(x) = \sin(\exp(x)) = \sin e^x$ の数値微分に基づく導関数を次の方法で求めるスクリプト diff_poly.py を作り，正しい導関数 $p'(x)$ が導出できていることを，下記の方法で確認せよ。

1. 適当な区間 $[a, b]$ を設定し，これを n 等分した時の分点を $x_i \in [a, b]$ とする。
2. すべての分点 x_i において小さい値 h を用いて計算した数値微分の値 $p'(x_i)$ の相対誤差を求め，0 に近いことを確認する。

5.4 1 変数多項式と代数方程式

n 次多項式 $p_n(x) = \sum_{i=0}^{n} a_i x^i$ を左辺として持つ非線形方程式 $p_n(x) = 0$ を代数方程式 (algebraic equation) と呼びます。既に見てきた 1 次方程式，2 次方程式は $n = 1, 2$ の代数方程式です。まずは NumPy

の polynomial モジュールの使い方から見ていくことにします。

1 変数多項式については，NumPy では poly1d モジュールが伝統的に使われてきましたが，NumPy 1.4 からは polynomial パッケージを使うことが推奨されています。以下は多項式の定義（係数のみ），値の計算，導関数（多項式の微分），原始関数（多項式の積分），解の計算と検算を行っています。

例えば 3 次多項式 $p_3(x) = 2 + 2x^2 + 4x^3$ に対して，

$$
\begin{aligned}
p_3(3) &= 2 + 2 \times 3^2 + 4 \times 3^3 = 128 \\
p_3'(x) &= 4x + 12x^2 \\
p_3'(3) &= 4 \times 3 + 12 \times 3^2 = 120 \\
P_3(x) &= \int p_3(x)dx = 2x + \frac{2}{3}x^3 + x^4 \\
P_3(3) &= 2 \times 3 + \frac{2}{3} \times 3^3 + 3^4 = 105
\end{aligned}
$$

の計算を行うスクリプトは下記のようになります。

ソースコード 5.6　eval_poly.py

```
1  # eval_poly.py : 多項式の計算
2  import numpy as np # NumPy
3  import numpy.polynomial.polynomial as npppoly # Polynomial モジュール
4
5  # 係数を指定した公式の定義
6  # p(x) = 2 + 0 * x + 2 * x^2 + 4 * x^3
7  p_coef = np.array([2, 0, 2, 4]) # 単なるリストも可
8  print('p(x) = ', p_coef)
9
10 # p(3)の値の計算
11 print('p(3) = ', npppoly.polyval(3, p_coef))
12
13 # p(x)の導関数p'(x)とp'(3)の計算
14 dp_coef = npppoly.polyder(p_coef) # 係数の計算
15 print('p\'(x) = ', dp_coef)
16 print('p\'(3) = ', npppoly.polyval(3, dp_coef))
17
18 # p(x)の原始関数P(x)とP(3)の計算
19 ip_coef = npppoly.polyint(p_coef) # 係数の計算
20 print('P(x) = ', ip_coef)
21 print('P(3) = ', npppoly.polyval(3, ip_coef))
22
23 # p(x) = 0の解
24 sols = npppoly.polyroots(p_coef)
25 print('roots of p(x) = ', sols)
26
27 # 検算 p(x) == 0?
28 p_vals = npppoly.polyval(sols, p_coef)
29 print('p(roots) = ', p_vals)
```

このスクリプトを実行すると，下記のように表示されます。

eval_poly.py の実行結果

```
p(x) =  [2 0 2 4]
p(3) =  128.0
p'(x) =  [ 0.  4. 12.]
p'(3) =  120.0
P(x) =  [0.         2.         0.         0.66666667 1.        ]
P(3) =  105.0
roots of p(x) =  [-1.  +0.j          0.25-0.66143783j  0.25+0.66143783j]
p(roots) =  [-3.55271368e-15+0.00000000e+00j -4.44089210e-16+1.22124533e-15j
 -4.44089210e-16-1.22124533e-15j]
```

詳細の説明は省きますが，係数 a_i が既知の代数方程式の解 (根, root) は行列固有値として求めることができます。これを利用して，下記のように根として，1, 2, ..., 20 を持つ代数方程式を生成し，真の解との相対誤差を求める Python スクリプトを下記に示します。

ソースコード 5.7　roots_poly.py

```
1  # roots_poly.py : 代数方程式を解く
2  import numpy as np # NumPy
3  import numpy.polynomial.polynomial as npppoly # Polynomial モジュール
4
5  # 次数
6  max_deg = 20
7
8  # 真の解 : true_roots = [n, n-1, ..., 1]
9  true_roots = np.array(range(max_deg, 0, -1))
10 print('true_roots␣=␣', true_roots)
11
12 # 多項式p(x) = (x - n) * ... * (x - 1) の係数を生成
13 poly_coef = npppoly.polyfromroots(true_roots)
14 print('polynomial␣=␣', poly_coef)
15
16 # 代数方程式の解 (根)を導出
17 approx_roots = npppoly.polyroots(poly_coef)
18 approx_roots.sort() # 並び替え
19 true_roots.sort() # 並び替え
20 print('approx_roots␣=␣', approx_roots)
21
22 # 相対誤差の計算と表示
23 relerr_approx_roots = np.abs((approx_roots - true_roots) / true_roots)
24 print('relerr␣␣␣␣␣␣␣=␣', relerr_approx_roots)
```

roots_poly.py の実行結果

```
true_roots =  [20 19 18 17 16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1]
polynomial =  [ 2.43290201e+18 -8.75294804e+18  1.38037598e+19 -1.28709312e+19
  8.03781182e+18 -3.59997952e+18  1.20664780e+18 -3.11333643e+17
  6.30308121e+16 -1.01422999e+16  1.30753501e+15 -1.35585183e+14
  1.13102770e+13 -7.56111184e+11  4.01717716e+10 -1.67228082e+09
  5.33279460e+07 -1.25685000e+06  2.06150000e+04 -2.10000000e+02
  1.00000000e+00]
approx_roots =  [ 1.          2.          3.          4.00000002  4.99999947  6.0000
0709
  6.99993943  8.00035997  8.99843499 10.00522433 10.98679214 12.0273193
 12.95810032 14.05250478 14.95015682 16.03436037 16.98177036 18.00624863
 18.99865001 20.00013198]
relerr       =  [1.46549439e-14 2.34479103e-13 1.15821427e-10 5.38099831e-09
 1.05630513e-07 1.18136622e-06 8.65267594e-06 4.49959094e-05
 1.73889574e-04 5.22433082e-04 1.20071417e-03 2.27660827e-03
 3.22305251e-03 3.75034119e-03 3.32287863e-03 2.14752316e-03
 1.07233198e-03 3.47145955e-04 7.10522741e-05 6.59883799e-06]
```

あまり精度が良くないことが分かります。次数が大きくなるにつれて精度が落ちる問題は，次第に悪条件 (ill-conditioned) 化していると言えます。

問題 5.4

以下の多項式 $p_n(x)$ を左辺とする代数方程式 $p_n(x) = 0$ の解を求めて下さい。また，導出した解の検算 ($p_n(x) = 0$ の確認) も行って下さい。

1. 自分の学籍番号を係数とする $p_n(x)$。例えば学籍番号が 1823054 の時は $p_6(x) = x^6 + 8x^5 + 2x^4 + 3x^3 + 5x + 4$ となる。
2. $p_n(x) = \sum_{i=0}^{n} x^i (n = 1, 2, 5, 10)$

5.5 NumPy の NDarray 機能

線形代数の基本であるベクトル (vector) や行列 (matrix) の計算は，NumPy が提供する N 次元配列，すなわち NDarray(N-Dimensional array) の機能を使って行います。以下，ベクトル，行列演算の機能をざっと見ていくことにしましょう。

5.5.1 ベクトルと行列の定義と演算

ベクトルは 1 次元，行列は 2 次元の NDarray として定義します。これによって，リストでは定義されていない，配列単位での加減算やスカラー倍が可能となります。

■ベクトルの定義と演算　$\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$ として

$$\mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix},\ \mathbf{b} = \begin{bmatrix} -3 \\ -2 \\ -1 \end{bmatrix} \tag{5.7}$$

とし，$3\mathbf{a} + \mathbf{b}$ を

$$3\mathbf{a} + \mathbf{b} = 3 \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + \begin{bmatrix} -3 \\ -2 \\ -1 \end{bmatrix} = \begin{bmatrix} 3 \times 1 + (-3) \\ 3 \times 2 + (-2) \\ 3 \times 3 + (-1) \end{bmatrix} = \begin{bmatrix} 0 \\ 4 \\ 8 \end{bmatrix} \tag{5.8}$$

として計算するスクリプトを下記に示す。

ソースコード 5.8　first_vector.py

```
1  # first_vector.py: 基本線型計算
2  import numpy as np # NumPy
3
4  # ベクトル
5  vec_a = np.array([1, 2, 3])
6  vec_b = np.array([-3, -2, -1])
7
8  print('vec_a␣=␣', vec_a)
9  print('vec_b␣=␣', vec_b)
10
11 # ベクトル演算ができる
12 vec_c = 3 * vec_a + vec_b
13 print('vec_c␣=␣', vec_c)
```

上記のスクリプトを実行し，下記のように，$\mathbf{a}$, $\mathbf{b}$ が (5.7) 式で定義した通りになっているか，(5.8) 式に示した演算の結果が得られているかを確認して下さい。

first_vector.py の実行結果

```
vec_a =  [1 2 3]
vec_b =  [-3 -2 -1]
vec_c =  [0 4 8]
```

問題 5.5

$\mathbf{a}, \mathbf{b} \in \mathbb{C}^3$ として

$$\mathbf{a} = \begin{bmatrix} 1+\mathrm{i} \\ 2+2\mathrm{i} \\ 3+3\mathrm{i} \end{bmatrix},\ \mathbf{b} = \begin{bmatrix} -3-3\mathrm{i} \\ -2-2\mathrm{i} \\ -1-\mathrm{i} \end{bmatrix} \tag{5.9}$$

とし，$\sqrt{2}\mathbf{a} - \sqrt{3}\mathbf{b}$ を求めるスクリプト first_cvector.py を作れ。

■行列の定義と演算　$A, B \in \mathbb{R}^{3\times3}$ として

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix},\ B = \begin{bmatrix} -3 & -2 & -1 \\ -2 & -1 & 0 \\ -1 & 0 & 1 \end{bmatrix} \tag{5.10}$$

とし，$3A+B$ を

$$3A+B=3\begin{bmatrix} 3\times1+(-3) & 3\times2+(-2) & 3\times3+(-1) \\ 3\times2+(-2) & 3\times3+(-1) & 3\times4+0 \\ 3\times3+(-1) & 3\times4+0 & 3\times5+1 \end{bmatrix}=\begin{bmatrix} 0 & 4 & 8 \\ 4 & 8 & 12 \\ 8 & 12 & 16 \end{bmatrix} \tag{5.11}$$

として計算するスクリプトを下記に示す。

ソースコード 5.9 first_matrix.py

```
1  # first_matrix.py: 最初の行列演算
2  import numpy as np # NumPy
3
4  # 行列
5  mat_a = np.array([
6      [1, 2, 3],
7      [2, 3, 4],
8      [3, 4, 5]
9  ])
10 mat_b = np.array([
11     [-3, -2, -1],
12     [-2, -1, 0],
13     [-1, 0, 1]
14 ])
15
16 print('mat_a␣=␣\n', mat_a)
17 print('mat_b␣=␣\n', mat_b)
18
19 # 行列演算ができる
20 mat_c = 3 * mat_a + mat_b
21 print('mat_c␣=␣\n', mat_c)
```

上記のスクリプトを実行し，下記のように行列 A, B が (5.10) 式で定義した通りになっているか，(5.11) 式に示した演算の結果が得られているかを確認して下さい。

first_matrix.py の実行結果

```
mat_a =
 [[1 2 3]
 [2 3 4]
 [3 4 5]]
mat_b =
 [[-3 -2 -1]
 [-2 -1  0]
 [-1  0  1]]
mat_c =
 [[ 0  4  8]
 [ 4  8 12]
 [ 8 12 16]]
```

問題 5.6

$A, B \in \mathbb{C}^{3\times 3}$ として

$$A = \begin{bmatrix} 1+\mathrm{i} & 2+2\mathrm{i} & 3+3\mathrm{i} \\ 2+2\mathrm{i} & 3+3\mathrm{i} & 4+4\mathrm{i} \\ 3+3\mathrm{i} & 4+4\mathrm{i} & 5+5\mathrm{i} \end{bmatrix},\ B = \begin{bmatrix} -3-3\mathrm{i} & -2-2\mathrm{i} & -1-\mathrm{i} \\ -2-2\mathrm{i} & -1-\mathrm{i} & 0 \\ -1-\mathrm{i} & 0 & 1+\mathrm{i} \end{bmatrix} \tag{5.12}$$

とし，$(\sqrt{2}+\mathrm{i})A - (\sqrt{3}\mathrm{i})B$ を求めるスクリプト `first_cmatrix.py` を作れ。

■特殊なベクトル，行列の定義　全ての要素が 0 となるベクトルをゼロベクトル (zero vector), すべての要素が 0 となる行列をゼロ行列 (zero matrix) と呼び，それぞれ $\mathbf{0} \in \mathbb{C}^n$, $O \in \mathbb{C}^{n\times n}$ と書きます。また Python では

```
np.zeros(3), np.zeros((3, 3))
```

と記述します。

同様に，全ての要素が 1 となる n 次元ベクトルや n 次正方行列は

```
np.ones(3), np.ones((3, 3))
```

と記述します。

`zeros_ones.py` の実行結果

```
np.zeros(3) =  [0. 0. 0.]
np.zeros((3, 3)) =
 [[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
np.ones(3) =  [1. 1. 1.]
np.ones((3, 3)) =
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

■ドット積と内積，行列・ベクトル積，行列積　ドット積 (dot product) は，二つのベクトル $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$ の各要素 $a_i, b_i \in \mathbb{C}$ の積和のことです。これはちょうどベクトルを $n^{\times}1$ の行列としてみた時，$\mathbf{a}^T\mathbf{b}$ という行列積に相当するものです

$$\mathbf{a}^T\mathbf{b} = \sum_{i=1}^{n} a_i b_i \tag{5.13}$$

これに対し，内積 (inner product) は $\mathbf{a}$ の共役 $\overline{\mathbf{a}} = [\overline{a}_1 \ \ldots \ \overline{a}_n]^T$ と $\mathbf{b}$ のドット積で，$(\mathbf{a}, \mathbf{b})$ と記述します。

$$(\mathbf{a}, \mathbf{b}) = \overline{\mathbf{a}}^T\mathbf{b} = \sum_{i=1}^{n} \overline{a}_i b_i \tag{5.14}$$

これらを NumPy で実行するためには，NDarray クラスに備わっている dot 関数を使用します。

ソースコード 5.10　products.py(1/2)

```
1  # products.py: ベクトルのドット積,内積,行列・ベクトル積,行列積
2  import numpy as np
3
4  # ベクトル
5  vec_a = np.array([1, 2, 3])
6  vec_b = np.array([-3, -2, -1])
7
8  print('vec_a␣=␣', vec_a)
9  print('vec_b␣=␣', vec_b)
10
11 # ドット積
12 ip_ab = vec_a.dot(vec_b)
13 print('(a,␣b)␣=␣', ip_ab)
14
15 # 複素ベクトル
16 vec_c = np.array([1 + 1j, -2 + 2j])
17 vec_d = np.array([-2 + 3j, -1 - 4j])
18 print('vec_c␣=␣', vec_c)
19 print('vec_d␣=␣', vec_d)
20
21 # ドット積と内積
22 dot_cd = vec_c.dot(vec_d)
23 ip_cd = np.conj(vec_c).dot(vec_d)
24 print('␣c␣.␣d␣=␣', dot_cd)
25 print('(c,␣d)␣=␣', ip_cd)
```

行列ベクトル積，行列積もこの dot 関数で計算できます。行列乗算については@(アットマーク）が演算子代わりに使用できます。

ソースコード 5.11　products.py(2/2)

```
36 # 行列
37 mat_a = np.array([[1, 2, 3], [2, 2, 3], [3, 3, 3]])
38 mat_b = np.array([[-3, -3, -3], [-3, -2, -2], [-3, -2, -1]])
39 print('mat_a␣=␣\n', mat_a)
40 print('mat_b␣=␣\n', mat_b)
41
42 # 行列・ベクトル積
43 mat_av = mat_a.dot(vec_a)
44 print('A␣*␣a␣=␣', mat_av)
45
46 # 行列乗算
47 mat_ab = mat_a.dot(mat_b)
48 print('A␣*␣B␣=␣\n', mat_ab)
49 mat_ab_at = mat_a @ mat_b
50 print('A␣@␣B␣=␣\n', mat_ab_at)
```

products.py の実行結果

```
vec_a =  [1 2 3]
vec_b =  [-3 -2 -1]
(a, b) =  -10
vec_c =  [ 1.+1.j -2.+2.j]
vec_d =  [-2.+3.j -1.-4.j]
 c . d =  (5+7j)
(c, d) =  (-5+15j)
mat_a =
 [[1 2 3]
 [2 2 3]
 [3 3 3]]
mat_b =
 [[-3 -3 -3]
 [-3 -2 -2]
 [-3 -2 -1]]
A * a =  [14 15 18]
A * B =
 [[-18 -13 -10]
 [-21 -16 -13]
 [-27 -21 -18]]
A @ B =
 [[-18 -13 -10]
 [-21 -16 -13]
 [-27 -21 -18]]
```

■単位行列と逆行列　単位行列 I_n は，数字の 1 に相当するもので

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \tag{5.15}$$

という対角行列 (diagonal matrix) です。

n 次正方行列 $A \in \mathbb{C}^{n\times n}$ が $\det(A) \neq 0$ の時，逆行列 A^{-1} を持つことが知られています。これは

$$AA^{-1} = A^{-1}A = I_n \tag{5.16}$$

という性質があります。

以下に，単位行列の定義，逆行列の導出と，(5.16) 式が成立することを確認するスクリプトを示します。

ソースコード 5.12　inv_eye.py

```
1 # inv_eys.py: 単位行列と逆行列
```

```
2  import numpy as np
3
4  # 単位行列 I
5  print('Unit matrix:\n', np.eye(3)) # 正方行列
6  print('Unit matrix:\n', np.eye(3, 3)) # 行数と列数の指定
7  print('Unit matrix:\n', np.diag([1, 1, 1])) # 対角行列
8
9  # 逆行列
10
11 # 元の行列
12 mat_a = np.array([
13     [3, 4, 0],
14     [4, 3, 4],
15     [3, 4, 3]
16 ])
17
18 print('A =\n', mat_a)
19
20 # 逆行列は存在するか?
21 # det(A) != 0 ?
22 print('det(A) = ', np.linalg.det(mat_a))
23 # rank(A) == n ?
24 print('rank(A) = ', np.linalg.matrix_rank(mat_a))
25
26 # NumPy の線型代数パッケージ(linalg)
27 # で逆行列 (inv)を導出
28 inv_mat_a = np.linalg.inv(mat_a)
29 print('A^(-1) =\n', inv_mat_a)
30
31 # 行列 * 逆行列
32 print('A * A^(-1) =\n', mat_a @ inv_mat_a)
33 print('A^(-1) * A =\n', inv_mat_a @ mat_a)
```

これを実行すると，下記のように単位行列，逆行列が導出でき，(5.16) 式が近似的に成立することも見て取れます。

inv_eye.py の実行結果

```
Unit matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Unit matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
Unit matrix:
 [[1 0 0]
 [0 1 0]
 [0 0 1]]
A =
 [[3 4 0]
 [4 3 4]
 [3 4 3]]
det(A) =  -21.0
rank(A) =  3
A^(-1) =
 [[ 0.33333333  0.57142857 -0.76190476]
 [ 0.         -0.42857143  0.57142857]
 [-0.33333333  0.          0.33333333]]
A * A^(-1) =
 [[1.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 1.00000000e+00 0.00000000e+00]
 [5.55111512e-17 0.00000000e+00 1.00000000e+00]]
A^(-1) * A =
 [[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00]
 [-5.55111512e-17  0.00000000e+00  1.00000000e+00]]
```

■ベクトル，行列のノルム　ノルム (norm) とは，ベクトル $\mathbf{v} \in \mathbb{C}^n$ や行列 $A \in \mathbb{C}^{n\times n}$ から正の実数への関数の一種で，$\|\mathbf{v}\|_p$，$\|A\|_p (\geq 0)$ と書き，絶対値が持つ次の性質を持っているものの呼称です。

正値性　全てのベクトル $\mathbf{v} \in \mathbb{C}^n$ や行列 $A \in \mathbb{C}^{n\times n}$ に対し，必ず $\|\mathbf{v}\|_p$, $\|A\|_p \geq 0$。また，$\mathbf{v} = 0$, $A = O$ の時にのみ $\|\mathbf{v}\|_p$, $\|A\|_p = 0$ となる。

スカラー倍　定数 $\alpha \in \mathbb{C}$ に対し，$\|\alpha\mathbf{v}\|_p = |\alpha| \cdot \|\mathbf{v}\|_p$, $\|\alpha A\|_p = |\alpha| \cdot \|A\|_p$。

三角方程式　$\mathbf{v}, \mathbf{w} \in \mathbb{C}^n$, $A, B \in \mathbb{C}^{n\times n}$ に対し

$$\begin{aligned} \|\mathbf{v}+\mathbf{w}\|_p &\le \|\mathbf{v}\|_p + \|\mathbf{w}\|_p \\ \|A+B\|_p &\le \|A\|_p + \|B\|_p \end{aligned} \tag{5.17}$$

ベクトルにしろ行列にしろ，サイズが大きくなると比較が難しくなります。そこで活躍するのがこのノルムです。よく使用するのは，$p = 1, 2, \infty$ で，ベクトルノルムでは

$$\text{1 ノルム}\quad \|\mathbf{a}\|_1 = \sum_{i=1}^{n} |a_i|$$

$$\text{ユークリッドノルム}\quad \|\mathbf{a}\|_2 = \sqrt{\sum_{i=1}^{n} |a_i|^2} \quad = \sqrt{(\mathbf{a},\mathbf{a})}$$

$$\text{無限大ノルム}\quad \|\mathbf{a}\|_\infty = \max_i |a_i|$$

となり，行列ノルムでは

$$\text{1 ノルム}\quad \|A\|_1 = \max_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_1}{\|\mathbf{x}\|_1} = \max_j \sum_{i=1}^{n} |a_{ij}|$$

$$\text{ユークリッドノルム}\quad \|A\|_2 = \max_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_2}{\|\mathbf{x}\|_2} = \sqrt{\max_i \lambda_i(A^*A)}\ \text{（ここで } \lambda_i(A) \text{ は } A \text{ の固有値）}$$

$$\text{無限大ノルム}\quad \|A\|_\infty = \max_{\mathbf{x}\neq 0} \frac{\|A\mathbf{x}\|_\infty}{\|\mathbf{x}\|_\infty} = \max_i \sum_{j=1}^{n} |a_{ij}|$$

となります。また行列ノルムではフロベニウスノルムというものも使用されます。これは行列要素を縦に並べてベクトル化した時のユークリッドノルムに相当します。

$$\|A\|_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n} |a_{ij}|^2}$$

これらのノルムを計算する NumPy の機能は全て線形代数モジュール (linalg) にまとめられています。

ソースコード 5.13　norm.py

```
1  # norm.py: ベクトルと行列のノルム
2  import numpy as np
3
4  # ベクトル
5  vec_a = np.array([1, 2, 3])
6  print('vec_a␣=␣', vec_a)
7
8  # 行列
9  mat_a = np.array([[1, 2, 3], [2, 2, 3], [3, 3, 3]])
10 print('mat_a␣=␣\n', mat_a)
11
12 # ユークリッドノルム
13 print('||vec_a||_2␣=␣', np.linalg.norm(vec_a))
14 print('||mat_a||_2␣=␣', np.linalg.norm(mat_a))
15
16 # 1ノルム
17 print('||vec_a||_1␣=␣', np.linalg.norm(vec_a, 1))
```

```
18 print('||mat_a||_1␣=␣', np.linalg.norm(mat_a, 1))
19
20 # 無限大ノルム
21 print('||vec_a||_inf␣=␣', np.linalg.norm(vec_a, np.inf))
22 print('||mat_a||_inf␣=␣', np.linalg.norm(mat_a, np.inf))
23
24 # フロベニウスノルム(行列のみ)
25 print('||mat_a||_F␣=␣', np.linalg.norm(mat_a, 'fro'))
```

手計算では確認が難しい行列のユークリッドノルムも含め，全てのノルムが計算できていることが確認できます。

norm.py の実行結果

```
vec_a =  [1 2 3]
mat_a =
 [[1 2 3]
 [2 2 3]
 [3 3 3]]
||vec_a||_2 =  3.7416573867739413
||mat_a||_2 =  7.615773105863909
||vec_a||_1 =  6.0
||mat_a||_1 =  9.0
||vec_a||_inf =  3.0
||mat_a||_inf =  9.0
||mat_a||_F =  7.615773105863909
```

問題 5.7

1. $\|\mathbf{a}\|_2 = \sqrt{(\mathbf{a}, \mathbf{a})}$ であることを確認する Python スクリプトを作れ。
2. ベクトル $\mathbf{v}$, $\mathbf{w}$ と行列 A, B が与えられているとき，三角不等式が成立することを確認する Python スクリプトを作れ。

5.6 乱数列の生成，統計関数，行列乗算ベンチマーク

現在の AI を支える根本技術である深層学習は，適当な非ゼロの値から出発し，学習を重ねながら正解を導くようにニューロンの重みづけを変更していきます。この時使用される「適当な非ゼロの値」は乱数 (random number) として数学的に生成されたものです。まずは NumPy から乱数を NDarray としてベクトルや行列として生成し，その平均 (average), 分散 (variant), 標準偏差 (standard deviation) を計算するスクリプトを動かしてみましょう。

ソースコード 5.14　random_test.py

```
1 # random_test.py: 乱数と統計関数
2 import numpy as np # NumPy
3
4 # 乱数seed の指定
5 rng = np.random.default_rng(seed=20240405)
```

```
6
7  # 乱数列生成
8  rand_vec = rng.random(3)
9  print('rand_vec,␣type(rand_vec)␣=␣', rand_vec, type(rand_vec))
10 # 平均,分散,標準偏差
11 print('ave,␣var,␣std␣=␣%7.3g,␣%7.3g,␣%7.3g' % (np.average(rand_vec), np.var(rand_vec),
       np.std(rand_vec)))
12
13 # 乱数行列生成
14 rand_mat = rng.random((3, 3))
15 print('rand_mat␣=␣\n', rand_mat)
16 print('ave,␣var,␣std␣=␣%7.3g,␣%7.3g,␣%7.3g' % (np.average(rand_mat), np.var(rand_mat),
       np.std(rand_mat)))
```

これを実行すると下記のようになります。乱数ですので，環境によって数字は異なります。

norm.py の実行結果

```
rand_vec, type(rand_vec) =  [0.04816281 0.77801654 0.9651537 ] <class 'numpy.ndarray'>
ave, var, std =   0.597,   0.157,   0.396
rand_mat =
 [[0.57626548 0.94308848 0.17285614]
 [0.3705871  0.11832061 0.01597963]
 [0.13692318 0.25497593 0.93845524]]
ave, var, std =   0.392,   0.109,   0.331
```

本章の最後に，行列積の計算時間を計測するスクリプトを作ってみましょう。時間計測は time パッケージの time 関数を使っています。

ソースコード 5.15　matmul_bench.py

```
1  # matmul_bench.py: 行列乗算ベンチマーク
2  import numpy as np # NumPy
3  from time import time # 時間計測
4
5  # 乱数seedの指定
6  rng = np.random.default_rng(seed=20240405)
7
8  # 行列サイズ入力
9  str_dim = input('Input␣dimension␣=␣')
10 dim = int(str_dim)
11
12 # 乱数行列生成
13 mat_a = rng.random((dim, dim))
14 mat_b = rng.random((dim, dim))
15
16 # 行列乗算 1
17 stime = time()
18 mat_c_dot = mat_a.dot(mat_b)
19 dot_time = time() - stime
20
21 # 行列乗算 2
22 stime = time()
```

```
mat_c_at = mat_a @ mat_b
at_time = time() - stime

# 出力
print('||C_dot||_F␣=␣%25.17e' % np.linalg.norm(mat_c_dot, 'fro'))
print('||C_at␣||_F␣=␣%25.17e' % np.linalg.norm(mat_c_at , 'fro'))
print('s(C_dot),␣s(C_at)␣=␣%7.3f,␣%7.3f' % (dot_time, at_time))
```

問題 5.8

matmul_bench.py を自分の PC 上で実行し，$\dim = 2000, 5000, 10000$ の時の計算時間を計測し，その変化について考察せよ。その際，計算時間は 10 回反復した平均値を使用すること。また標準偏差も導出すること。