2018年航空宇宙情報システム学第２
第2部「プログラミングと数値計算」

Numpy による数値計算入門１
行列・ベクトル演算

2018年6月26日

NumPyとは？

- 公式サイト: http://www.numpy.org/
- Pythonで科学計算を行うためのライブラリ
  - 行列・ベクトルを表現するための多次元配列（multi-dimensional array）を提供
  - 線形代数計算用の関数群を提供
  - 高い計算効率
  - Matlabの代替として使われることも
- 様々な機能を持つが、今回は行列・ベクトル計算に焦点を当てて説明
- Matplotlib（グラフ描画），SciPy（科学技術計算）などとセットで利用されることも多い

NumPyのインポート

- ECCSの環境（Anaconda）では既にインストールされているので、すぐに利用できる

  >> import numpy

  ただし、これだと、numpy.array([1.5, 2.0])のように、呼び出す必要があり不便

  >> import numpy as np

  とすれば、np.array([1.5,2.0]) と記述が短くなる。さらに、

  >>from numpy import *

  とすれば、array([1.5, 2.0])で済む。
  （ただし、名前の衝突に注意すること。）

Array型による行列・ベクトルの表現

- NumPyでは行列やベクトルをarrayで表現する
  - リストと比べて数値計算に最適化されている
- 行列は2次元Array型を用いて表現する
- 作成例：

```
>>> A = array([[1.0,2.0,0.0],[-1.0,3.0,2.0],[0.0,-1.0,1.0]])
>>> print A
[[ 1.  2.  0.]
 [-1.  3.  2.]
 [ 0. -1.  1.]]
```

数値のリストのリストから作成

- リスト同様、インデックスは0から始まる

Array 型によるベクトル表現

[方法1] 1次元arrayによるベクトル表現

```
>>> b = array([1.0, 2.0, 3.0])
>>> print b
[ 1.  2.  3.]  # 行・列ベクトルの区別無し
```

[方法2] 2次元array（行列）によるベクトル表現 :

```
>>> b = array([[1.0, 2.0, 3.0]])
>>> print b
[[ 1.  2.  3.]]  # 行ベクトル
>>> print b.T  # 転置
[[ 1.]
 [ 2.]
 [ 3.]]  # 列ベクトル
```

- 特別な事情が無ければ、シンプルな[方法1]を採用

ベクトルの要素（成分）

- リストとほぼ同様にアクセスできる

```
>>> a = array([0.0, 1.0, 2.0])
>>> print a[1]
1.0
```

- 変更可能（リストと同じ）

```
>>> a[1] = -1.0
>>> print a
[ 0. -1.  2.]
```

- スライス（これもリストとほぼ同じ）

```
>>> print a[1:]
[-1.  2.]
```

特殊なベクトルの生成

- 零ベクトル、1ベクトル

```
>>> a = zeros(3)
>>> b = ones(3)
>>> print(a,b)
[ 0.  0.  0.] [ 1.  1.  1.]
```

- arange, linspace: 等間隔のベクトルを生成

```
>>> a = arange(1.0, 10.0, 4.0)   #間隔（幅）を指定
>>> b = linspace(1.0, 10.0, 4)   #要素数（次元）を指定
>>> print a,b
[ 1.  5.  9.] [  1.   4.   7.  10.]
```

- arange(n) は、ベクトル [0,1,...,n-1] を生成

ベクトルの代入と複製

- リストやクラスオブジェクトの場合と同様

```
>>> a = arange(3)  # a = array([0,1,2])と同じ
>>> b = a   # 別名を付けているだけで複製にならない
>>> b[0] = -1.0
>>> print(a)
[-1.  1.  2.]
```

- 複製を作る方法はいろいろある

```
>>> b = a.copy()   # copyメソッドを使う
>>> b = array(a)   # a と同じ中身のベクトルをもう一つ作る
>>> b = a[:]  # スライスを使ってコピーする
```

ベクトル演算(1):足し算・引き算

- 足し算・引き算(同次元のベクトル同士)

```
>>> a = array([1.0, 2.0, 3.0])
>>> b = array([2.0, 4.0, 5.0])
>>> a+b   # または、add(a,b)
array([ 3.,  6.,  8.])
>>> a-b    # または、subtract(a,b)
array([-1., -2., -2.])
```

- ベクトルにスカラーを足す/引く場合

```
>>> a+1.0
array([ 2.,  3.,  4.])   # 全要素に足される
```

ベクトル演算（2）：要素同士の掛け・割り算

- 要素同士の掛け算・割り算

```
>>> a*b   #またはmultiply(a,b)
array([  2.,   8.,  15.])
>>> a/b   #またはdivide(a,b)
array([ 0.5,  0.5,  0.6])
```

- スカラーをベクトルに掛ける/割る

```
>>> 2*a
array([ 2.,  4.,  6.])
>>> a/2
array([ 0.5,  1. ,  1.5])
```

ベクトル演算（3）：要素単位の関数

- 各要素ごとに適用される関数（ufunc と呼ばれる）

```
>>> absolute(a-b)  #要素ごとに絶対値を計算
array([ 1.,  2.,  2.])
>>> exp(a)  #要素ごとにexp(x)を計算
array([  2.71828183,   7.3890561 ,  20.08553692])
>>> sqrt(a) #要素ごとに平方根
array([ 1.        ,  1.41421356,  1.73205081])
>>> sin(a)   # sin, tan, arcsin 等も同様
array([ 0.84147098,  0.90929743,  0.14112001])
```

- 他にも多数の算術関数あり

ベクトル演算（4）：内積と外積

- 内積（inner product, dot product）

  >>> vdot(a,b)　# ベクトルaとベクトルbの内積

  25.0　$\boldsymbol{a} \cdot \boldsymbol{b} = \boldsymbol{a}^T \boldsymbol{b} = a_1 b_1 + a_2 b_2 + \ldots + a_D b_D$ （結果はスカラー）

  – 1次元arrayの場合、dot(a,b) と inner(a,b)も同じ結果

- 外積（outer product）

  – もうひとつの外積（exterior product）とは別物!

```
>>> outer(a,b)
array([[  2.,   4.,   5.],
       [  4.,   8.,  10.],
       [  6.,  12.,  15.]])
```

$$\boldsymbol{a} \otimes \boldsymbol{b} = \boldsymbol{a}\boldsymbol{b}^T = \begin{bmatrix} a_1 \\ \vdots \\ a_D \end{bmatrix} \begin{bmatrix} b_1 & \cdots & b_D \end{bmatrix}$$

（結果は行列）

ベクトル演算(5):ユークリッドノルム

1. linalgサブパッケージのnorm関数を使う

```
>>> linalg.norm(a)  #ユークリッドノルム(2-ノルム)
3.7416573867739413
```

– より一般にp-ノルムを計算することも可能

```
>>> linalg.norm(a-b,1)  #a-bの1ノルム
5.0
>>> linalg.norm(a-b,inf) #無限大ノルム
2.0
```

2. 定義に従って計算　$\|\boldsymbol{a}\|_2 = \sqrt{\boldsymbol{a} \cdot \boldsymbol{a}}$

```
>>> sqrt(vdot(a,a))
3.7416573867739413
```

行列の要素

- 単独要素へのアクセス

```
>>> A = array([[1.0,2.0],[3.0,4.0]])
>>> A[0,0]   # インデックスが0から始まることに注意
1.0
```

  – 要素単位で値を変更可能

- 行列から(行/列)ベクトルを取り出す

```
>>> A[:,0]   # 列ベクトル
array([ 1.,  3.])
>>> A[1,:]   # 行ベクトル
array([ 3.,  4.])
```

A[:,0]

A[1,:]

$$\begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

行列演算(1):足し算・引き算

- 行列同士の足し算、引き算

$$A = \begin{bmatrix} 1.0 & 2.0 \\ -2.0 & 1.0 \end{bmatrix}$$

$$B = \begin{bmatrix} 2.0 & -1.0 \\ 2.0 & 1.0 \end{bmatrix}$$

```
>>> A = array([[1.0, 2.0],[-2.0,1.0]])
>>> B = array([[2.0, -1.0],[2.0, 1.0]])
>>> A+B
array([[ 3.,  1.],
       [ 0.,  2.]])
```

- スカラーを足すと全ての要素に適用される

```
>>> A+1
array([[ 2.,  3.],
       [-1.,  2.]])
```

行列演算（２）：掛け算

$$\boldsymbol{A} = \begin{bmatrix} 1.0 & 2.0 \\ -2.0 & 1.0 \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} 2.0 & -1.0 \\ 2.0 & 1.0 \end{bmatrix}$$

- スカラー倍

```
>>> 2.5*A
array([[ 2.5,  5. ],
       [-5. ,  2.5]])
```

- 行列同士の掛け算：dot関数を使う

```
>>> dot(A,B)    # 行列同士の積（行列積）
array([[ 6.,  1.],
       [-2.,  3.]])
>>> A*B    # 要素同士の積（要素積）
array([[ 2., -2.],
       [-4.,  1.]])
```

行列演算（３）：転置・変形

- 転置

```
>>> A = array([[1.0,2.0],[3.0,4.0]])
>>> A.T    # A.transpose() でも同じ結果
array([[ 1.,  3.],
       [ 2.,  4.]])
```

$$\boldsymbol{A} = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

- 変形（行数・列数の変更）

```
>>> linspace(0.0,11.0,12).reshape(3,4)  #3x4行列に変形
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])
>>> A.reshape(4) # 2x2行列を4次元ベクトルに変換
array([ 1.,  2.,  3.,  4.])
```

行列演算(4):単位行列・零行列

- 単位行列

```
>>> eye(3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

- 零行列

```
>>> zeros((2,2))  #タプルでサイズを与えている
array([[ 0.,  0.],
       [ 0.,  0.]])
```

  – 全成分が1の行列も ones((2,2))のように作成

行列演算(5):連結

- 水平方向に連結

```
>>> C = hstack((A,B))
>>> print C
[[1 2 4 3]
 [3 4 2 1]]
```

$$\boldsymbol{A} = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} 4.0 & 3.0 \\ 2.0 & 1.0 \end{bmatrix}$$

の場合

- 垂直方向に連結

```
>>> D = vstack((A,B))
>>> print D
[[1 2]
 [3 4]
 [4 3]
 [2 1]]
```

(参考)行列とベクトルの連結

$$\boldsymbol{b} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

```
>>> b = array([1,1])
>>> E = hstack((A,b.reshape((2,1))))
```

2x1 の列ベクトルに変形してから連結する必要がある

array型の連結には、stack, append, concatenate などの関数も使うことができる。

行列演算（6）：逆行列・行列式

- 逆行列：linalg サブモジュールのinv関数

```
>>> linalg.inv(A)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> dot(A,linalg.inv(A)) #元の行列との積を確認
array([[  1.00000000e+00,   1.11022302e-16],
       [  0.00000000e+00,   1.00000000e+00]])
```

厳密に0にならない

- 行列式

```
>>> linalg.det(A)
-2.0000000000000004
```

厳密に2にならない

行列演算(7):固有値･固有ベクトル

- 行列Aの固有値、固有ベクトル

```
>>> A = array([[1.0,2.0],[3.0,4.0]])
>>> lmd,V = linalg.eig(A)
>>> lmd
array([-0.37228132,  5.37228132])  # 2つの固有値
>>> V
array([[-0.82456484, -0.41597356],  #各列が固有ベクトル
       [ 0.56576746, -0.90937671]])
```

  - Aが実対称行列の場合は linalg.eigh()関数を使った方が良い

行列とベクトルの積

- dot関数を用いる

```
>>> A = array([[1.0,2.0],[3.0,4.0]])
>>> b = array([-1.0, 2.0])
>>> dot(A,b)    # A*bではない!
array([ 3.,  5.])
```

$$\boldsymbol{A} = \begin{bmatrix} 1.0 & 2.0 \\ 3.0 & 4.0 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} -1.0 \\ 2.0 \end{bmatrix}$$

ちなみに、dot(b,A)とすると、

```
>>> dot(b,A)
array([ 5.,  6.])
```

\# $\boldsymbol{b}^T \boldsymbol{A}$ を計算する

連立一次方程式の解

- $\boldsymbol{A}$ (n × n行列), $\boldsymbol{b}$ (n次元ベクトル)に対して、
  $\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b}$ を満たす 解ベクトル $\boldsymbol{x}$ を求める

```
>>> A = array([[-1.0, 1.0, 1.0],[2.0,-1.0, 1.0],[1.0, 0.0, -1.0]])
>>> b = array([0.0, -1.0, 2.0])
>>> x = linalg.solve(A,b)
>>> print x
[ 1.  2. -1.]
>>> dot(A,x)   # 検算
array([ 0., -1.,  2.])
>>> dot(linalg.inv(A),b)  # x = A^-1 bを計算
array([ 1.,  2., -1.])
```

$$\boldsymbol{A} = \begin{bmatrix} -1 & 1 & 1 \\ 2 & -1 & 1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} 0 \\ -1 \\ 2 \end{bmatrix}$$

乱数生成（numpy.randomモジュール）

1. 0から1までの一様乱数 (random 関数)

```
>>> np.random.random(3)  # 3個サンプリング
array([ 0.64446154,  0.65876711,  0.1386073 ])
```

2. 平均0, 標準偏差1の標準正規分布に従う乱数 (randn 関数)

```
>>> np.random.randn(3)
array([ 1.28688442, -0.41394705, -0.31861007])
```

3. 0からK-1までの整数からランダムに選択(choice 関数)

```
>>> np.random.choice(10,5) #5個サンプリング（復元抽出）
array([6, 8, 4, 6, 0])
```

複素数

- pythonでは最初から複素数が使える

```
>>> a,b = 1.0+2.0j, 2.0+3.0j
>>> a*b
(-4+7j)
```

- Numpy でも普通に使える

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ -1 & 2 \end{bmatrix}$$

```
>>> A = array([[2,3],[-1,2]])
>>> linalg.eig(A)
(array([ 2.+1.73205081j,  2.-1.73205081j]),
 array([[ 0.8660254+0.j ,  0.8660254-0.j ],
        [ 0.0000000+0.5j,  0.0000000-0.5j]]))
```

固有値 $2 \pm \sqrt{3}\ i$

固有ベクトル
（列ベクトル）

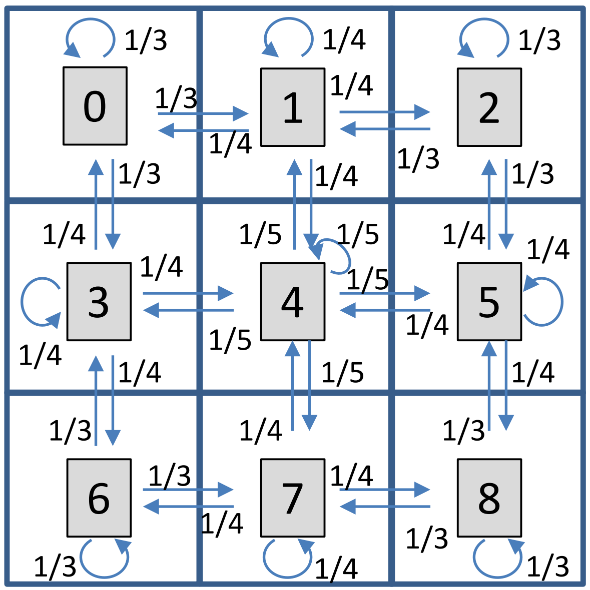
今回の課題（マルコフ連鎖の定常分布）

次ページの図のような3×3のマス目があり、0から8の番号が付いているとする。最初、駒が0番マスにあり、各時刻ごとに次の規則に従ってランダムに移動する。十分に時間が経過したとき、各マスに居る確率を求めなさい。

[移動規則]

- 角のマス(0,2,6,8)に居る時は、確率1/3ずつで隣のマスに移動するか現在のマスに留まる。
- 辺のマス(1,3,5,7)に居る時は、確率1/4ずつで隣のマスに移動するか現在のマスに留まる。
- 真ん中のマス(4)に居る時は、確率1/5ずつで隣のマスに移動するか現在のマスに留まる。

今回の課題(続き)



確率遷移行列 T

(i,j) 成分は、マスjから
マスiに遷移する確率

$$\mathbf{T}=\begin{bmatrix} 1/3 & 1/4 & 0 & 1/4 & 0 & 0 & 0 & 0 & 0 \\ 1/3 & 1/4 & 1/3 & 0 & 1/5 & 0 & 0 & 0 & 0 \\ 0 & 1/4 & 1/3 & 0 & 0 & 1/4 & 0 & 0 & 0 \\ 1/3 & 0 & 0 & 1/4 & 1/5 & 0 & 1/3 & 0 & 0 \\ 0 & 1/4 & 0 & 1/4 & 1/5 & 1/4 & 0 & 1/4 & 0 \\ 0 & 0 & 1/3 & 0 & 1/5 & 1/4 & 0 & 0 & 1/3 \\ 0 & 0 & 0 & 1/4 & 0 & 0 & 1/3 & 1/4 & 0 \\ 0 & 0 & 0 & 0 & 1/5 & 0 & 1/3 & 1/4 & 1/3 \\ 0 & 0 & 0 & 0 & 0 & 1/4 & 0 & 1/4 & 1/3 \end{bmatrix}$$

時刻 t のときに、各マスに居る確率を9次元ベクトル $\mathbf{p}_t$ で表すと、

$$\mathbf{p}_{t+1}=\mathbf{T}\mathbf{p}_t$$

が成り立つ。求めたいのは、$\mathbf{p}_\infty=\lim_{t\to\infty}\mathbf{p}_t$

今回の課題（続き）

様々な解き方が考えられる

1. 確率遷移の式に従って、$\mathbf{p}_t$ が定常になるまで計算する。
2. 定常状態では、$\mathbf{p}_\infty = \mathbf{T}\,\mathbf{p}_\infty$ になることを利用して、固有値・固有ベクトルから求める。
   - $\boldsymbol{T}$ の最大固有値(=1)に対応する固有ベクトルが $\boldsymbol{p}_\infty$
   - linalg.eig 関数を使っても良いが、固有値が分かっているので容易に計算できる。
3. 移動規則に従って、ランダムシミュレーションを行い、ヒストグラム（頻度）を計算して求める。（モンテカルロ法）

- homework06.py という名前を付けて、ITC-LMSから提出
- 締切: 7月3日午前8時（次々回授業の直前）

発展課題

余裕のある人は以下の問題も考えてみよう。

- 遷移行列 T の成分を手で入力するのは面倒。規則性を用いて効率的に求める方法を考えてみよう。
- 対称性を考えれば、3×3 の行列を使って定常確率分布を計算することもできるはず。
- 定常分布が一様分布（つまり、すべてのマス目に居る確率が等しい）になるような遷移規則は存在するだろうか？
  - ただし、たてよこ1マスずつしか動けないという規則は維持するとする。