第 6 章

基盤モジュール: SciPy

NumPy が提供する NDarray や数学関数をはじめとする基本機能をベースに，より複雑な統計計算，科学技術計算を行うためのパッケージが SciPy です。ここでは，高度な線形代数問題，積分，常微分方程式の初期値問題を例に，その機能の一部を紹介していきます。

6.1 連立一次方程式の求解

n 次正方行列 A と n 次元ベクトル $\mathbf{b}$ が与えられた時

$$A\mathbf{x} = \mathbf{b} \tag{6.1}$$

を満足する未知の n 次元ベクトル $\mathbf{x}$ を求める問題が連立一次方程式 (linear system of equations) の求解問題です。もし行列 A が逆行列 A^{-1} を持てば，左右両辺にこれを左から乗じて

$$\mathbf{x} = A^{-1}\mathbf{b}$$

が解となりますが，このやり方では計算量が多く，次元数 n が大きくなると時間を要します。計算量の少ない方法がありますので，Python では SciPy の線形代数パッケージ (linalg) の `solve` 関数を用いて解いてみましょう。この中では解いた結果の検証をノルム相対誤差

$$rE_p(\widetilde{\mathbf{x}}) = \begin{cases} \frac{\|\widetilde{\mathbf{x}}-\mathbf{x}\|_p}{\|\mathbf{x}\|_p} & (\mathbf{x} \neq 0) \\ \|\widetilde{\mathbf{x}}-\mathbf{x}\|_p & (\mathbf{x} = 0) \end{cases} \tag{6.2}$$

で行っています。p は使用したノルム $(1, 2, \infty)$ が入ります。

ソースコード 6.1 `linsolve.py`

```
1  # linsolve.py: 連立一次方程式を高速に解く
2  import numpy as np # NumPy
3  import scipy.linalg as sclinalg # Scipy.linalg
4  import time # time 関数
5
6  # 乱数seed の指定
7  rng = np.random.default_rng(seed=20240521)
8
9  # 正方行列の次数
10 input_str = input('Input␣size␣of␣square␣matrix␣>␣')
11 n = int(input_str)
```

```
12
13 # 乱数行列生成
14 mat_a = rng.random((n, n)) # n x n 行列
15 print('||mat_a||_2␣=␣', np.linalg.norm(mat_a))
16 # 解を生成
17 true_x = rng.random((n, 1)) # n 次元ベクトル
18 print('||x||_2␣=␣', np.linalg.norm(true_x))
19
20 # 定数ベクトルb を生成
21 b = mat_a @ true_x
22
23 # (1)逆行列を求めて連立一次方程式を解く
24 start_time = time.time()
25 inv_x = sclinalg.inv(mat_a) @ b
26 end_time_inv = time.time() - start_time
27 relerr_inv = np.linalg.norm(inv_x - true_x) / np.linalg.norm(true_x)
28
29 # (2) solve 関数を用いて連立一次方程式を解く
30 start_time = time.time()
31 solve_x = sclinalg.solve(mat_a, b)
32 end_time_solve = time.time() - start_time
33 relerr_solve = np.linalg.norm(solve_x - true_x) / np.linalg.norm(true_x)
34
35 # ノルム相対誤差と計算時間の表示
36 print( '␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣inv␣␣␣␣␣␣␣␣␣␣␣solve')
37 print(f'Computational␣time␣(s):␣{end_time_inv:10.2g},␣{end_time_solve:10.2g}')
38 print(f'Norm2␣Relative␣error␣␣:␣{relerr_inv:10.2e},␣{relerr_solve:10.2e}')
```

これを実行してみると，次元数が大きくなると inv 関数を用いるより，solve 関数を用いた方が高速に解けることが分かります。

linsolve.py の実行結果

```
Input size of square matrix > 10000
||mat_a||_2 =  5773.377361270599
||x||_2 =  57.722332431809136
                                inv       solve
Computational time (s):          32,          13
Norm2 Relative error  :   2.44e-11,   2.93e-12
```

問題 6.1

linsolve.py を改良し，A, $\mathbf{b}$ をそれぞれ n 次複素正方行列, n 次元複素ベクトルとして与える clinsolve.py を作成し，

1. inv 関数を使用して $\mathbf{x} := A^{-1}\mathbf{b}$ を計算
2. solve 関数を使用して $\mathbf{x}$ を導出

の計算時間とノルム相対誤差をそれぞれ求められるようにして下さい。

6.2 正方行列の固有値と固有ベクトル

n 次正方行列 $A \in \mathbb{C}^{n\times n}$ に対して，定数 $\lambda \in \mathbb{C}$ と非ゼロ n 次ベクトル $\mathbf{v} \in \mathbb{C}^n$ が

$$A\mathbf{v} = \lambda\mathbf{v} \tag{6.3}$$

という関係を満足するとき，λ を A の右固有値 (right eigenvalue)，$\mathbf{v}$ を λ に対応する A の右固有ベクト (right eigenvector) と呼びます。また，定数 $\varpi \in \mathbb{C}$ と非ゼロ n 次ベクトル $\mathbf{w} \in \mathbb{C}^n$ に対して

$$\overline{A}^T\mathbf{w} = \varpi\mathbf{w} \tag{6.4}$$

という関係を満足するとき，ω を A の左固有値 (left eigenvalue)，$\mathbf{w}$ を ω に対応する A の左固有ベクトルと呼びます。定義式 (6.4) は

$$\overline{\mathbf{w}^T}A = \omega\overline{\mathbf{w}^T} \tag{6.5}$$

とも定義されます。

単に固有値，固有ベクトルという時には一般には右固有値，右固有ベクトルを意味します。

NumPy にも固有値・固有ベクトルを求める `np.linalg.eig` 関数はありますが，SciPy の `scipy.linalg.eig` 関数の方がより多くのバリエーションに対応していますので，特に支障ない場合はこちらの利用をお勧めします。

ソースコード 6.2 `eig.py`

```
1  # eig.py: 正方行列の固有値・固有ベクトル
2  import numpy as np # NumPy
3  import scipy.linalg as sclinalg # SciPy.linalg
4
5  # 乱数seedの指定
6  rng = np.random.default_rng(seed=20240521)
7
8  # 正方行列の次数
9  input_str = input('Input␣size␣of␣square␣matrix␣>␣')
10 n = int(input_str)
11
12 # 乱数行列生成
13 mat_a = rng.random((n, n))
14 print('mat_a␣=␣\n', mat_a)
15
16 # 全ての固有値
17 eigen_values = sclinalg.eig(mat_a)
18
19 # 固有値，(右)固有ベクトル
20 eigen_values, Vr = sclinalg.eig(mat_a, right=True)
21 print('Eigenvalues␣=␣', eigen_values)
22 print('Right␣eigenvectors␣=␣', Vr)
23
24 # 固有値,左固有ベクトル，(右)固有ベクトル
25 eigen_values, Vl, Vr = sclinalg.eig(mat_a, left=True, right=True)
26 print('Eigenvalues␣=␣', eigen_values)
27 print('Left␣eigenvectors␣=␣\n', Vl)
28 print('Right␣eigenvectors␣=␣\n', Vr)
```

```
29
30 print('index␣||(A␣-␣eig␣*␣I)␣Vr␣||␣/␣||Vr||␣␣||(A␣-␣eig␣*␣I)␣Vl||␣/␣||Vl||')
31 for index in range(n):
32   # A * Vr == Labmda * Vr ?
33   print(f'{index:5d}␣{np.linalg.norm((mat_a␣-␣eigen_values[index]␣*␣np.eye(n))␣@␣Vr[:,
         ␣index])␣/␣np.linalg.norm(Vr[:,␣index]):30.17e}␣{np.linalg.norm((mat_a.T␣-␣
         eigen_values[index].conj()␣*␣np.eye(n))␣@␣Vl[:,␣index])␣/␣np.linalg.norm(Vl[:,␣
         index]):30.17e}')
```

乱数行列の左右固有値，固有ベクトルを求め，検算として定義式 (6.3), (6.4) を満足しているか，ノルム相対誤差を求めて確認しています。例えば $n = 2$ の場合はという結果を得ることができます。

eig.py の実行結果

```
Input size of square matrix > 2
mat_a =
 [[0.7763316  0.54878817]
 [0.97934203 0.65359049]]
Eigenvalues =  [ 1.45063602+0.j -0.02071393+0.j]
Right eigenvectors =  [[ 0.63122694 -0.56710364]
 [ 0.77559819  0.82364644]]
Eigenvalues =  [ 1.45063602+0.j -0.02071393+0.j]
Left eigenvectors =
 [[ 0.82364644 -0.77559819]
 [ 0.56710364  0.63122694]]
Right eigenvectors =
 [[ 0.63122694 -0.56710364]
 [ 0.77559819  0.82364644]]
index ||(A - eig * I) Vr || / ||Vr||  ||(A - eig * I) Vl|| / ||Vl||
 0        2.2204604604925031308e-16        2.4825341532472731 2e-16
 1        1.1102230246251565 4e-16        2.2204604925031308e-16
```

問題 6.2

eig.py を改良し，A を n 次複素正方行列として与える ceig.py を作成し，左右固有値，固有ベクトルを導出し，検算として定義式 (6.3), (6.4) を満足しているか，ノルム相対誤差を求めて確認できるようにして下さい。

6.3 定積分

例えば

$$\int_2^3 x \exp(\sin(x^2))dx \tag{6.6}$$

という定積分を計算するためには，SciPy の integrate モジュールを使って次のように計算します。

ソースコード 6.3 integration.py

```
# integration.py: 数値積分による定積分 +問題 12.4
import numpy as np
import scipy.integrate as scint # 積分パッケージ

# 被積分関数
def func1(x):
    return x * np.exp(np.sin(x ** 2))
# 定積分
a, b = 2, 3
ret = scint.quad(func1, a, b)

print('integral[', a, ',␣', b, ']␣:␣', ret[0])
print('ret␣=␣', ret)
print('aE(ans)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣', ret[1])
print('rE(ans)␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣', np.abs(ret[1] / ret[0]))
```

integration.py の実行結果

```
integral[ 2 ,  3 ] :  3.4199262740710528
ret =  (3.4199262740710528, 7.773062132546322e-12)
aE(ans)               =  7.773062132546322e-12
rE(ans)               =  2.2728741819610433e-12
```

問題 6.3

上記のスクリプトを使用して下記の定積分の値を求めよ。

1. $\int_0^{\pi} \exp(\sin x)\ dx$
2. $\int_0^3 \cos x^2 dx$

6.4 常微分方程式

ここで使用する常微分方程式は

$$\frac{d^r \mathbf{y}}{dx^r} = \widetilde{\phi}\left(x, \mathbf{y}, \frac{d\mathbf{y}}{dx}, \frac{d^2\mathbf{y}}{dx^2}, ..., \frac{d^{r-1}\mathbf{y}}{dx^{r-1}}\right) \tag{6.7}$$

という形式で表現できるもののみ扱うことにします。解はベクトル関数 $\mathbf{y}(x) = [y_1(x)\ ...\ y_n(x)]^T$ となります。

特に基本となるのは一階の一次元常微分方程式

$$\frac{dy}{dx} = f(x, y) \tag{6.8}$$

で，2 階以上の常微分方程式については

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_r \end{bmatrix} = \begin{bmatrix} y \\ dy/dx \\ \vdots \\ d^{r-1}y/dx^{r-1} \end{bmatrix}$$

と置き換えることによって，r 階常微分方程式 (6.7) を 1 階の常微分方程式

$$\frac{d}{dx}\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{r-1} \\ y_r \end{bmatrix} = \begin{bmatrix} y_2 \\ y_3 \\ \vdots \\ y_r \\ \widetilde{\phi}(x, y_1, y_2, ..., y_{r-1}) \end{bmatrix} \tag{6.9}$$

と同一視できます。したがって，以降は n 次元 1 階常微分方程式

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x, \mathbf{y}) \tag{6.10}$$

のみ考えることにします。

常微分方程式の解は無数に存在することは，最も簡単な次の例題で分かります。

例題 6.1

1 次元の常微分方程式

$$\frac{dy}{dx} = y$$

の解は右辺を左辺に移項して両辺を x について積分することによって得られる。この時，積分定数 $c \in \mathbb{R}$ が残り，結局，解 $y(x)$ は

$$y(x) = c\exp(x)$$

となる。すなわち，この解は無数に存在する。

解を一意に定めるためには，常微分方程式に対して条件を設定する必要があります。そのうち，変数 x が $x = x_0$ と固定された地点での $\mathbf{y}(x_0) = \mathbf{y}_0$ を与えられた問題を，「初期値問題」(initial value problem, IVP) と呼び，この $\mathbf{y}_0$ を初期値もしくは初期条件 (initial condition) と呼びます。したがって，常微分方程式の初期値問題は

$$\left\{ \begin{array}{rcl} \frac{d\mathbf{y}}{dx} & = & \mathbf{f}(x, \mathbf{y}) \\ \mathbf{y}(x_0) & = & \mathbf{y}_0 \end{array} \right. \tag{6.11}$$

という形で与えられます。これを Python で扱うには，SciPy の integrate パッケージを使用します。

■Python スクリプト例　常微分方程式のソルバーは SciPy.integrate パッケージにあります。初期値問題の場合は `solve_ivp` 関数をソルバーとして使うのが標準で，独立変数 x は `t` としてソルバーでは使用されます。

以下のスクリプトでは，ソルバーに積分区間内の評価点を決めさせる方法（デフォルト）と，ユーザが必要となる評価点を指定する方法で，それぞれ数値解を求め，そのグラフを描いています。後者のようにしておくと，積分区間におけるグラフを滑らかに描くことができるようになります。

ソースコード 6.4　`ode_ivp.py`

```
1 # ode_ivp.py: 常微分方程式の初期値問題
2 import numpy as np
3 import scipy.integrate as scint # ODE ソルバー
4 import matplotlib.pyplot as plt # グラフ描画
5
6
7 # 陽的形式の右辺
8 # y' = func(t, y) = y
```

```
 9 def func(t, y):
10   return y
11
12
13 # 初期値
14 # y(0) = 1
15 y0 = [1.0]
16
17 # t = [0, 1]
18 t_interval = [0.0, 1.0]
19 print(t_interval)
20
21 # 常微分方程式を解く
22 ret = scint.solve_ivp(func, t_interval, y0) # 評価点t が可変になる
23 ret_fix = scint.solve_ivp(
24   func, t_interval, y0,
25   t_eval=np.linspace(t_interval[0], t_interval[1], 10)
26 ) # 評価点t が固定化される
27
28 # 結果を表示
29 print(ret)
30
31 # y を表示
32 print(ret.y)
```

ode_ivp.py の実行結果

```
[0.0, 1.0]
message: The solver successfully reached the end of the integration interval.
success: True
 status: 0
      t: [ 0.000e+00  1.000e-01  1.000e+00]
      y: [[ 1.000e+00  1.105e+00  2.718e+00]]
 sol: None
  t_events: None
  y_events: None
   nfev: 14
   njev: 0
 nlu: 0
 [[1.         1.10519301 2.718327  ]]
```

問題 6.4

上記のスクリプトを使用して下記の常微分方程式の初期値問題の解を指定区間において求めよ。

1. $y' = -y, y(0) = 1, t \in [0, 1]$
2. $y' = -xy, y(0) = 1, t \in [0, 1]$