J. Plasma Fusion Res. Vol.94, No.3 (2018) 141‐149

講座 今日からはじめる Python

3．Python による科学技術計算

3. Scientific Computing in Python

3.1　NumPy/SciPy によるデータ解析

3.1 NumPy/SciPy for Data Analysis

釼 持 尚 輝
KENMOCHI Naoki
東京大学大学院　新領域創成科学研究科

（原稿受付：2018年１月23日）

NumPyはFortranのような多次元配列と科学技術計算をサポートするライブラリです．これにより配列全体への演算が可能となります．本章では，実際にプラズマ実験のデータ解析を行いながら，配列の読み書き，作成，配列演算について解説します．さらに，高度な科学技術計算に関する様々な機能を提供するパッケージであるSciPyについても，使い方を紹介します．例として，簡単なシミュレーションもできるSciPyパッケージのodeintモジュールを用いた微分方程式の数値解法例を示します．

Keywords:
python, numpy, scipy, simulation

3.1.1　はじめに

本章までに，Python で簡単な解析やプログラムの構築はできるようになりました．次にみなさんが行いたいのは，実験データの解析やシミュレーションでしょう．そこで，Python における科学技術計算の基礎ライブラリである NumPy の使い方を，実際の実験データを解析しながら紹介します．更に，高度な数学的アルゴリズムを提供する SciPy に関する簡単な紹介とともに，シミュレーションの例としてPredator-Preyモデルの簡単な解析例を示します．本章で興味を持たれた方は，NumPy/SciPy の公式 HP [1,2]や，近年充実してきている日本語書籍[3,4]を参考にして，より理解を深めていただければと思います．

3.1.1.1　NumPy/SciPy とは

NumPy

NumPy は"Numerical Python"の略語で，科学技術計算やデータ分析のための基本的なパッケージです．Python は一般に，C や Fortran 等のコンパイラー型言語と比較して性能が犠牲になっています．そこで NumPy では多次元配列 ndarray を C 言語で実装することで，使い勝手の良いインターフェイスを提供しつつも高速な演算を実現しています．特に，配列のデータをシステムのメモリ（RAM）に隙間なく配置することで，以下のような理由により高速化することができています．

・データをCPUレジスタにまとめて効率的に読み出せる
・スライス，転置などの操作を実際にデータをコピーせずに実現できる
・CPU のベクトル化演算の恩恵を受けられる

SciPy

SciPy はNumPyを利用するパッケージであり，高速フーリエ変換，最適化，数値積分，信号処理などの科学技術計算に関する機能を提供しています．SciPy では，Fortran のプログラムで実装された多くの関数群を提供しており，Python のスクリプト言語としての機能を大幅に強化しています．このおかげで，Python が科学技術計算において MATLAB，IDL，Octave，及び Scilab に匹敵するシステムになっています．

SciPy は，以下の表に示すような多くのサブパッケージ群から構成されています．

cluster	ベクトル量子化／K 平均法
constants	物理／数学定数
fftpack	FFT の関数
integrate	積分と常微分方程式ソルバー
interpolate	内挿とスムージングスプライン
io	データ入出力
linalg	線形代数ルーチン
ndimage	N 次元画像パッケージ
odr	直交距離回帰（Orthogonal Distance Regression）
optimize	最適化及び解探索ルーチン
signal	信号処理
sparse	疎行列と関連する関数
spatial	空間データ構造とアルゴリズム

Graduate School of Frontier Sciences, The University of Tokyo, Kashiwa, CHIBA 277-8561, Japan
author's e-mail: kenmochi@ppl.k.u-tokyo.ac.jp

©2018 The Japan Society of Plasma Science and Nuclear Fusion Research

special	任意の数学特殊関数
stats	統計分布，統計関数
weave	C/C＋＋統合

これらの機能を全て紹介することはできませんが，SciPyの公式ドキュメント[2]を閲覧するか，Pythonのインタラクティブシェルでヘルプを表示させることで詳細を調べることができます*1.

NumPyとSciPyの関係

SciPyのDocstringの冒頭には，「SciPyはNumPyの名前空間から全ての関数をimportし，加えて以下のサブパッケージを提供する」ということが書かれています．つまり，SciPyをimportすると，基本的に全てのNumPy関数を使えるようになります．ただし，SciPyの関数はNumPyの同一関数よりも最適化されていたり，機能が拡張されている場合が多いため，両者に関数が存在する場合には，SciPyの関数を用いるほうが計算速度の面で有利なことが多いです．

3.1.1.2 NumPy/SciPyの利用

それでは早速，NumPy/SciPyを使っていきましょう．第１章を参考にanacondaを使ってPythonをインストールした人は，既にNumPyは入っていると思います．そこで，まずはNumPy/SciPyの有無を確認してみます．

Pythonコンソールで

```
In [1]: import numpy

In [2]: import scipy
```

と打ってみてエラーがなければ無事にインストールされています．`No Module Named numpy`のようなエラーが出る場合は，ターミナルで以下のコマンドを入れてインストールしてください*2.

```
$ conda install numpy

$ conda install scipy
```

NumPyのインストールが完了したら，プログラム中で使用するためにimportを行います．外部パッケージの使用に関する詳細は第2章を参照してください．NumPyをimportするには，プログラム冒頭で以下のように宣言します．

```
In [3]: import numpy

In [4]: from numpy import *
```

`from モジュール名 import *`というコードは，既にスコープに存在する変数を知らない間に上書きしてしまう恐れがあります．そのため，本章ではNumPyの呼び出しは

```
In [5]: import numpy as np

In [6]: import scipy as sp
```

に統一してあります．読者の皆さんにも`np.`**関数名**での呼び出し記法を強く推奨します．

3.1.2 NumPy/SciPyを用いた実験データ解析

NumPy/SciPyを使う準備ができましたので，実際にプラズマ実験で得られたデータに対して解析をしてみましょう．ここでは，東京大学が所有する磁気圏型プラズマ装置RT-1[5]において得られた2視線のマイクロ波干渉計のデータを例にします．今回解析対象とする実験では，変化が分かりやすいように時刻t＝2.0 secに5 msec間のガスパフ入射を行っています．

なお，今回の記事で紹介する計測データにはhttps://github.com/PlasmaLib/python_tutorial/tree/master/dataからアクセスできます．ぜひ自身のPCにダウンロードして，実際に手を動かして操作感を感じていただければと思います．

3.1.2.1 実験データの読み込み

まずは実験データを読み込んでNumPyの配列を生成します．NumPyではファイル形式にバイナリとテキストを選びファイルの読み書きを行うことができますが，ここでは`np.loadtxt`を使用してテキスト形式で保存されている実験データを読み込んでみます．

```
In [1]: IF = np.loadtxt("data/IF_20170608_74_raw.txt", delimiter=',')
```

NumPyにおけるテキスト形式での読み書きには，以下の特徴があります．

- 他のアプリケーションと互換性のある.dat，.csv，.txt形式のファイルの読み書きができる
- 保存できる配列の次元は2次元まで

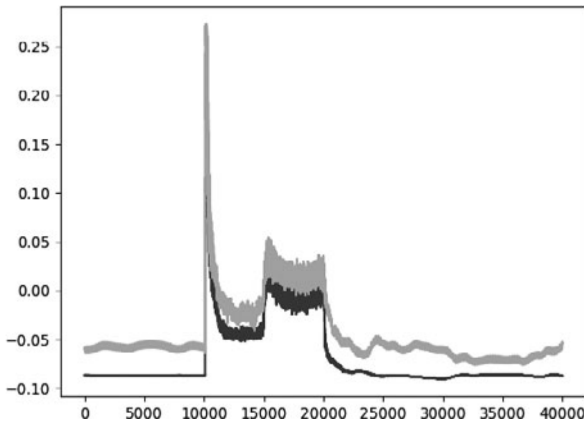
なお，バイナリ形式の読み書きには，`np.load`，`np.save`，`np.savez`，`np.savez_compressed`を使います．これらの関数は3次元以上のndarray配列も効率的にそのまま保存できますが，扱うファイル形式（.pickle，.npz，.npy）に他のアプリケーションとの互換性が殆どないことに注意が必要です．バイナリ形式での読み書きに関する詳細は，公式HP[6]を参照してください．

読み込んだデータの確認のため，Pythonで広く用いられるグラフ描写ライブラリであるMatplotlibを使って，グラフに表示してみます．Matplotlibの詳細は次章に譲るとして，ここでは以下のようにMatplotlibを読み込んでおきます．

```
In [2]: import matplotlib.pyplot as plt


In [3]: plt.plot(IF)
Out[3]:
[<matplotlib.lines.Line2D at 0x7f67d7503dd8>,
 <matplotlib.lines.Line2D at 0x7f67d74ff4a8>]
```

＊1 たとえばscipy.linalgのヘルプを表示させたい場合は，IPythonなどで`scipy.linalg?`と入力すればヘルプを参照することができます．

＊2 `$ pip install numpy`や`$ pip install scipy`でもインストールはできますが，condaを使うとIntel製の高性能行列ライブラリMKLが使えるようになるため，自動的に全てのコアを使って計算してくれるようになります．



3.1.2.2 配列の生成

次に，上図の時間軸を表示するための配列変数を作成します．時間軸のような等差数列の生成には，`np.arange` や `np.linespace` を使用します．なお，RT-1 のマイクロ波干渉計では，時刻 $t=0.5$ sec から $t=4.5$ sec まで，サンプル周波数 10 kHz でデータ収集を行っています．

```
In [4]: sampling_time = 1.0e-4

In [5]: delay = 0.5

In [6]: time = npo.arange(len(IF)) * sampling_time
+ delay
```

ここで，時間軸などの生成によく利用する `np.arange` と `np.linspace` の使い方を簡単に紹介します．

numpy.arange

`np.arange`は，連番や等差数列を生成します．使い方はPythonの組み込み関数rangeと似ており，以下のように引数を取ります．なお，[]で囲んだ引数は省略できるということを意味します．

```
arange([start,] stop, [step,] dtype=None)
```

`start` で指定した数から `stop` で指定した数まで，`step` 間隔の数字列を生成します．第 2 引数 `stop` 以外は省略ができますが，第 3 引数 `step` を指定するときは同時に第 1 引数 `start` も設定する必要があります．なお，第 2 引数 `stop` だけを指定した場合は，初項 0 で交差 1 の等差数列を要素とする ndarray を生成します．

numpy.linspace

`np.linspace` は等差数列を生成する関数です．同様の関数として先程紹介した `np.arange` がありますが，`np.linspace` を使用すると指定した区間をＮ等分した配列を生成しているということが明確になります．

```
linspace(start, stop, num=50, endpoint=True,
retstep=False, dtype=None)
```

の形で使用し，生成する等差数列の始点と終点を `start` と `stop` で指定します．第 3 引数 `num` で配列の長さを，第 4 引数`endpoint`で終点を配列の要素として含むかどうかを指定します．

3.1.2.3 配列の演算

データを読み込んで配列が生成できたところで，計測信号の較正値を適用して干渉計の位相信号を密度の値に変換し，そこからオフセットを差し引きます．

NumPy では ndarray で表現した行列に対して，行列の和・積，逆行列の計算，行列式の計算，固有値計算などさまざまな計算を行うメソッドや関数が用意されています．ここで，行列計算では ndarray の +（和），−（差），＊（積），/（除算），＊＊（べき乗），//（打ち切り除算），%（剰余）は要素同士の計算になるという点に注意が必要です．行列積を計算するには，dot メソッドを使うか，@演算子（Python3.5 以上かつ NumPy1.10 以上）を使う必要があります．

今回の例では，まず較正係数を適用して信号値を位相差の値に変換します．

```
In [7]: a1 = -0.005

In [8]: a2 = 0.000

In [9]: b1 = 0.135

In [10]: b2 = 0.300

In [11]: IF[:, 0] = np.arcsin((IF[:, 0]-a1)/b1)*
180/np.pi

In [12]: IF[:, 1] = np.arcsin((IF[:, 1]-a2)/b2)*
180/np.pi
```

次に，位相差を線積分密度の値に変換します．

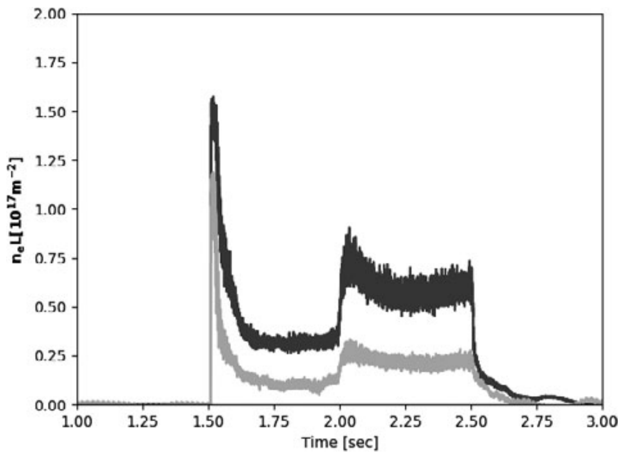
```
In [13]: IF = IF*5.58/360
```

最後に，プラズマのない時間帯の値をオフセットとして差し引きます．

```
In [14]: IF -= np.mean(IF[:5000], axis=0)
```

始めに作成した時間軸の配列とともにグラフに表示してみます．

```
In [15]: plt.plot(time, IF[:, 0]);

In [16]: plt.plot(time, IF[:, 1]);

In [17]: plt.xlim(1.0, 3.0);

In [18]: plt.ylim(0.0, 2.0);

In [19]: plt.xlabel('Time [sec]');

In [20]: plt.ylabel('$mathbf{n_eL [10^{17}m^{-2}]}$')
Out[20]: <matplotlib.text.Text at 0x7f67da09d5c0>
```



上記で用いた`IF[:5000]`は，プラズマがない時間帯（5000番目まで）のデータを切り出しています．このような処理を**インデキシング（Indexing）**と呼びます．`[]`の中身の`:5000`で配列IFの第0軸（この場合は時間方向に相当）の先頭から5000番目までの部分を示しています．

切り出した配列に対し`np.mean`では，`axis`でどの軸（axis）に沿って平均を求めていくのかを決めています．今回は各視線ごとの平均値を求めることが目的のため，`axis=0`として行方向，つまり列ごとの平均である1次元の2要素（視線1，視線2のデータ）のベクトルを求めています．

`IF -= np.mean(IF[:5000], axis=0)`は，元のデータから上記で求めた平均を差し引く操作です．2次元データである`IF`と，`np.mean`によって求めた1次元配列との引き算は，大きさが異なるため計算できないように思えます．その後の処理の，較正係数の引き算，除算も同様です．実はNumPyでは，**ブロードキャスティング（Broadcasting）**と呼ばれる仕組みにより，大きさを揃える操作を自動的に行っています．

インデキシング

上の例のようにNumPyでは，インデキシングという処理により，配列の任意の要素・行・列を切り出すことができます．ただし，切り出し方によりコピーを生成するかビュー（参照）を生成するかという違いがありますので注意が必要です．本講座の第2章で紹介したように，Pythonのリストやタプルにも実装されているスライシング（Slicing）をndarrayに対して行うと，その部分配列がビューとして返ってきます．つまり，その部分配列はデータのコピーではなく，元の配列の一部を参照していることになります．そのため，部分配列に対する変更はオリジナルのndarrayを変更してしまいます．

試しに，1列目の干渉計のプラズマ着火前の信号を抜き出してみます．

```
In [21]: IF_slice = IF[:5000, 0]
```

IF_sliceの中身を0に変更してみます．

```
In [22]: IF_slice[:] = 0

In [23]: IF[:5000, 0]
Out[23]: array([ 0.,  0.,  0., ...,  0.,  0.,  0.])
```

この例では，配列IF_sliceはビューですので，元の配列IFに変更が反映されています．

他の配列指向の言語ではスライスのようなデータ片はコピーとして生成する仕様のものが多いため，このインデキシングの仕様に驚く方は多いと思います．NumPyは，大量のデータ処理を目的として開発されてきました．ビューを用いると元のデータのコピーがメモリ上に作成されないため，特に大きな配列の操作に適しています*3

ブロードキャスティング

＋－＊/等の四則演算や，ユニバーサル関数を使ってndarray同士の演算を行う際に，異なるサイズの2つのndarrayを使って計算を行わなければならないことがあります．こういった処理を簡単・効率的に行うため，NumPyでは配列演算の拡張ルールであるブロードキャスティングを採用しています．以下にブロードキャスティングの一例として，1次元配列と2次元配列の配列演算を紹介します（**図1**）．

```
#1から12までの等差数列を作成し，形状を(4, 3)に変更する
In [24]: b = np.arange(1, 13, 1).reshape((4, 3))

In [25]: b
Out[25]:
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

In [26]: c = np.array([1, 2, 3])

In [27]: c.shape   # cの形状(shape)を確認する
Out[27]: (3,)

In [28]: b + c
Out[28]:
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12],
       [11, 13, 15]])
```

NumPyには，配列の全要素に対して要素ごとに演算処理を行う，ユニバーサル関数が組み込まれています．ユニバーサル関数はCやFortranで実装されており，かつ線形演算ではBLAS/LAPACKのおかげでC/C++と遜色のないほど高速に動作します．例えば，`exp`関数に配列を渡すことで，全要素に指数関数を適用した配列を生成することができます．

```
In [29]: np.exp(c)
Out[29]: array([  2.71828183,   7.3890561 ,
```

＊3　スライスをndarrayの実コピーとして生成する場合には，明示的に`arr2d[1, 1:].copy()`のようにします．

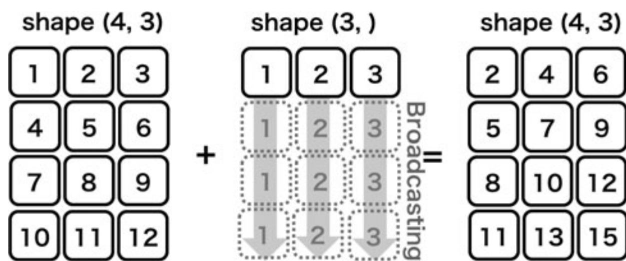


図1　ブロードキャスティングによる配列演算．

```
  20.08553692])
```

このように，NumPy では複数の配列要素に対して処理を一度に実行できます．こうすることで，ループ構造を用いるより圧倒的に高速に計算することができます．

Python のコードで良いパフォーマンスを得るには，以下の事が重要です．

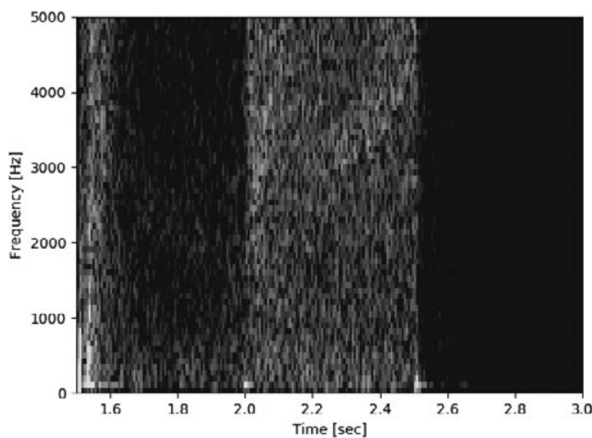
- Python のループと条件分岐のロジックを，配列操作と真偽値の配列の操作に変換する
- 可能なときは必ずブロードキャストする
- 配列のビュー（スライシング）を用いてデータのコピーを防ぐ
- ユニバーサル関数を活用する

特に，Pyhon の言語仕様に慣れないうちは for ループを多用しがちですが，これらに気をつけると Python でも高速で動作するプログラムを作ることができます．

3.1.2.4　SciPy を用いたデータ解析

時系列データの配列を作成することができたので，解析を行っていきましょう．今回の例では，SciPy の信号処理に関するサブモジュール scipy.signal の中の関数 spectrogram を用いて，上記のデータに短時間フーリエ変換を施し，プラズマの不安定性の有無を調べてみます．

```
In [30]: import scipy.signal as sig

In [31]: f, t, Pxx = sig.spectrogram(IF, axis=0, fs=1/sampling_time, window='hamming', nperseg=128,
noverlap=64, mode='complex')

In [32]: plt.pcolormesh(t+0.5, f, np.log(np.abs(Pxx[:, 0]) + 1e-15));

In [33]: plt.xlim(1.5, 3.0);

In [34]: plt.xlabel('Time [sec]');

In [35]: plt.ylabel('Frequency [Hz]');

In [36]: plt.clim(-9, -6)
```



ここで，`sig.spectrogram` には，元のデータ `IF` のほか，どの次元に対してフーリエ変換を施すかを `axis` オプションで，サンプリング周波数や，窓関数を `fs`，`window` オプションで指定して渡しています．$t = 2.2$ sec，周波数 3～4 kHz あたりに何か構造があるような気もします．

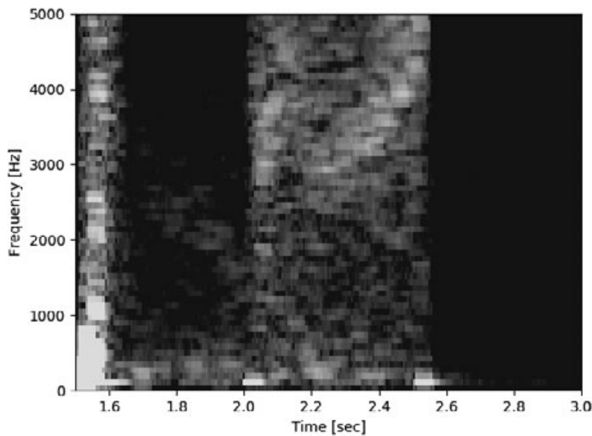
もう少しノイズを除去するために，２つの干渉計信号のクロススペクトルを計算してみましょう．クロススペクトルは，以下の式で計算される量です．

$$\langle f_1 f_2^* \rangle$$

$\langle x \rangle$ は x に関するサンプル平均を表します．ここでは移動平均で代用することにしましょう．

```
In [37]: def moving_average(x, N):
   ....:     x = np.pad(x, ((0, 0), (N, 0)), mode='constant')
   ....:     cumsum = np.cumsum(x, axis=1)
   ....:     return (cumsum[:, N:] - cumsum[:, :-N]) / N
   ....:
```

```
# クロススペクトルを求める
In [38]: Pxx_run = moving_average(Pxx[:, 0] * np.conj(Pxx[:, 1]), 8)

In [39]: plt.pcolormesh(t+0.5, f, np.log(np.abs(Pxx_run)));

In [40]: plt.xlim(1.5, 3.0);

In [41]: plt.clim(-19, -15);

In [42]: plt.xlabel('Time [sec]');

In [43]: plt.ylabel('Frequency [Hz]')
Out[43]: <matplotlib.text.Text at 0x7f67cdf55dd8>
```



ここで，np.conj(x)は複素共役を求めるユニバーサル関数で，配列の要素ごとに適用されます．移動平均を取る関数moving_averageの説明は省略しますが，スライシングと累積和を用いることで効率よく計算しています．

上記操作により，3〜4 kHz付近の構造を可視化することができました．このように，NumPy/SciPyの既存のツールを用いることで，スペクトル解析を簡単・高速に行うことができます．Matplotlibで描画することで，その結果をすぐに可視化しながら高速に解析を進めることができるでしょう．

3.1.2.5 解析データの書き込み

最後に，物理量に変換した配列を時間軸と一緒にテキスト形式で保存します．

```
In [44]: np.savetxt('time_IF.txt', np.c_[time,
IF], delimiter=',')
```

ここでは，配列の結合にnp.c_というオブジェクトを使用しています．np.c_はaxis=1の方向（2次元の場合は列方向）に，np.r_はaxis=0方向（2次元の場合は行方向）に配列を結合します．どちらも関数ではなくオブジェクトなので，全て[]の中に配列や値を入れて操作していきます．np.c_やnp.r_について更に詳しく知りたい場合は，docstring等を参照してください*4．なお他にも，np.concatenate，np.hstack，np.vstackなどの関数を用いても配列の結合を行うことができます．

3.1.3 SciPyを用いたPredator-Preyモデルのシミュレーション

本章の最後に，SciPyを用いた微分方程式の解法例として，Predator-Preyモデルのシミュレーションについて紹介します．

帯状流と乱流の相互作用は，捕食者－被食者（Predator-Prey）モデルで記述されることが知られており[7]，このモデルは1階の連立微分方程式の形をしています．SciPyパッケージのodeintモジュールを使うと，1階の常微分方程式の数値解を簡単に得ることができます*5．odeintはLSODA（Livermore Solver for Ordinary Differential equations with Automatic switching for stiff and non-stiff problems）法を利用した汎用的な積分器ですが，詳しくはODEPACK Fortran library[8]を参照してください．

まずは，ソースコードを見てみましょう．

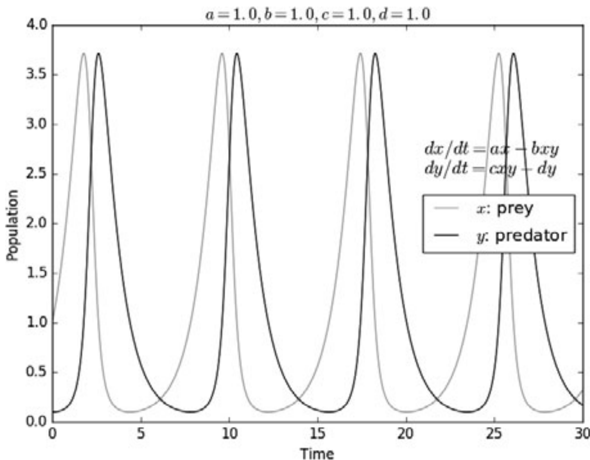
```
1 #!/usr/bin/env python
2 ¥PYGZdq¥PYGZdq¥PYGZdq
3     Sample Code
```

＊4 IPythonなどでnp.r_?と呼び出してdocstringを確認することができます．

＊5 なお，高階の微分方程式でも，1階の微分方程式に変換することでodeintを用いて計算することができます．

```
4
5      Status
6      ------
7      Version 1.0
8
9      Authour
10     -------
11     Shigeru Inagaki
12     Research Institute for Applied Mechanics
13     inagaki@riam.kyushu-u.ac.jp
14
15     Revision History
16     ----------------
17     [11-April-2017] Creation
18
19     Copyright
20     ---------
21     2017 Shigeru Inagaki (inagaki@riam.kyushu-u.ac.jp)
22     Released under the MIT, BSD, and GPL Licenses.
23
24 """
25 import numpy as np
26 import scipy.integrate as desol
27 import matplotlib.pyplot as plt
28
29 def predator_prey(f, t, a, b, c, d):
30     """ return left hand sides of ordinary differential equations
31
32         model equation:
33             dx/dt = ax - bxy
34             dy/dt = cxy - dy
35
36     f[0]    - x: Population of prey
37     f[1]    - y: Population of predator
38     t       - Time
39     a,b,c,d - Control parameters
40     """
41     return [a*f[0]-b*f[0]*f[1], c*f[0]*f[1]-d*f[1]]
42
43
44 #model
45 #eq1 = r"$fracdxdt = ax - bxy$"
46 #eq2 = r"$fracdydt = cxy - dy$"
47 eq1 = r"$dx/dt = ax - bxy$"
48 eq2 = r"$dy/dt = cxy - dy$"
49
50 #input parameters
51 a = 1.0
52 b = 1.0
53 c = 1.0
54 d = 1.0
55 header = r"$a=0:.1f, b=1:.1f, c=2:.1f, d=3:.1f$".format(a, b, c, d)
56
57 #initial condition
58 f0 = [1.0, 0.1]
59
60 #independent variable
61 nt = 1000
62 tmax = 30.0
63 dt = tmax / nt
64 t = dt * np.arange(nt)
65
66 f = desol.odeint(predator_prey, f0, t, args=(a,b,c,d))
67
68 #plot
69 prey = f[:,0]
70 predator = f[:,1]
71
72 fig = plt.figure()
73 ax = fig.add_axes([0.15, 0.1, 0.8, 0.8])
```

```
74 ax.plot(t, prey, color='r', label=r"$x$: prey")
75 ax.plot(t, predator, color='b', label=r"$y$: predator")
76 handles, labels = ax.get_legend_handles_labels()
77 ax.legend(handles, labels, loc='best')
78 ax.text(21, 2.7, eq1, fontsize=16)
79 ax.text(21, 2.5, eq2, fontsize=16)
80 ax.set_xlabel("Time")
81 ax.set_ylabel("Population")
82 ax.set_title(header)
83 plt.show()
```



プログラムの内容は以下のようになっています．

1．解析する関数（この場合 predator_prey）を定義する
　・第１引数 f が微分方程式中の未知関数
　・第２引数 t が関数のパラメータ（時間に対応）
　・第３‐６引数 a，b，c，d が定数
　・戻り値がパラメータ t における dx/dt，dy/dt を与える
2．微分方程式の定数 a，b，c，d を与える
3．微分方程式の初期値 f0 を与える
4．未知関数の解析範囲（時間）を与えるパラメータ列 t を用意する
5．関数 `scipy.integrate.odeint` に1-4を引数にして呼び出す
6．戻り値がパラメータtに対応する未知関数fの各値となる

帯状流とプラズマ乱流の相互作用を当てはめて考えてみると，乱流を餌として発生・成長する帯状流は捕食者の役割を，またプラズマ圧力勾配により発生する線形不安定性を源として成長する乱流は被食者の役割を果たします．

このようにPython を用いることで，簡単にモデルの計算と可視化をすることができます．コーディングの時間を短縮し，試行錯誤に多くの時間を割けるのが Python の利点でもありますので，みなさんもまずは簡単なプログラムを作成し，動作を確認してみてください．

3.1.4　まとめ

本章では，NumPy/SciPyの特徴と基本的な使用法，簡単なシュミレーションの例を紹介しました．NumPy が他の多くのライブラリの基礎となっているため，NumPy の基本を理解することが Python を用いた科学技術計算にとって重要です．本章で紹介したNumPyにおけるndarrayやユニバーサル関数，ブロードキャスティングの概念は Python の機能を大幅に拡張しており，これらの概念に慣れることがプログラミングの効率を大きく向上させます．さらに，SciPyを用いることでNumPyの機能の上に構築された様々な科学技術計算アルゴリズムを利用できます．SciPy は非常に巨大なパッケージですので，効率良く計算を進めるため，処理を実装する前に SciPy で既に実装されていないかどうかぜひ確認してみてください．

科学技術計算において，特に解析対象や解析方法がその都度変化するプラズマ実験では，実験条件に対応して柔軟にコードを組まなければなりません．例えば，数分間の実験周期中に直前のプラズマ放電で得られたデータを解析し，それを基に次の放電の条件を決めるといった場合，非常に短時間にコードを組んで解析を進めることが要求されます．こういった状況では，実行速度よりも開発速度が重要になることが多く，Python はその用途に適しています．

本講座で Python の使い方を一通り覚えたら，まずは自身の研究でも試してみてください．すぐに Python の柔軟性や開発のスピード感を味わってもらえると思います．

参 考 文 献

［1］http://www.numpy.org
［2］https://www.scipy.org
［3］Wes McKinney：Python によるデータ分析入門（オライリー・ジャパン，2013）．
［4］中久喜健司：科学技術計算のための Python 入門（技術評論社，2016）．

［5］Z.Yoshida *et al.*, Phys. Plasmas, **17**, 112507 (2010).

［6］https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.io.html

［7］小林すみれ　他：プラズマ・核融合学会誌 **92**, 211 (2016).

［8］http://people.sc.fsu.edu/~jburkardt/f77_src/odepack/odepack.html

釼持尚輝（けんもち なおき）

1987年静岡県浜松市生まれ．東京大学大学院新領域創成科学研究科　助教．2011年京都大学工学部物理工学科卒業．2016年同大学院エネルギー科学研究科博士後期課程修了．博士（エネルギー科学）．学生時代は京都大学 Heliotron J で，現在は東京大学 RT-1装置にて熱・粒子輸送を研究．両装置でトムソン散乱計測装置開発に従事．趣味は空手（京都大学空手道部コーチ）．最近，生命保険に加入したが，空手が危険スポーツとみなされ保険料が大幅に上がってしまった．