Python 3 覚書

桂田 祐史

2016年2月23日, 2024年1月8日

1 はじめる

1.1 Python 3 を始める理由

Python を始める理由については、Python 覚書 1.1 節「Python を始める理由」[1] に書いておいた。

久しぶりにまた少しいじってみようかな、という気になった。そろそろ 3 を試してみようか、と考えた。Mac OS に付属の Python は相変わらず 2.7 系列だけど、1 台のマシンに 2 つインストールするのならば、バージョンが違う方がかえって混乱がないかもしれない、というのと、あまりいつまでも古いのと付き合っていると、自分自身が遅れてしまうかなあ、と。

1.2 Mac での利用

MacPorts を使うことにした。

```
sudo port install py35-numpy +openblas
sudo port install py35-scipy +openblas
sudo port install py35-matplotlib
```

LLVM のコンパイルなどを始めて、結構な大事になるけれど、特に問題なく終了する。

python3.5 を python, python3 で起動するようにするには、

```
sudo port select --set python python35
sudo port select --set python3 python35
```

のようにするとか。

どういう選択肢があるかは、`port select --list python` で分かる。python は python 2.x 用にしておくべきかも (Python 3 は、python というコマンド名ではインストールしないのがデフォールトだそうだ)。

元々 Mac OS X では、`python` で Python 2 が起動されるようになっているので、後者 (`python3` で Python 3.5 を起動するように設定) だけを採用した。

同様に cython-3.5 を cython で起動するようにするには

```
port select --set cython cython35
```

とするのだとか。実は、これを書いている瞬間、Cython とは何か知らないのだが…

[1] `http://nalab.mind.meiji.ac.jp/~mk/labo/text/python/node2.html`

1.3 情報の入手

ドキュメント中のチュートリアルは出来が良いと思う。まあ，それ以外にもインターネット上に山のように情報がある。

- Python 3.5.1 documentation[2]
- Python 3.5.1 ドキュメント[3] (2016 年 2 月現在 Stable な 3.5.1 の日本語ドキュメント)
  チュートリアル[4]
- Style Guide[5] (こんなふうに書け、というスタイル・ガイド) ???
- Numpy for Matlab users[6]

1.4 個人用のメモ

- インデントがブロックを決めている？タブは使わず、空白 4 個が基本。
- 文末にセミコロンは要らない。
- 複素数あるけれど、`j` または `J` というのがなあ…
- 同時の代入 `a,b=0,1` というのがある。これを使うとフィボナッチ数列の計算は `a,b = b,a+b` で OK.

2 最初にいくつかのプログラムを試してみる

2.1 いわゆる Hello

prog1.py

```
print("Hello, python.")
```

```
% python3 prog1.py
Hello, python.
%
```

あるいは先頭に

```
#!/opt/local/bin/python3
```

のように書いて、

```
chmod +x prog1.py
```

として、スクリプトとして起動する、というのもあり。

[2] http://docs.python.org/3.5/
[3] http://docs.python.jp/3.5/
[4] http://docs.python.jp/3.5/tutorial/index.html
[5] http://www.python.org/dev/peps/pep-0008/
[6] https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

```
#!/usr/bin/env python3
```

とする，と書いてあるものが多い。どういうことだろう？と検索したら何か凄いことになっている。#!/opt/local/bin/python3 のように Python の具体的なパスを書く方が良いのじゃないかなあ，という人に凄い剣幕でツッコミを入れている人もいる。くわばら，くわばら。

文字列についていくつか。

- 引用符は、" でも ’ でも良い。
- " で括った文字列の中では ’ は普通に使える。’ で括った文字列の中では " は普通に使える。
- 改行 \n, タブ \t, バックスラッシュ \\, シングルクォート \’, クォーテーション・マーク \"
- 文字列は + で連結できる。リテラルの場合は ’A’ ’BC’ のように単に並べるだけで連結される。
- 文字列は * で繰り返しできる。print("ばんざーい！"*3)
- 数値は str(数値を表す式) で文字列にできる。書式を指定したければ .format() を使う。

```
>> n=10
>> x=3.141592
>> print('n={0:}, x={1:.2f}'.format(n,x))
n=10, x=3.14
```

2.2 簡単な日本語表示

Python 2 の場合は、「簡単な日本語表示」[7] のようにすれば良かった。

Python 3 では、何も指定しなければ、UTF-8 が使われるので、特に意識せずに済む、らしい。
(2021/8/24 追記) 久しぶりに指定なしでやったら叱られてしまった (仕様が固定されていないのは何だかなあ)。Python 3 でも指定が必要らしい。最初の行か次の行に # coding: utf-8 のようなのを書く (utf_8 と utf-8 のどっちという話があるけれど、ここに書くのは、utf-8 の方が良いらしい)。私は emacs つかいなので、おまけをつけて -*- coding: utf-8 -*- としている。(まあ私は emacs の方でも設定しているので、デフォールトで UTF-8 で編集することになるけれど。人に渡すことを考えて？だったら相手が emacs を使っていることも仮定できないよね。)。
(2021/11/18 追記) あれ、指定なしで動くぞ (/opt/local/bin の方でも ~/opt/anaconda3/bin の方でも)。8/24 のは何かの勘違いかな。

prog2.py

```
#!/usr/bin/env python3
print("こんにちは、パイソン。")
```

[7] http://nalab.mind.meiji.ac.jp/~mk/labo/text/python/node10.html

```
% python3 prog2.py
こんにちは、パイソン。
%
```

対話モードの時にうまく動かず、どうすれば良いのか困っている。— と以前書いたけれど、今やってみるとスイスイ動く。

2.3 __name__ の利用, コマンド・ライン引数

prog3.py

```
#!/usr/bin/env python3
print("これは常に実行される")

def test():
    print("関数：test を呼び出しました")

if __name__ == "__main__":
    print("ここは単独のスクリプトとして起動された場合のみ実行する")
    test()
```

```
% ./prog3.py
これは常に実行される
ここは単独のスクリプトとして起動された場合のみ実行する
関数：test を呼び出しました
%
```

コマンド・ライン引数は sys.argv で得られる。C 言語の argc はないが len(sys.argv) で分かる。C 言語では atoi(), atof() を良く使うが、Python では int(), float() を使えば良い。

```
if __name__ == "__main__":
    argc = len(sys.argv)
    if argc == 1:
        testrungekutta()
    elif argc == 2:
        testrungekutta(float(sys.argv[1]))
    elif argc == 3:
        testrungekutta(float(sys.argv[1]), int(sys.argv[2]))
```

この機能は便利だが、Jupyter notebook で実行した場合も、__name__ == "__main__" は真になり、sys.argv[1] は '-f', sys.argv[2] は ' なんとか.json' になるので、それに対応するようにするには一工夫が必要そう。

あまり自信のない書き直し

```
if __name__ == "__main__":
    argc = len(sys.argv)
    if argc == 1:
        testrungekutta()
    elif argc == 2:
        testrungekutta(float(sys.argv[1]))
    elif (argc == 3 and sys.argv[1] != '-f'):
        testrungekutta(float(sys.argv[1]), int(sys.argv[2]))
    else:
        print('usage: testrungekutta(Tmax=50.0,N=1000)')
```

2.4 とにかく for

(実際には、なるべく for を避けるのが高速化の秘訣のようなところがあるけれど) 何と言っても繰り返しが大事、for の使い方を知ろう。

```
for i in range(10):
    print(i)
```

これは

```
for i in range(0,10):
    print i
```

や

```
for i in range(0,10,1):
    print i
```

や

```
for i in [0,1,2,3,4,5,6,7,8,9]:
    print i
```

と同じ。

Python 2 では

```
for i in xrange(10):
    print i
```

の方が速いということだったが、Python 3 では xrange() が存在しないようだ。

3 簡単な入出力

3.1 標準出力への出力

C 言語だと printf() だが、Python だと print() を使うのだろう。

C 言語

```
printf("Hello, world.\n");
printf("n=%d, error=%f\n", n, e);
```

Python

```
print('Hello, world.')
print('n=', n, ', error=', e)
```

書式を指定した出力をどうするか。

C 言語

```
printf("n=%4d, error=%12.5f\n", n, e);
```

色々なやり方が用意されている。

Python

```
n=1234; e=12345.6789

print('n=%4d, error=%12.5f' % (n, e))

print(f'n={n:4d}, error={e:12.5f}')

print('n={0:4d}, error={1:12.5f}'.format(n,e))
```

私の感想としては、迷走していると思う (Java の「書式の指定」の "The March of Progress"[8])。のようなことにならないことを祈る。

3.2 標準入力からの入力 (C 言語の scanf() の真似)

input() という関数があり、C 言語のプログラムの翻訳をすると使いたくなることが多いが、使っている実例をあまり見たことがない。

C 言語

```
printf("n,x=");
scanf("%d%f", &n, &x);
```

Python

```
print('n,x=')
str=input()
s=str.split()
n=int(s[0])
x=float(s[1])
```

split() の代わりに strip() が良い場合があるかもしれない。
str=input('n,x=') とすると、"n,x=" と改行なしに表示出来る。
一度に 1 つの入力しかしないならば、split() とかしないで

[8] http://nalab.mind.meiji.ac.jp/~mk/labo/java/intro-java/node11.html

```
str=input('N='); N=int(str);
```

のような感じの単純なコードで済む。

3.3 ファイル入出力

(準備中)

例えば、C 言語の次のようなプログラムと同じようなことをするにはどうしたら良いか。

```
#include <stdio.h>
#include <math.h>

int main(void)
{
  int N, i;
  double dx, x;
  FILE *f;
  char line[512];
  f=fopen("test.txt", "w");
  N=90;
  dx=4*atan(1.0)/2/N;
  for (i = 0; i <= N; i++) {
    x = i * dx;
    printf("%d %f %f\n", i, sin(x), cos(x));
  }
  fclose(f);

  f=fopen("test.txt", "r");
  while (fgets(line,sizeof(line),f) != NULL) {
    double s, c;
    sscanf(line, "%d%lf%lf", &i, &s, &c);
    printf("%d %f %f\n", i, s, c);
  }
}
```

要するに、0 度から 90 度までの 1 度刻みで、sin, cos の値を出力したファイルを作り、それを読み込んで表示する、という内容である。行単位でお仕事をする、というクラシックなプログラムである。

さすがに scanf() 系統の関数は Python には用意されていないようだ。空白などで区切られているならば、split() を使えばバラせるだろう。

```
# sincostable.py
import math
with open('test.txt', mode='w') as f:
    N=90
    dx=math.pi/2/N
    for i in range(N+1):
        x=i*dx
        print('%d %f %f' % (i,math.sin(x),math.cos(x)), file=f)

with open('test.txt', mode='r') as f:
    line=f.readline()
    while line != '':
        s=line.split()
        print('%d %f %f' % (int(s[0]), float(s[1]), float(s[2])))
        line=f.readline()
```

4 数学関数

import math とすれば math.sin() のように、C 言語の数学関数にあるものが、math. 名前 () で使えるようになる。

そのうち書くけれど、とりあえず「math — 数学関数」[9] とか見て下さい。

5 数値計算

5.1 はじめに

Python は基本インタープリターであるので、工夫をしないと数値計算の効率が上がらない。まずは、Numpy, Scipy 等のパッケージを使うことを考えること。

どんなふうに使えるか、簡単な入門は、「Numpy と Scipy」[10]

正式なドキュメントは、Numpy and Scipy Documentation[11]

5.2 Numerical Python (NumPy)

せっかちな MATLAB 使いには，NumPy for Matlab Users[12] という WWW ページが分りやすいかも。でも…

NumPy v1.13 Manual[13]

NumPy Reference[14]

[9] https://docs.python.org/ja/3/library/math.html
[10] https://www.eidos.ic.i.u-tokyo.ac.jp/~tau/lecture/computational_physics/slide/numpy.pdf
[11] https://docs.scipy.org/doc/
[12] http://www.scipy.org/NumPy_for_Matlab_Users
[13] https://docs.scipy.org/doc/numpy/index.html
[14] https://docs.scipy.org/doc/numpy/reference/routines.html

5.2.1 使うために最初にするおまじない

利用するには，最初に

```
import numpy

(これで例えば numpy.array() のように使える。)
```

とするか

```
from numpy import *

(これで例えば直接 array() のように使える。逆に numpy.array() では使えない。)
```

と宣言するか，あるいは例えば

```
import numpy as np

(これで例えば np.array() のように使える。)
```

のように別名 (大抵は短縮名) を与えて宣言する。

最後のやり方が普通なのだろうか？

Numpy には自己チェック機能がある。テストするには

```
>>> import numpy as np
>>> np.test('full')
```

(…とあるのだけど、どうなったら OK なのか、良く分からない。)

5.2.2 array

array クラス (ndarray と言うのか？ N dimensional array から来ているそうだ) はいわゆる配列であるらしい。

array() は array 型のデータ (array クラスのインスタンスというのか？) を作れる。

array() の引数にリストを指定すると，そのリストの成分を持つ array が出来る。

```
>>> from numpy import *
>>> array([1,2,3])
array([1, 2, 3])
>>> array(range(1,4))
array([1, 2, 3])
>>> array([[1,2,3],[2,3,4]])
array([[1, 2, 3],
       [2, 3, 4]])
```

あるいは

```
>>> import numpy as np
>>> np.array([1,2,3])
>>> np.array(range(1,4))
>>> np.array([[1,2,3],[2,3,4]])
```

のようにも使える。以下は `from numpy import *` とした場合で説明する。

(i,j) 成分は `a[`i`,`j`]` でアクセス出来る。C 言語の配列のように 0 から始まることに注意する。

```
>>> a=array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> a[1,1]=5
>>> a
array([[1, 2],
       [3, 5]])
```

単なる代入 `b=a` は,「参照による代入」であり，別名がつくだけでコピーされるわけでなく，ソースを変更するとデスティネーションも変更される。`b` を `a` のコピーにするには，明示的に `b=a.copy()` とする。

```
>>> a=array([[1,2],[3,4]])
>>> b=a
>>> a[0,0]=10
>>> a
array([[10,  2],
       [ 3,  4]])
>>> b
array([[10,  2],
       [ 3,  4]])
```

`b=a` とした場合，`a` をいじったら，`b` も変わってしまいました。

```
>>> b=a.copy()
>>> a[0,0]=1
>>> a
array([[1, 2],
       [3, 4]])
>>> b
array([[10,  2],
       [ 3,  4]])
```

`b=a.copy()` とした場合，`a` をいじっても，`b` は変わらない。

array というクラスを用意したのは，もちろん実行効率上の理由も大きいだろう。

Mathematia のリストは，行列やベクトルを表すのに使えるが，Python のリストでは全然無理 (スカラー倍すら出来ない)。Python の array は，加法やスカラー倍は自然に出来る。

しかし array の掛け算 `*` は成分毎の積になる。内積や行列としてのを計算するには `dot()`

を用いるか (この辺は Mathematica のドット演算子 . を想起させる)，後で紹介する matrix にする必要がある。array クラスは，次元が 1,2 より大きいものも使える，つまりベクトル・行列向けに特殊化せずに一般的である，ということだ。この辺はなるほどと思う。

論よりラン

```
>>> 2*[1,2,3]
[1, 2, 3, 1, 2, 3]
>>> 2*array([1,2,3])
array([2, 4, 6])
>>> array([1,2,3])+array([2,4,6])
array([3, 5, 7])
>>> array([1,2,3])*array([2,4,6])
array([2, 8, 18])
>>> dot(array([1,2,3]),array([2,4,6]))
28
```

reshape() でサイズを変更できる。

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

5.2.3 zeros(), ones(), empty(), identity()

zeros() は成分がすべて 0 の array を作れる。

```
>>> zeros(2)
array([ 0.,  0.])
>>> zeros((3,2))
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
>>> help(zeros)
```

ones() は成分がすべて 1 の array を作れる。

```
>>> ones(3)
array([ 1.,  1.,  1.])
>>> ones((3,2))
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

empty() は初期化しない array を作れる。

```
>>> empty(3)
array([0.299, 0.587, 0.114])
>>> empty((3,2))
array([[0.50798611, 0.33611111],
       [0.50798611, 3.35611111],
       [5.15798611, 0.33611111]])
```

(結果はもちろん不定である。)

identity(n) は n 次単位行列の成分を持つ 2 次元 array を作れる。

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

5.2.4 乱数 — numpy.random パッケージ

(MATLAB の zeros(), ones(), eye() と来たか，次は rand() だ，と思うところ…)
numpy.random パッケージの関数 random() を用いる。

```
>>> random.random()
0.9587940341101487
>>> random.random(3)
array([ 0.4923399 ,  0.88603936,  0.56831053])
>>> random.random((3,2))
array([[ 0.15967363,  0.51198445],
       [ 0.60262639,  0.33262596],
       [ 0.25892059,  0.4649679 ]])
```

細かいことだけど，random.random(), random.random(1), random.random((1,1)) は，いずれも 1 つの乱数を返すわけだけど，型がみな違う。安易な同一視はしないわけね。

5.2.5 線形演算 — numpy.linalg パッケージ

以下，行列の行列式 (linalg.det())，逆行列 (linalg.inv())，固有値・固有ベクトルの計算 (linalg.eig() 等) をする例

```
>>> a=array([[1,2],[3,4]])
>>> a
array([[1, 2],
       [3, 4]])
>>> linalg.det(a)
-2.0000000000000004
>>> linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
>>> (lam,v)=linalg.eig(a)
>>> lam
array([-0.37228132,  5.37228132])
>>> v
array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]])
>>> dot(linalg.inv(v),dot(a,v))
array([[ -3.72281323e-01,   8.88178420e-16],
       [ -5.55111512e-17,   5.37228132e+00]])
```

他にもノルム (linalg.norm())，冪乗 (linalg.matrix_power())，エルミート行列の固有値・固有ベクトル (linalg.eigh())，QR 分解 (linalg.qr())，特異値分解 (linalg.svd())，Cholesky 分解 (linalg.cholesky()) などがある。

```
>>> help(linalg.lapack_lite)
```

5.2.6 matrix クラス

matrix クラスは特殊な 2 次元 array である．掛け算演算子などが行列としての積として作用する (逆に成分毎の掛け算をするには，multiply(a,b) とする — 変なの)。

(念のため: linalg パッケージの関数は，matrix に対しても array と同じように使える。結果は array でなく matrix になる。)

関数 mat() (matrix()) の引数に 2 次元 array またはそれを「表す」文字列を与えることで作れる。

```
>>> a=mat('1,2;3,4')
>>> a
matrix([[1, 2],
        [3, 4]])
>>> a=mat([[1,2],[3,4]])
>>> a
matrix([[1, 2],
        [3, 4]])
```

メンバー関数として，逆行列を計算する getI(), Hermite 共役を計算する getH(), 転置を計算する getT() がある。それぞれ a.I, a.H, a.T で呼び出せる。

```
>>> a.I
matrix([[-2. ,  1. ],
        [ 1.5, -0.5]])
>>> a*a.I
matrix([[  1.00000000e+00,   0.00000000e+00],
        [  8.88178420e-16,   1.00000000e+00]])
>>> a.H
matrix([[1, 3],
        [2, 4]])
>>> a.T
matrix([[1, 3],
        [2, 4]])
```

5.2.7 これはどうやる？

- a\b は

```
linalg.solve(a,b)
```

5.3 Scientific Tools for Python (SciPy)

"Sigh Pie" (サイパイ) と読むのだそうだ。
ScientificPython というのがあるが，それとは関係ない。
SciPy Reference[15]

5.3.1 インストール

MacPorts でない場合どうするかの経験談 (ただし Python 2) は、「インストール」[16] にある。
MacPorts の場合は，NumPy のインストールと同様に，次のような 1 行コマンドで OK.

```
sudo port install py35-scipy +openblas

(35 は対応する Python のバージョンが 3.5 であることを意味する。)
```

MacPorts は簡単でよろしい。

5.3.2 使うために最初にするおまじない

利用の仕方は Numpy と同様で，最初に

```
from scipy import *
```

と宣言するか，あるいは例えば

[15] https://docs.scipy.org/doc/scipy/reference/
[16] http://nalab.mind.meiji.ac.jp/~mk/labo/text/python/node24.html

```
import scipy
```

のように宣言して以下 scipy. 何某 () で呼び出すか,

```
import scipy as sci
```

のように宣言して以下 sci. 何某 () で呼び出す。

5.3.3 実例: LU 分解を用いて $Ax = b$ を解く

(ある二日間の試行錯誤の記録。最初に MATLAB のやり方を踏襲しようとして，イマイチな結果となって，後でよりまともそうなやり方を見つけた，というストーリー。お急ぎの方は最後の例だけ見て下さい。)

MATLAB では $Ax = b$ はこうやって解く

```
n=10000;
a=rand(n,n);
[L,U,P]=lu(a);
x=ones(n,1);
b=a*x;
x2=U\(L\(P*b));
norm(x-x2)
```

あるいは

MATLAB では $Ax = b$ はこうやって解く (置換をベクトルで扱う版)

```
n=10000;
a=rand(n,n);
[L,U,p]=lu(a,'vector');
x=ones(n,1);
b=a*x;
x2=U\(L\(b(p)));
norm(x-x2)
```

これと同等なことを Scipy でやってみよう、ということ。

```
>>> from numpy import *
>>> import scipy as sci
>>> import scipy.linalg
```

Numpy の関数はダイレクトに名前 (mat() とか linalg.solve() とか) で使える。scipy の関数は sci. を先頭につけて (sci.linalg.lu() とか) 使える。scipy の多くのサブパッケージは，一々インポートしないと使えないものが多い。ここでも scipy.linalg をインポートする。

```
>>> a=mat([[1,2],[3,4]])
>>> help(scipy.linalg.lu)
  help(sci.linalg.lu) でも OK
>>> P,L,U=sci.linalg.lu(a)
```

(この P, L, U は array である。)

```
>>> P=mat(P)
>>> L=mat(L)
>>> U=mat(U)
>>> P
matrix([[ 0.,  1.],
        [ 1.,  0.]])
>>> L
matrix([[ 1.        ,  0.        ],
        [ 0.33333333,  1.        ]])
>>> U
matrix([[ 3.        ,  4.        ],
        [ 0.        ,  0.66666667]])
>>> P*L*U
matrix([[ 1.,  2.],
        [ 3.,  4.]])
>>> x=mat(ones((2,1)))
>>> b=a*x
>>> linalg.solve(U,linalg.solve(L,P.T*b))
matrix([[ 1.],
        [ 1.]])
```

scipy.linalg.lu() は Scipy 用に書き下ろされたものだそうだ (LAPACK とかではなくて)。これは出来がイマイチみたい (P, L, U を使って連立 1 次方程式を解くとき、あまり速く解けない)。

sci.linalg.lu_factor(), sci.linalg.lu_solve() というのもある。

```
>>> import scipy as sci
>>> import scipy.linalg
>>> a=sci.mat([[1,2],[3,4]])
>>> lu,piv=sci.linalg.lu_factor(a)
>>> lu
array([[ 3.        ,  4.        ],
       [ 0.33333333,  0.66666667]])
>>> piv
array([1, 1], dtype=int32)
```

(この lu, piv を見て「なるほど」という感じがする。)

```
>>> x=sci.mat([1,2]).T
>>> b=a*x
>>> x2=sci.linalg.lu_solve((lu,piv),b)
>>> x2
array([[ 1.],
       [ 2.]])
```

大きい問題を解いてみる。

```
import numpy as np
import scipy as sci
import scipy.linalg

n=10000
print(n,"次の乱数行列生成")
a=sci.mat(np.random.random((n,n)))
print("LU 分解")
lu,piv=sci.linalg.lu_factor(a)
x=np.ones((n,1))
print("掛け算")
b=a*x
print("方程式を解く")
x2=sci.linalg.lu_solve((lu,piv),b)
print("誤差=",sci.linalg.norm(x-x2))
```

こちらはマルチコアをちゃんと使って速い (time で測って 580% という数値が出た)。

6 matplotlib

ドキュメントはどこですか？

User Guide[17] を読んでいたけれど，The Matplotlib FAQ の Usage[18] を読んで初めて腑に落ちたことが多い。

[17] http://matplotlib.org/users/index.html
[18] http://matplotlib.org/faq/usage_faq.html

「【Matplotlib 日本語訳】 使用ガイド（Usage Guide）」[19]

- ipython
- pyplot
- pylab というのは matplotlib のモデュールで，numpy

6.1 pyplot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 10, 0.2)
y = np.sin(x)
plt.plot(x,y)
plt.show()
```

6.2 pylab

```
from pylab import *
x=arange(0, 10, 0.2)
y=sin(x)
plot(x,y)
show()
```

6.3 backend (よく分からない)

Matplotlib Examples[20] の例でエラーが生じた。

```
AttributeError: 'FigureCanvasMac' object has no attribute 'copy_from_bbox'
```

ネットで調べたら，backend を変更してみろ，例えば

```
import matplotlib
matplotlib.use('TkAgg')
```

を最初に (正確には，matplotlib.pyplot や matplotlib.pylab をインポートする前に) やっておけ，とあったので，そうしたら動いた。

matplotlibrc に

```
backend : TkAgg
interactive : True
```

と書いておくものなのかな？

[19] https://python-pyramid.com/?p=38\#i-20
[20] http://matplotlib.org/examples/index.html

現在，macosx というバックエンドは，非対話モードで blocking show() が出来ないとか何とか。

Using matplotlib in a python shell[21] に色々書いてあるので，そのうち解読しよう。

バックエンドについては、Matplotlib Usage[22] に色々書いてある (これはなかなか良さそうな説明)。

6.4 メモ 1

そのうちまとめるけれど、とりえず、これまでどんなのを使っているか、並べておく。

- 公式情報は「Matplotlib: Visualization with Python」[23], 特に「Matplotlib x.y.z documentation」[24] から。
- matplotlib のインストールの仕方は “色々”. コマンドラインから `pip install matplotlib` とか、`sudo port install py310-matplotlib` とか。Anaconda なら最初から入っている。
- 使うには事前に import する必要がある。例えば

```
import matplotlib.pyplot as plt
```

- x, y が同じ長さの ndarray であれば、`plot()` で折れ線が描ける。

```
plt.plot(x,y)
plot.show()
```

`scatter()` で散布図が描ける。

- `plot()` は、`,label=`文字列 でラベルをつけられる。

```
plt.plot(t,sol, label='x0='+'{:6.1f}'.format(x0))
```

ラベルの位置は制御できる。`legend()` に `,loc=` で指定する。

```
plt.legend(loc='upper left')
```

`loc='best'` とかもある。

- `plot()` の線の色は、1 文字 `'b'` (青), `'g'` (緑), `'r'` (赤), `'c'` (シアン), `'m'` (マゼンタ), `'y'` (黄), `'k'` (黒), `'w'` (白) くらいでもつけられる (「matplotlib で指定可能な色の名前と一覧」[25])。`'b--'` とすると点線になる。

```
plt.plot(t,x[:,0],'b', label='S')
```

[21] http://matplotlib.org/users/shell.html
[22] http://matplotlib.org/faq/usage_faq.html\#what-is-a-backend
[23] https://matplotlib.org/
[24] https://matplotlib.org/stable/index.html
[25] https://pythondatascience.plavox.info/matplotlib/色の名前

- ,marker='o' (丸), ,marker='^' (三角), ,marker='s' (四角)
- 図のタイトルは plt.title(文字列)

```
plt.title('Malthus: dx/dt=ax, x(0)=x0; a='+str(a))
```

- 軸のラベルは plt.xlabel(文字列), plt.ylabel(文字列)
- plt.grid()
- xlim(minx,maxx), ylim(miny,maxy) で範囲が指定できる。

```
plt.xlim(0.0, 1.0)
plt.ylim(-1.0, 2.0)
```

- アニメーション。理解できていないが

```
  plot.ion()
  line,=plt.plot(x,u)
...
  # u の内容を更新
  line.set_ydata(u)
  plt.pause(0.001)
```

アニメーションの例 (よく理解できていない) http://nalab.mind.meiji.ac.jp/~mk/labo/text/python3/node36.html

- 図を保存するには savefig(パス名) を用いる。

```
plt.savefig('myfig.png')
```

plt.show() した後に plt.savefig() するのは、うまく行かないことがある (普通 plt.show() すると、current figure (直訳すると「現在の図」) がリセットされるので、plt.show() する前に plt.savefig() すべき。Jupyter の中など、動く場合もあるが、それが普通と考えてはいけないらしい)。

savefig() は plt.show() の前にする

```
plt.savefig('なんとか.png')
plt.show()
```

Saving a figure after invoking pyplot.show() results in an empty file[26] がとても分かりやすい。

少し面倒だけれど、絵を描き出す前に fig, ax = plt.subplots() として、fig に figure を覚えさせておいて、plt.savefig() でなく fig.savefig() を実行するのが、一つの堅実なやり方。確かにこういうコードはよく目にする。次のような感じ (上記サイトに載っているコード例)。

[26] https://stackoverflow.com/questions/21875356/saving-a-figure-after-invoking-pyplot-show-results-in-

これがお勧めのやり方か

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1, 1, 100)
y = x**2

fig, ax = plt.subplots()
ax.plot(x, y)
fig.savefig('fig1.pdf')
plt.show()
fig.savefig('fig2.pdf')
```

別解として、show() する前に、current figure を fig=plt.gcf() で覚えておいて、fig.savefig() するという手もあるそうだ (gcf は "get current figure" , つまり「現在の図を得る」という意味)。

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-1, 1, 100)
y = x**2

plt.plot(x, y)
fig = plt.gcf()
fig.savefig('fig1.pdf')
plt.show()
fig.savefig('fig2.pdf')
```

(でも、ax を使いたい場合が多いので、この別解を採用する気はあまりない。)

- こんなのもあるか。figure()

```
  fig = plt.figure()
  ...
  fig.savefig('fig2-1.png')
```

- animation.FuncAnimcation() というのがあるけれど、自分でやった範囲では、Python 3 では特に高速化されないので、とりあえずスルーしておく (気にしないこと、のススメ)。
- subplot() については、「matplotlib のめっちゃまとめ」[27] が参考になった。自分のプログラム例としては http://nalab.mind.meiji.ac.jp/~mk/labo/text/intro-ode-simulation/node67.html の testsir6.py とか。
- Jupyter Notebook では、%matplotlib inline がないと表示できないことがある。
- 2 変数関数の可視化については実例のみ。「ウォーミング・アップで Poisson 方程式」[28]

[27] https://qiita.com/nkay/items/d1eb91e33b9d6469ef51
[28] http://nalab.mind.meiji.ac.jp/~mk/labo/text/python3/node50.html

6.5　1 変数関数の可視化

(準備中)

6.6　2 変数関数の可視化 (contour(), plot_surface(), quiver(), streamplot())

等高線　contour(), contourf() で等高線の描画が出来る (後者は塗りつぶしをする)。

test_contour.py

```
# test_contour.py

# 2 変数関数の等高線

import numpy as np
import matplotlib.pyplot as plt

# データの準備
xmin=-1; xmax=1; ymin=-1; ymax=1; nx=100; ny=100
xs=np.linspace(xmin,xmax,nx+1)
ys=np.linspace(ymin,ymax,ny+1)
x,y=np.meshgrid(xs,ys)
z=np.sqrt(x*x+y*y+0.1)

# 等高線描画
level=np.linspace(0.0,2.0,30+1) # リストでも可
fig,ax=plt.subplots()
ax.contour(x,y,z,levels=level,cmap='jet')
# plt.xlim([xmin,xmax]); plt.ylim([ymin,ymax])
ax.set_aspect('equal') # 縦/横 を指定

plt.show()
```

グラフの鳥瞰図

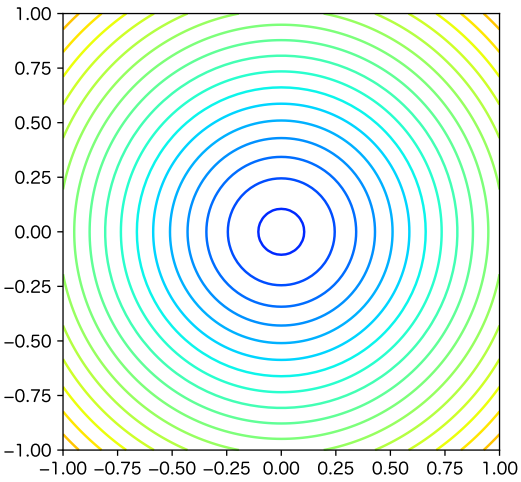


図 1: $f(x, y) = x^2 + y^2$ の等高線

test_graph.py

```
# 2変数関数のグラフの鳥瞰図
import numpy as np
import matplotlib.pyplot as plt

# データを準備
xmin=-1; xmax=1; ymin=-1; ymax=1; nx=100; ny=100
xs=np.linspace(xmin,xmax,nx+1)
ys=np.linspace(ymin,ymax,ny+1)
x,y=np.meshgrid(xs,ys)
z=np.sqrt(x*x+y*y+0.1)

# グラフを描画
fig=plt.figure(figsize=(6,6),facecolor='w')
ax = fig.add_subplot(111, projection='3d') # 111, はなくても良い
mysurface=ax.plot_surface(x,y,z,cmap='jet') # cmap='jet' とか color='b'

plt.show()
```

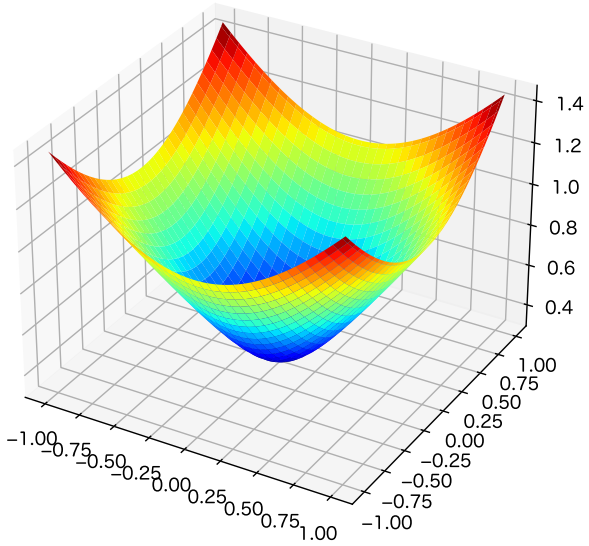
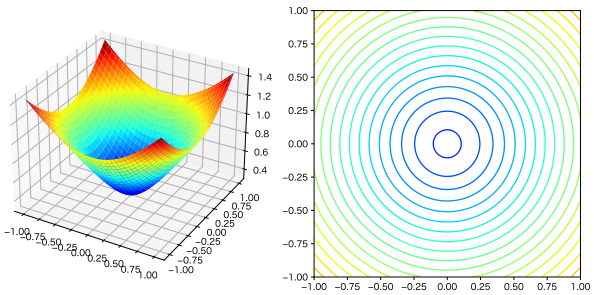


図 2: $f(x, y) = x^2 + y^2$ のグラフ

test_graph_contour.py

```
# 2変数関数のグラフと等高線

import numpy as np
import matplotlib.pyplot as plt

# データ準備
xmin=-1; xmax=1; ymin=-1; ymax=1; nx=100; ny=100
xs=np.linspace(xmin,xmax,nx+1)
ys=np.linspace(ymin,ymax,ny+1)
x,y=np.meshgrid(xs,ys)
z=np.sqrt(x*x+y*y+0.1)

# グラフ
fig=plt.figure(figsize=(12,6),facecolor='w')
ax1 = fig.add_subplot(121, projection="3d") # 1行2列の1番目 --- 左
mysurface=ax1.plot_surface(x,y,z,cmap='jet')

# 等高線
ax2 = fig.add_subplot(122) # 1行2列の2番目 --- 右
level=np.linspace(0.0,2.0,30+1)
ax2.contour(x,y,z,levels=level,cmap='jet') # plt.contour() でも表示できる
ax2.set_aspect('equal')
#ax2.set_xlim([xmin,xmax]) # plt.xlim() でも出来る
#ax2.set_ylim([ymin,ymax]) # plt.ylim() でも出来る

plt.show()
```



ベクトル場

quiver() を用いるとベクトル場を描くことが出来る (MATLAB がそうなんだな。quiver には「震える」という意味があって、それしか知らなかったので変な名前だと思ったけれど、「箙 (えびら) の矢; 矢筒」という意味があるそうな…えびらなんて知らない)。

test_quiver.py

```
# test_quiver.py

# 2 変数関数のグラフの鳥瞰図
import numpy as np
import matplotlib.pyplot as plt

R0=2.5

# データを準備
smin=0; smax=1; imin=0; imax=1; ns=15; ni=15
xs=np.linspace(smin,smax,ns+1)
ys=np.linspace(imin,imax,ni+1)
s,i=np.meshgrid(xs,ys)
v1=-R0*s*i
v2=R0*s*i-i

# グラフを描画
# plt.quiver() とも出来るけれど
fig,ax = plt.subplots()
ax.set_aspect('equal')
ax.quiver(s,i,v1,v2)
#ax.streamplot(s,i,v1,v2)
ax.grid()

plt.show()
```

quiverkey() で矢印の説明 (凡例というのかな) が出来る。

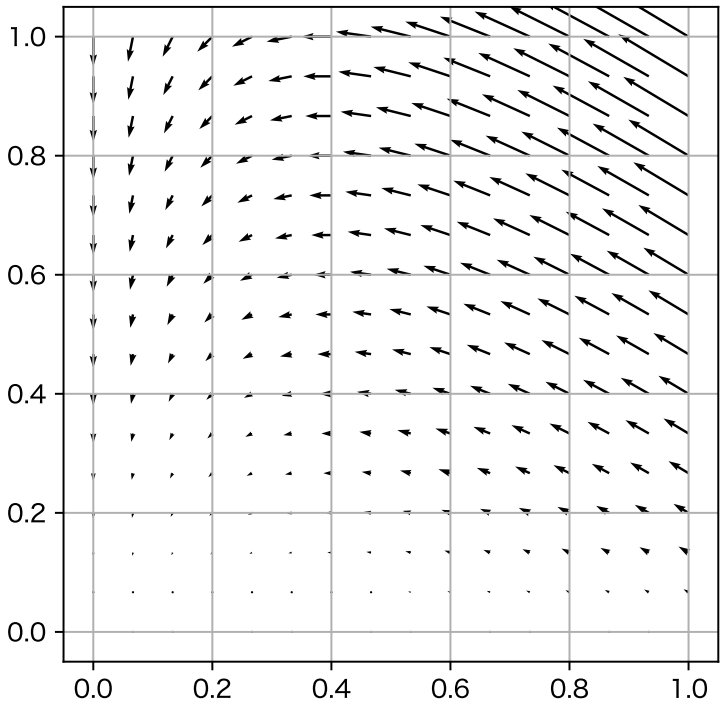


図 3: ベクトル場 $\boldsymbol{f}(S,I) = \begin{pmatrix} -R_0SI \\ R_0SI - I \end{pmatrix}$

```
q=ax.quiver(s,i,v1,v2)
ax.quiverkey(q, X=0.3, Y=1.05, U=1,
             label='Quiver key, length = 1', labelpos='E')
```

矢印はたくさん描くと見にくくなるので、上のプログラムでは ns, ni を 15 としたが、何か他の目的で幅の狭い格子点での値を計算する場合は、スライスを使って (s[::5,::5] のように) “飛ばして” 値を拾うと良い。

また矢印の長さを調節するには ,scale=比率 が使える。

```
ns = 100; ni = 100

(中略)

q=ax.quiver(s[::5,::5],i[::5,::5],v1[::5,::5],v2[::5,::5],scale=10)
```

ベクトル場の流線

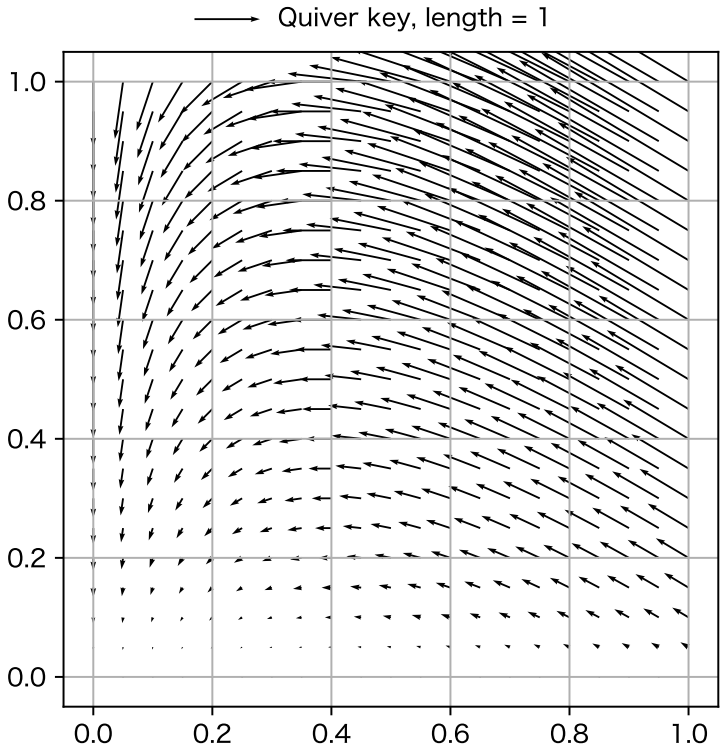


図 4: ベクトル場 $\boldsymbol{f}(S,I) = \begin{pmatrix} -R_0SI \\ R_0SI - I \end{pmatrix}$ (微小修正)

test_streamplot.py

```
# test_quiver.py

# 2変数関数のグラフの鳥瞰図
import numpy as np
import matplotlib.pyplot as plt

R0=2.5

# データを準備
smin=0; smax=1; imin=0; imax=1; ns=15; ni=15
xs=np.linspace(smin,smax,ns+1)
ys=np.linspace(imin,imax,ni+1)
s,i=np.meshgrid(xs,ys)
v1=-R0*s*i
v2=R0*s*i-i

# グラフを描画
# plt.quiver() とも出来るけれど
fig,ax = plt.subplots()
ax.set_aspect('equal')
#ax.quiver(s,i,v1,v2)
ax.streamplot(s,i,v1,v2)
ax.grid()

plt.show()
```

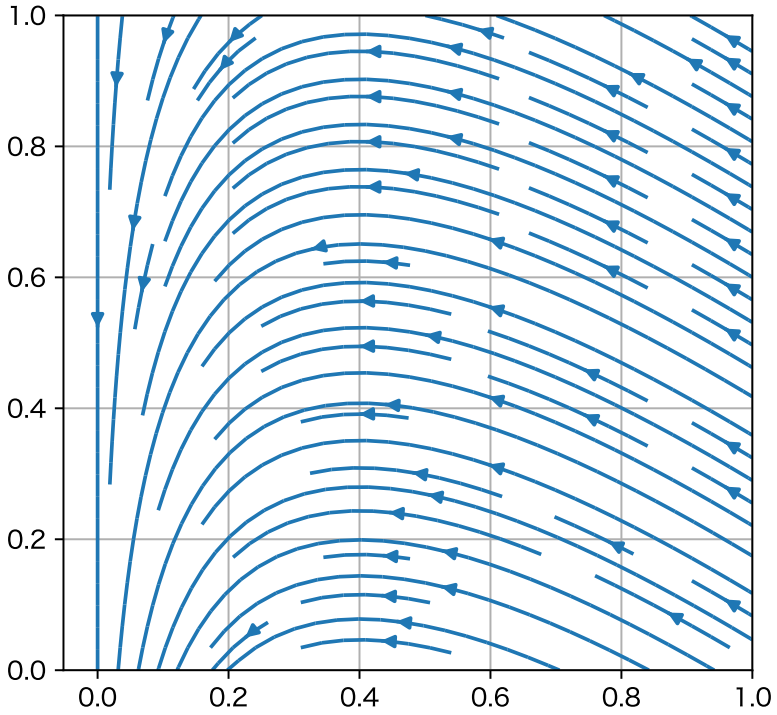


図 5: ベクトル場 $\boldsymbol{f}(S,I) = \begin{pmatrix} -R_0SI \\ R_0SI - I \end{pmatrix}$ の流線

7　1 次元熱方程式を試す

桂田研では毎度おなじみの，熱伝導方程式の初期値境界値問題

(1) $$u_t(x,t) = u_{xx}(x,t) \quad ((x,t) \in (0,1) \times (0,\infty)),$$

(2) $$u(0,t) = u(1,t) = 0 \quad (t \in (0,\infty)),$$

(3) $$u(x,0) = f(x) \quad (x \in [0,1])$$

を差分法で解け，というプログラム。f は与えられた関数で，以下のプログラムでは，次のように決め打ち。

$$f(x) := \min\{x, 1-x\} = \begin{cases} x & (x \in [0,1/2]) \\ 1-x & (x \in (1/2,1]). \end{cases}$$

数値計算環境の個人的な評価をするために便利だと思っている。差分方程式，連立 1 次方程式，グラフィックスをその環境でどう実現するか，自分の頭を働かせることになるので。

事前の考えでは

- Numpy を使えば差分方程式の扱いは問題ないだろう。
- 同様に連立 1 次方程式も多分大丈夫 (以下に見るように，興が乗って，C 言語で書いたモジュールを使ってみた)。
  もっとも空間の次元によってかなり違うかな？空間 1 次元では，とりあえず三項方程式 (tridiagonal system of linear equation) を解くことになる。

- グラフィックスはどうなるのか？
  (やり始めた時点で matplotlib ほとんど知らない)

叩き台はこれまで書いた C プログラム (「どこでも 1 次元熱方程式の差分法シミュレーション」[29])。今回作る Python プログラムもそのうちそちらに書き足すのだろう。

7.1 陽解法

最初の素朴なバージョン。全部計算して，描画していって，最後に一気に表示する。

heat1d-v0.py

```
#!/usr/bin/env python3
# heat1d-v0.py

import numpy as np
import matplotlib.pyplot as plt

def f(x):
    y=x.copy()
    for i in range(0,len(y)):
        if y[i] > 0.5:
            y[i] = 1-y[i]
    return y

N=50

x=np.linspace(0.0, 1.0, N+1)
u=f(x)
newu=np.zeros(N+1)

h=1.0/N
lam=0.5  # lambda は予約語で使えない？
tau=lam*h*h
dt=0.01
skip=int(dt/tau)

Tmax=1.0
nmax=int(Tmax/tau)

plt.plot(x,u)

for n in range(1,nmax):
    for i in range(1,N):
        newu[i]=(1-2*lam)*u[i]+lam*(u[i-1]+u[i+1])
    if n%skip == 0:
        plt.plot(x,newu)
    u=newu.copy()

plt.show()
```

少し改善したバージョン。初期値 f の計算に numpy の `vectorize()` を使うとか，内側の `for` を取って `newu[]` を省略するとか，計算しながら表示するとか。

[29] http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat1d-everywhere/

heat1d-e.py

```
#!/usr/bin/env python3
# heat1d-e.py --- 1 次元熱方程式
# http://d.hatena.ne.jp/Megumi221/20080306/1204770689 を参考にした

import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return min(x,1-x)

N=50
x=np.linspace(0.0, 1.0, N+1)
# vf=np.vectorize(f)
# u=vf(x)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5  # lambda は予約語で使えない？
tau=lam*h*h
Tmax=1.0
nmax=int(Tmax/tau)
dt=0.001
skip=int(dt/tau)

plt.ion()
line,=plt.plot(x,u)

for n in range(1,nmax):
    u[1:N]=(1-2*lam)*u[1:N]+lam*(u[0:N-1]+u[2:N+1])
    if n%skip == 0:
        line.set_ydata(u)
        plt.pause(0.00001) # 除くと動かない。必要な理由を理解できてない
```

7.2 陰解法

いわゆる θ 法 (「発展系の数値解析」[30]) によるプログラムで，C 言語によるプログラム (「公開プログラムのページ」[31] にある `heat1d-i-glsc.c` 等) があるので，陽解法のプログラムが出来ていれば後は比較的簡単。

三項方程式を解くために `trilu()` (LU 分解), `trisol()` (三項方程式の係数行列が LU 分解してあるとして，三項方程式を解く) という関数をどう実現するかが問題。

[30] http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat-fdm-0.pdf
[31] http://nalab.mind.meiji.ac.jp/~mk/program/

heat1d-i.py

```
#!/usr/bin/env python3
# heat1d-i.py
# http://d.hatena.ne.jp/Megumi221/20080306/1204770689 を参考にした

import numpy as np
import matplotlib.pyplot as plt

def trilu(n, al, ad, au):
    for i in range(0,n-1):
        al[i+1] = al[i+1] / ad[i]
        ad[i+1] = ad[i+1] - au[i] * al[i+1]

def trisol(n, al, ad, au, b):
    nm1 = n-1
    for i in range(0,nm1):
        b[i+1] = b[i+1] - b[i] * al[i+1]
    b[nm1] = b[nm1] / ad[nm1]
    for i in range(n-2,-1,-1):
        b[i] = (b[i] - au[i] * b[i+1]) / ad[i]

def f(x):
    return min(x,1-x)

N=50
x=np.linspace(0.0, 1.0, N+1)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5  # lambda は予約語で使えない？
tau=lam*h*h
theta=0.5
Tmax=1.0
nmax=int(Tmax/tau)
dt=0.001
skip=int(dt/tau)

plt.ion()
line,=plt.plot(x,u)

# 三重対角行列を用意して LU 分解
ad=(1+2*theta*lam)*np.ones(N-1)
al=-theta*lam*np.ones(N-1)
au=-theta*lam*np.ones(N-1)
trilu(N-1,al,ad,au)

for n in range(1,nmax):
    b=(1-2*(1-theta)*lam)*u[1:N]+(1-theta)*lam*(u[0:N-1]+u[2:N+1])
    trisol(N-1,al,ad,au,b)
    u[1:N]=b
    if n%skip == 0:
        line.set_ydata(u)
        plt.pause(0.001)
```

(ちなみに陽解法と陰解法で、ユーザー時間と経過時間、いずれも 22,3 秒で、ほとんど差がない。計算そのものよりもアニメーションの実現部分に時間が取られているらしい。)

7.3 trilu(), trisol() を C で書いて高速化 (?)

Python 2 を使っていたとき、「三次元日誌 numpy の配列を受け取る C モジュールを作る」[32] を参考にして C モジュールを作ってみた。C モジュールはどうやって作るかを知るのが目的だった。

Python 3 では、そのとき作成した C モジュールは使えない。ちょっとショック？まあ、その辺はリサーチ不足だった。

「Python インタプリタの拡張と埋め込み」[33] を読んでやり直し。無事に出来た。

7.3.1 モジュールの書き方

まずモジュールの名前を決める。

某という名前のモジュールのインプリメンテーションが入った C のソースファイルは、某`module.c` としよう。

(1) 最初に書くのは次の 1 行である。

```
#include <Python.h>
```

(2) Python で 某. なんとか (string) という呼び出しをして呼ばれるメソッド (C 言語のプログラムとしては「関数」) の実体を用意する。ここは自分がやりたいことを書くわけで、ケース・バイ・ケースである。

```
static PyObject *
某_なんとか(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command); // コマンドを実行する
    return PyLong_FromLong(sts);
}
```

(3) モジュールのメソッドテーブルを用意する。

```
static PyMethodDef 某Methods[] = {
    ...
    {"なんとか",  某_なんとか, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

[32] http://d.hatena.ne.jp/ousttrue/20091205/1260035679
[33] http://docs.python.jp/3.5/extending/index.html

(4) モジュール定義構造体を用意する。

```
static struct PyModuleDef 某module = {
   PyModuleDef_HEAD_INIT,
   "某",   /* name of module */
   某_doc, /* module documentation, may be NULL */
   -1,       /* size of per-interpreter state of the module,
                or -1 if the module keeps state in global variables. */
   某Methods
};
```

(5) 初期化関数を用意する。

```
PyMODINIT_FUNC
PyInit_某(void)
{
    return PyModule_Create(&某module);
}
```

7.3.2 モジュールを作ってみる

さて、では作ってみよう。

```
/*
 * tridmodule.c --- 三重対角行列の LU 分解をする Python 用 C モデュール
 */

/* 三重対角行列の LU 分解 (pivoting なし) */
void trilu(int n, double *al, double *ad, double *au)
{
  int i, nm1 = n - 1;
  /* 前進消去 (forward elimination) */
  for (i = 0; i < nm1; i++) {
    al[i + 1] /= ad[i];
    ad[i + 1] -= au[i] * al[i + 1];
  }
}

/* LU 分解済みの三重対角行列を係数に持つ 3 項方程式を解く */
void trisol(int n, double *al, double *ad, double *au, double *b)
{
  int i, nm1 = n - 1;
  /* 前進消去 (forward elimination) */
  for (i = 0; i < nm1; i++) b[i + 1] -= b[i] * al[i + 1];
  /* 後退代入 (backward substitution) */
  b[nm1] /= ad[nm1];
  for (i = n - 2; i >= 0; i--) b[i] = (b[i] - au[i] * b[i + 1]) / ad[i];
}

#include <Python.h>
#include <numpy/arrayobject.h>
#include <numpy/arrayscalars.h>
#include <stdlib.h>
```

```
static PyObject *trid_trilu(PyObject *self, PyObject *args)
{
  int n;
  PyArrayObject *al, *ad, *au;

  if (!PyArg_ParseTuple(args, "iOOO", &n, &al, &ad, &au))
    return NULL;

  if (al->nd != 1 || al->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg2 types does not much");
    return NULL;
  }
  if (ad->nd != 1 || ad->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg3 types does not much");
    return NULL;
  }
  if (au->nd != 1 || au->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg4 types does not much");
    return NULL;
  }
  trilu(n, (double*)al->data, (double*)ad->data, (double*)au->data);
  return Py_BuildValue(""); // return Py_RETURN_NONE; ち OK?
}

static PyObject *trid_trisol(PyObject *self, PyObject *args)
{
  int n;
  PyArrayObject *al, *ad, *au, *b;

  if (!PyArg_ParseTuple(args, "iOOOO", &n, &al, &ad, &au, &b))
    return NULL;

  if (al->nd != 1 || al->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg2 types does not much");
    return NULL;
  }
  if (ad->nd != 1 || ad->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg3 types does not much");
    return NULL;
  }
  if (au->nd != 1 || au->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg4 types does not much");
    return NULL;
  }
  if (b->nd != 1 || b->descr->type_num != PyArray_DOUBLE) {
    PyErr_SetString(PyExc_ValueError, "arg5 types does not much");
    return NULL;
  }

  trisol(n,
         (double*)al->data, (double*)ad->data, (double*)au->data,
         (double*)b->data);
  return Py_BuildValue("");
}

static PyMethodDef tridMethods[] = {
  {"trilu", trid_trilu, METH_VARARGS, "LU factorize a tridiagonal matrix"},
  {"trisol", trid_trisol, METH_VARARGS, "solve linear equation"},
  {NULL,NULL,0,NULL} /* Sentinel */
};

static struct PyModuleDef tridmodule = {
```

```
  PyModuleDef_HEAD_INIT,
  "trid",   /* name of module */
  NULL, /* module  documentation, may be NULL --- "trid_doc" みたいの */
  -1,        /* size of per-interpreter state of the module,
                or -1 if the module keeps state in global variables. */
  tridMethods
};

// この辺は Python 2 とは全然違う
PyMODINIT_FUNC PyInit_trid(void)
{
  return PyModule_Create(&tridmodule);
}
```

7.3.3 setup.py

「Numpy の配列を利用する C モジュールを作る」[34] から頂きました (中身はまったく理解していません)。

```
from distutils.core import setup, Extension
from numpy.distutils.misc_util import get_numpy_include_dirs

setup(
    package_dir={'':''},
    packages=[
    ],
    ext_modules=[
        Extension('trid',
            sources=[
            'tridmodule.c'
            ],
            include_dirs=[] + get_numpy_include_dirs(),
            library_dirs=[],
            libraries=[],
            extra_compile_args=[],
            extra_link_args=[]
        )
    ]
)
```

7.3.4 ビルド＆インストール

```
python3 setup.py build
sudo python3 setup.py install
```

7.3.5 heat1d-i-v2.py

```
#!/usr/bin/env python3
# heat1d-i-v2.py
# http://d.hatena.ne.jp/Megumi221/20080306/1204770689 を参考にした
# http://www.scipy.org/Cookbook/Matplotlib/Animations

import numpy as np
import matplotlib.pyplot as plt
import trid
```

[34] http://codeit.blog.fc2.com/blog-entry-9.html

```
def f(x):
    return min(x,1-x)

N=50
x=np.linspace(0.0, 1.0, N+1)
# vf=np.vectorize(f)
# u=vf(x)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5  # lambda は予約語で使えない？
tau=lam*h*h
theta=0.5
Tmax=1.0
nmax=int(Tmax/tau)
dt=0.001
skip=int(dt/tau)

plt.ion()
line,=plt.plot(x,u)

ad=(1+2*theta*lam)*np.ones(N-1)
al=-theta*lam*np.ones(N-1)
au=-theta*lam*np.ones(N-1)
trid.trilu(N-1,al,ad,au)

for n in range(1,nmax):
    b=(1-2*(1-theta)*lam)*u[1:N]+(1-theta)*lam*(u[0:N-1]+u[2:N+1])
    trid.trisol(N-1,al,ad,au,b)
    u[1:N]=b
    if n%skip == 0:
        line.set_ydata(u)
        plt.pause(0.0001)
```

もう少しケチれるかな。b を消して余計なコピーを無くせる。

```
u[1:N]=(1-2*(1-theta)*lam)*u[1:N]+(1-theta)*lam*(u[0:N-1]+u[2:N+1])
trid.trisol(N,al,ad,au,u[1:N])
```

7.4 animation.FuncAnimcation() で高速化？

animation.FuncAnimcation() を使うと速くなる ?? 実は全然分かっていない。

Python 2 のときは、これを使って確かに速くなったのだけど、Python 3 では、これを使わなくても十分速くて、このプログラムでは違いが見い出せない。でも、一応 Python 3 でも動くので、このまま載せておく。

7.4.1 実は良く分かっていないので分っている部分，分かっていない部分をメモ

- まず引数が良く分からない。
- 第 1 引数は，描画する figure オブジェクトというのは問題ない。
- 第 2 引数は画面更新のために定期的に呼ばれる関数を指定する。その名前を “update” にする人が多い (もちろん指定できるようにしてあるわけで，そうする必要はない)。
- interval= というのは，フレームを更新する時間間隔をミリ秒単位で指定する。

- update() は引数なしには出来ない。呼ばれるのが何回目かを表す整数 (frame number と呼んでいる人がいる) が入るのがデフォールト？そのときは def update(i): みたいな宣言となる。
- update() に整数でない引数が入るとき，その引数を生成する関数みたいのが用意できて，それを FuncAnimation() の引数に指定するようだ。
- yield って何だろう？
- init_func= というので初期化関数らしきものが指定できる。“is a function used to draw a clear frame” だそうだけど。名前を init にする人が多いのは当然か。init_func=init とするわけ。
- update() にしても init() にしても，何か返すのだけど (それにしばしば line という名前をつけるのだけど)，一体なんだろう。http://jakevdp.github.com/blog/2012/08/18/matplotlib-animation-tutorial/ には，“Note that again here we return a tuple of the plot objects which have been modified. This tells the animation framework what parts of the plot should be animated.” なんて書いてある。
- return line と return line, があるけど？何となく後者が正しそうな匂いがする。
- frames= とは？描画するフレームの枚数ということらしいけれど，repeat= とからむのかしら？
- repeat=True と repeat=False と何が違うんだろう？(変えてみて実行したりしているけれど，差が分からない。)
- blit=True とは？何か変更したところだけ描画するみたいな指定？“ this tells the animation to only re-draw the pieces of the plot which have changed”
- frames= にしても，init_func= にしても，必要なければ書かないことも出来るし，frames=None や init_func=None のように必要ないことを明示することも出来る。
- fargs= とは？
- blit= が良く分からない。blit という単語は辞書にも載っていないし，ちゃらんぽらん Documents は困る。

7.4.2 heat1d-v3.py

とりあえず FuncAnimation() を使って動いた最初のプログラム。

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# heat1d-v3.py

import sys
import numpy as np
import matplotlib.pyplot as plot
import matplotlib.animation as animation

def f(x):
    return min(x,1-x)
N=50
```

```
x=np.linspace(0.0, 1.0, N+1)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5
tau=lam*h*h
dt=0.001
skip=int(dt/tau)

Tmax=1
nmax=int(Tmax/tau)
n=0

fig=plot.figure()
window=fig.add_subplot(111)
line,=window.plot(x,u)

def update(i):
    global n
    if n<nmax:
        for i in range(skip):
            u[1:N]=(1-2*lam)*u[1:N]+lam*(u[0:N-1]+u[2:N+1])
        n=n+skip
        line.set_ydata(u)
    else:
        sys.exit(0)
    return line,

ani=animation.FuncAnimation(fig, update, interval=1)
plot.show()
```

7.4.3 heat1d-i-v3.py

単に陰解法にしたバージョン。

```
#!/usr/bin/env python3
# heat1d-i-v3.py

import sys
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import trid

def f(x):
    return min(x,1-x)

N=50
x=np.linspace(0.0, 1.0, N+1)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5
tau=lam*h*h
theta=0.5
dt=0.001
skip=int(dt/tau)

Tmax=1
nmax=int(Tmax/tau)
n=0
```

```
ad = (1+2*theta*lam)*np.ones(N-1)
al = -theta*lam*np.ones(N-1)
au = -theta*lam*np.ones(N-1)
trid.trilu(N-1,al,ad,au)

fig=plt.figure()
window=fig.add_subplot(111)
line,=window.plot(x,u)

def update(i):
    global n
    if n<nmax:
        for j in range(skip):
            u[1:N]=(1-2*(1-theta)*lam)*u[1:N]+(1-theta)*lam*(u[0:N-1]+u[2:N+1])
            trid.trisol(N-1,al,ad,au,u[1:N])
        n=n+skip
        line.set_ydata(u)
    else:
        sys.exit(0)
    return line,

ani=animation.FuncAnimation(fig, update, interval=1)
plt.show()
```

7.4.4 heat1d-i-v4.py

意味は分からないけれど，Matplotlib Animation Tutorial[35] の真似をして書き換えてみた。

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# heat1d-i-v4.py

import matplotlib
matplotlib.use('TkAgg')
import sys
import numpy as np
import matplotlib.pyplot as plot
import matplotlib.animation as animation
import trid

def f(x):
    return min(x,1-x)

N=50
x=np.linspace(0.0, 1.0, N+1)
u=np.vectorize(f)(x)

h=1.0/N
lam=0.5
tau=lam*h*h
theta=0.5

Tmax=1
dt=0.001
skip=int(dt/tau)
nframes=int(Tmax/dt)
print nframes

ad = (1+2*theta*lam)*np.ones(N-1)
al = -theta*lam*np.ones(N-1)
```

[35] http://jakevdp.github.com/blog/2012/08/18/matplotlib-animation-tutorial/

```
au = -theta*lam*np.ones(N-1)
trid.trilu(N-1,al,ad,au)

fig=plot.figure()
window=fig.add_subplot(111)
ax = plot.axes(xlim=(-0.02, 1.02), ylim=(-0.02, 1.02))
line, = ax.plot([], [], lw=1)

def init():
    line.set_data([], [])
    return line,

def update(i):
    if i==0:
        line.set_data(x,u)
    else:
        for j in xrange(skip):
            u[1:N]=(1-2*(1-theta)*lam)*u[1:N]+(1-theta)*lam*(u[0:N-1]+u[2:N+1])
            trid.trisol(N-1,al,ad,au,u[1:N])
        line.set_data(x,u)
    return line,

ani=animation.FuncAnimation(fig,
                            update,
                            frames=nframes, init_func=init,
                            interval=1, blit=True, repeat=False)
plot.show()
```

8　2次元熱方程式を試す

(工事中)

8.1　ウォーミング・アップで Poisson 方程式

正方形領域 $\Omega = (0,1) \times (0,1)$ で

$$-\triangle u = f \quad \text{in } \Omega, \quad u = 0 \quad \text{on } \partial\Omega$$

を解く。ただし $f \equiv 1$ とする (この点は一般的なプログラムに直したい)。

差分方程式は

$$A\boldsymbol{U} = \boldsymbol{f},$$

ただし

$$\begin{aligned}
&A = I_{N_y-1} \otimes \frac{1}{h_x^2}(2I_{N_x-1} - J_{N_x-1}) + \frac{1}{h_y^2}(2I_{N_y-1} - J_{N_y-1}) \otimes I_{N_x-1}, \\
&\boldsymbol{f} = \left(f_1, \ldots, f_{(N_x-1)(N_y-1)}\right)^T, \quad \boldsymbol{U} = \left(U_1, \ldots, U_{(N_x-1)(N_y-1)}\right)^T, \\
&f_{(j-1)(N_x-1)+i} = f(x_i, y_j) \quad (1 \le i \le N_x - 1,\, 1 \le j \le N_y - 1), \\
&U_{(j-1)(N_x-1)+i} = U_{ij} \quad (1 \le i \le N_x - 1,\, 1 \le j \le N_y - 1).
\end{aligned}$$

この差分方程式の導出については，詳しくは例えば桂田 [1] を見よ。

MATLAB では，例えば次のようなプログラムが使える。

sparse_poisson.m

```
% sparse_poisson.m --- 正方形領域における Poisson 方程式 (2009/12/29)
function [x,y,u]=sparse_poisson(n)
  h=1/n;
  J=sparse(diag(ones(n-2,1),1)+diag(ones(n-2,1),-1));
  I=speye(n-1,n-1);
  A=-4*kron(I,I)+kron(J,I)+kron(I,J);
  b=-h*h*ones((n-1)*(n-1),1);
% 2 次元化を少し工夫
  U=zeros(n-1,n-1);
  U(:)=A\b;
  u=zeros(n+1,n+1);
  u(2:n,2:n)=U;

  x=0:1/n:1;
  y=x;
% まず鳥瞰図
  subplot(1,2,1);
  colormap hsv
  mesh(x,y,u);
  colorbar
% 等高線
  right=subplot(1,2,2);
  contour(x,y,u);
  pbaspect(right,[1 1 1]);
% PostScript を出力
  disp('saving graphs');
  print -depsc2 sparsepoisson.eps
```

使い方は単純で,

```
>> sparse_poisson(100)
```

のようにする (100 は各辺を 100 等分するということ)。

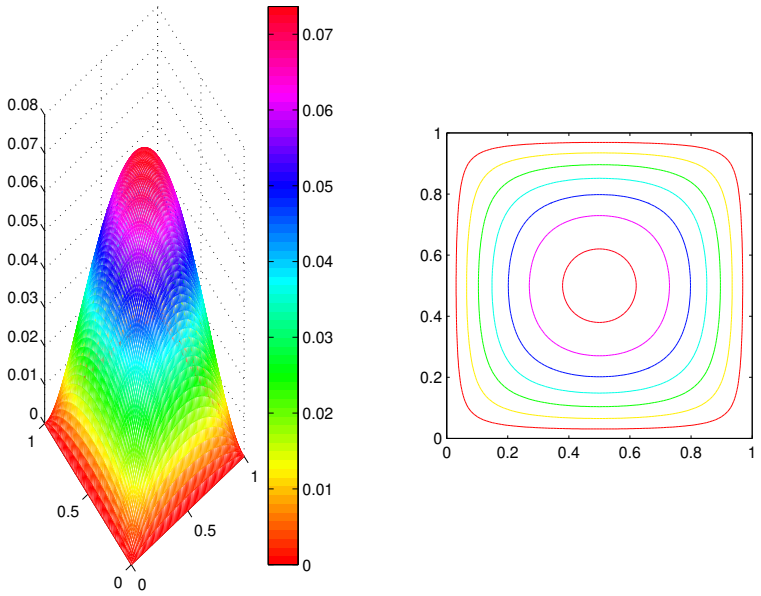


図 6: MATLAB プログラム sparse_poisson.m による計算

次に掲げるのは Python 用のプログラム (試作品) である。

poisson2_v3.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# poisson2_v3.py

import numpy as np
import scipy as sci
import scipy.sparse
import scipy.sparse.linalg
import matplotlib.pyplot as plot
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

n=100
h=1.0/n
I=sci.sparse.eye(n-1,n-1)
J=sci.sparse.spdiags([np.ones(n-1),np.ones(n-1)],[1,-1],n-1,n-1)
L=-2*I+J
A=sci.sparse.kron(I,L)+sci.sparse.kron(L,I)

b=-h*h*np.ones(((n-1)*(n-1),1))
x=sci.sparse.linalg.spsolve(A,b)

u=np.zeros((n+1,n+1))
u[1:n,1:n]=x.reshape((n-1,n-1))

x=np.linspace(0.0,1.0,n+1)
y=np.linspace(0.0,1.0,n+1)
x,y=np.meshgrid(x,y)

fig=plot.figure('Poission eq')

ax1=fig.add_subplot(121, aspect='equal')
ax1.contour(x,y,u)
ax2=fig.add_subplot(122, aspect='equal', projection='3d')
surf=ax2.plot_surface(x,y,u,rstride=1,cstride=1,
                      cmap=cm.coolwarm,linewidth=0,antialiased=False)
plot.show()
```

poisson2_v4.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# poisson2_v4.py

import numpy as np
import scipy as sci
import scipy.sparse
import scipy.sparse.linalg
import matplotlib.pyplot as plot
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

W=2.0
H=1.0
Nx=200
Ny=100
hx=W/Nx
hy=H/Ny
Ix=sci.sparse.eye(Nx-1,Nx-1)
Iy=sci.sparse.eye(Ny-1,Ny-1)
Jx=sci.sparse.spdiags([np.ones(Nx-1),np.ones(Nx-1)],[1,-1],Nx-1,Nx-1)
Jy=sci.sparse.spdiags([np.ones(Ny-1),np.ones(Ny-1)],[1,-1],Ny-1,Ny-1)
Lx=(-2*Ix+Jx)/(hx*hx)
Ly=(-2*Iy+Jy)/(hy*hy)
A=sci.sparse.kron(Iy,Lx)+sci.sparse.kron(Ly,Ix)

b=-np.ones(((Nx-1)*(Ny-1),1))
x=sci.sparse.linalg.spsolve(A,b)

# 桂田研の 2 次元配列の 1 次元配列化は実は Fortran 流！(column-major というやつ)
# そういう意味では、これを 2 次元配列に reshape() するには
#   u=np.zeros((Nx+1,Ny+1))
#   u[1:Nx,1:Ny]=x.reshape((Nx-1,Ny-1),'F')
# とするのが自然だ。
# ところが…meshgrid() で仮定されている配列は (Ny+1,Nx+1) という形だ！
# 上の u を描画するには、u.T と転置しないといけなくなる。
# そこで Fortran 流に並んでいるものを C 流 (row-major) に reshape() する。
# これで転置をしたことになる。
u=np.zeros((Ny+1,Nx+1))
u[1:Ny,1:Nx]=x.reshape((Ny-1,Nx-1))

x=np.linspace(0.0,W,Nx+1)
y=np.linspace(0.0,H,Ny+1)
x,y=np.meshgrid(x,y)

fig=plot.figure('Poission eq')

ax1=fig.add_subplot(121, aspect='equal')
ax1.contour(x,y,u)
ax2=fig.add_subplot(122, aspect='equal', projection='3d')
surf=ax2.plot_surface(x,y,u,rstride=1,cstride=1,
                      cmap=cm.coolwarm,linewidth=0,antialiased=False)
plot.show()
```

9 misc

9.1 自作モジュールのパス

そのうち由緒正しいやり方を考えるけれど。

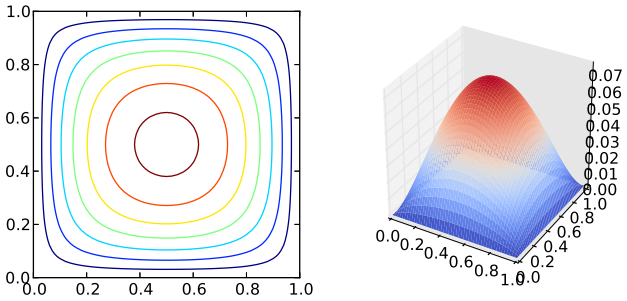


図 7: Python プログラム poisson_v2.py による計算

sys.path という変数にサーチパスが設定されるようになっているらしい。

```
>>> import sys
>>> sys.path
```

末尾に追加するには sys.path.append(’追加したいパス’) のようにする。

環境変数 PYTHONPATH に なんとか:かんとか と書くと、sys.path の先頭に “なんとか” と “かんとか” を置ける。

```
% cd
% pwd
/Users/mk
% mkdir mypython
% setenv PYTHONPATH ~/mypython
% python
Python 2.7.1 ...
オープニングのメッセージ
>>> import sys
>>> sys.path
[’’, ’/Users/mk/mypython’, ’/System/Library/....
(略)
>>>
```

9.2 時を測る

```
>>> import time
>>> time.asctime()
'Sat Jan 12 14:26:01 2013'
>>> time.time()
1357968383.564694
  (例の UNIX の時間)
>>> time.clock()
0.107231
  (そのプロセスが始まった時からの CPU 時間または実時間)
>>> time.sleep(3*60)
  (UNIX プログラマーにはおなじみの秒数指定スリープ。3 分間待つのだぞ。)
```

10 出来たら良いな

- 疎行列、特に ARPACK の利用
- 多倍長演算

10.1 疎行列をちょいと試す

まあ，8 でちょっとやってみたけれど。
「SciPy での疎行列の扱い、保存など」[36]

```
from numpy import *
from scipy import io, sparse

A = sparse.lil_matrix((3, 3))
A[0,1] = 3
A[1,0] = 2
A[2,2] = 5
```

こんな感じ？

```
solve=scipy.sparse.linalg.factorized(a)
x1=solve(b1)
x2=solve(b2)
...
xn=solve(bn)
```

まあ楽しみだ。
固有値問題もとりあえず大丈夫そう。例の問題をどれくらいの効率で解けるかが気になる。

[36] http://d.hatena.ne.jp/billest/20090906/1252269157

10.2 bigfloat で MPFR を利用する

Bigfloat[37]

インストールの記録

```
tar tzf bigfloat-0.3.0a2.tar.gz
cd bigfloat-0.3.0a2
setenv LIBRARY_PATH /opt/local/lib
setenv CPATH /opt/local/include
python2 setup.py build
sudo python2 setup.py install
```

これで良いかと思って試してみたら、早速叱られた。

```
% python2
>> from bigfloat import *
```

mpfr の module が見つからないとか。

`/opt/local/Library/Frameworks/Python.framework/Versions/2.7/lib/python2.7/site-packages/bigfloat`

に `bigfloat_config.py` というファイルを置く。僕は https://bitbucket.org/dickinsm/bigfloat/src/a43934e808cd/bigfloat_ctypes/bigfloat/bigfloat_config.py から持って来たが、実質全部注釈だった。内容は注釈を参考に書いた次の 1 行だけで良い。

```
mpfr_library_location = "/opt/local/lib/libmpfr.dylib"
```

http://pythonhosted.org/bigfloat/ などを読むと使い方が分かる。

動くかな？

```
>>> from bigfloat import *
>>> sqrt(2, precision(100))
BigFloat.exact('1.4142135623730950488016887242092', precision=100)
```

動いた。ちなみに precision の単位はビット数。デフォールトは 53 だ。

11 プログラムの終了

Python の REPL つかっているときは、`exit()` で終了する。
`import sys` として `sys.exit()` とする。
Google Colab では叱られる。
Jupyter Notebook では

12 数値計算おもちゃ箱

12.1 Newton 法

まずは定番の $x^2 - 2 = 0$ の解。

[37] http://packages.python.org/bigfloat/

test-newton.py

```
# test-newton.py
def newton(f, df, x0, eps):
  x = x0
  for i in range(10):
    dx = f(x) / df(x)
    x = x - dx
    # print('Δ x=%e, x=%g, f(x)=%e' % (dx, x, f(x)))
    if abs(dx) < eps:
      return x
  print('newton: 収束しませんでした。修正量=%e' % (dx))
  return x

def f(x):
  return x*x-2.0

def df(x):
  return 2*x

newton(f,df,2.0,1e-15)
```

方程式がパラメーターつき、つまり $f(x,\alpha)=0$ の形をしていることがあるので、オプションでパラメーターが渡せると良いのかな？

SIR モデルで、$t=0$ でほぼ全人口が感受性者の場合 $(O<I(0)\ll 1,\ S(0)\simeq 1,\ R(0)=0)$ に、最終的残る感受性者数 $S=S(\infty)$ は

$$1-S+\log\frac{S}{R_0}=0$$

の実数解のうち 1 でない方である。ここで R_0 は基本再生産数というパラメーターである (流行する $R_0>1$ という場合を考える)。

test-newton3.py

```
# test-newton3.py
import numpy as np
import matplotlib.pyplot as plt

def newton(f, df, x0, eps, *args):
  x = x0
  for i in range(10):
    dx = f(x, *args) / df(x, *args)
    x = x - dx
    # print('Δ x=%e, x=%g, f(x)=%e' % (dx, x, f(x)))
    if abs(dx) < eps:
      return x
  print('newton: 収束しませんでした。初期値=%g, 修正量=%e' % (x0,dx))
  return x

# 解きたい方程式 f(x)=0 の f()
def I(S,R0=2.5):
  return 1.0-S+np.log(S)/R0

# f() の導関数
def dI(S,R0=2.5):
  return -1.0 + 1.0/(R0*S)

# 感染せずに残る感受性者の割合を求める
n=40
R0s=np.linspace(1.1,5.0,n+1)
left=np.empty_like(R0s)
for i in range(n+1):
  R0=R0s[i]
  if i==0:
    left[i]=newton(I,dI,0.8,1e-15,R0)
  else:
    left[i]=newton(I,dI,0.5*left[i-1],1e-15,R0)

plt.plot(R0s,1.0-left)
plt.title('基本再生産数と最終的に感染した人の割合')
plt.show()
#for i in range(n+1):
#  print(left[i])
```

12.2 常微分方程式

「Python によるシミュレーション」

A　がらくた箱

A.1　クラス，型の判定

```
#!/opt/local/bin/python2

import numpy
import types

def foo(x):
  """ foo """
  if isinstance(x, float):
    print 'float'
  elif isinstance(x, complex):
    print 'complex'
  elif isinstance(x, int):
    print 'int'
  elif isinstance(x, list):
    print 'list'
  elif isinstance(x, str):
    print 'string'
  elif isinstance(x, numpy.ndarray):
    if isinstance(x[0], types.FloatType):
      print 'float array'
    elif isinstance(x[0], types.IntType):
      print 'int array'
    elif isinstance(x[0], types.ComplexType):
      print 'complex array'
    else:
      print('are')
  else:
    print 'error'

if __name__ == "__main__":
  foo(1)
  foo(1.0)
  foo(1.0+2.0*1j)
  foo([1,2,3])
  foo('Hello')
  foo(numpy.array([1,2,3]))
  foo(numpy.array([1+1j,2+2j,3+3j]))
```

A.2　poisson_v2.py

テストのために同じことを実現できそうなことを二つ書いて，結果を比較してみたり。

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# poisson2_v2.py

import numpy as np
import scipy as sci
import scipy.sparse
import scipy.sparse.linalg
import matplotlib.pyplot as plot

from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

#from pylab import *

n=50
h=1.0/n

######################################################################
# in MATLAB: J=sparse(diag(ones(n-2,1),1)+diag(ones(n-2,1),-1));

# Solution 1
data=np.array([np.ones(n-1),np.ones(n-1)])
diags=np.array([1,-1])
J=sci.sparse.spdiags(data,diags,n-1,n-1)

# Solution 2
J2=sci.sparse.diags(np.ones(n-2),1)+sci.sparse.diags(np.ones(n-2),-1)

(J-J2).todense()

######################################################################
# in MATLAB: I=speye(n-1,n-1);
I=sci.sparse.eye(n-1,n-1)

######################################################################
# in MATLAB: A=-4*kron(I,I)+kron(J,I)+kron(I,J);

# Solution
A=-4*sci.sparse.kron(I,I)+sci.sparse.kron(J,I)+sci.sparse.kron(I,J)

######################################################################
# in MATLAB: b=-h*h*ones((n-1)*(n-1),1);

# Solution 1
b=-h*h*np.ones((n-1)*(n-1))
b=sci.mat(b).T

# Solution 2
b2=-h*h*np.ones(((n-1)*(n-1),1))

x=sci.sparse.linalg.spsolve(A,b)
x2=sci.sparse.linalg.spsolve(A,b2)

x-x2

######################################################################
# in MATLAB:
#    U=zeros(n-1,n-1);
#    U(:)=A\b;
#    u=zeros(n+1,n+1);
#    u(2:n,2:n)=U;
```

```
u=np.zeros((n+1,n+1))
u[1:n,1:n]=x.reshape((n-1,n-1))

#################################################################

x=np.linspace(0.0,1.0,n+1)
y=np.linspace(0.0,1.0,n+1)
x,y=np.meshgrid(x,y)

fig=plot.figure('Poission eq')

ax1=fig.add_subplot(121, aspect='equal')
ax1.contour(x,y,u)

ax2=fig.add_subplot(122, aspect='equal', projection='3d')
surf=ax2.plot_surface(x,y,u,rstride=1,cstride=1,
                      cmap=cm.coolwarm,linewidth=0,antialiased=False)

plot.show()
```

B　数値実験のためのプログラム作成・実行手順について考える

(ここはあまり自信がない。分からない (知らない) ので自分なりに考えてみたことを整理するために書きとめた、ということ。)

プログラムの書き方は色々参考資料がある。
それはさておき、どういうふうに数値実験すべきなのだろう。

プログラムの作成・実行については、C 言語のようなプログラミング言語では、ソースプログラムをテキスト・エディタで作成して、それをコンパイルして実行する、くらいしかない。
その後、パラメーターの入力をどうするかとか (ソースプログラムに埋め込み、パラメーターを変えたい時は、プログラムを書き換えてコンパイルし直し…という人もいるだろうし、実行時にパラメーターあるいはパラメーター・ファイル名を入力、というのもある。個人的には「当然後者だろう」と思っているけど、うちの学科案外前者多いような気がする。後者だと make やシェル・スクリプトとの相性も良いと思うのだけど。)、との後も、計算の継続とか、結果をどう保存するか、どう可視化するかとか、色々考えることはあるが。

Python の処理系は (普通は) インタープリターであり、“対話的な” 処理も可能なので、最初の段階 (プログラムの作成・実行) でやり方が色々考えられる。
どうするかは何を調べたいかにもよるだろうけれど。まずは考えられるやり方を列挙してみる。

1. テキスト・エディターで Python プログラムを書き、それを `python` コマンドで実行する。

   ```
   emacs なんとか.py &
   ```

   (もちろん atom でも code でも何でもよい。)

   ```
   python なんとか.py
   ```

   このやり方はクラシックとも言えるけれど、(繰り返しになるけれど) make とかシェル・スクリプトとか相性が良さそうで、意外に便利なのでは。

2. テキスト・エディターで関数定義を書いた Python プログラムを書き、それを import して実行する。

なんとか.py

```
# なんとか.py
# -*- coding: utf_8 -*-
def かんとか(a,b,c):
  ...

if __name__ == "__main__":
  # これはおまけ的（プログラムの説明表示とか、典型的パラメーターで走らせるとか)
  ...
```

この “なんとか.py” を読み込んで使う。

```
>>> import なんとか
>>> なんとか().かんとか(a1,b1,c1)
>>> なんとか().かんとか(a2,b2,c2)
```

3. Jupyter notebook を使う。(Jupyter のインストールは、例えば MacPorts でインストールするには、`sudo port install py-jupyter` でよい。)

```
jupyter-3.9 notebook

(このシェルはビジーになっちゃうね。)
```

そうして 新規 から Python (ipkernel) などを選ぶ。

C　似ているものの使い分け: list, tuple, ndarray

リスト (list), タプル (tuple), numpy の ndarray は似ているけれど、異なる。

C.1 list

```
>>> x=[]
>>> x
[]

>>> x=[1]
>>> x
[1]

>>> x=[1,'two']
>>> x
[1, 'two']
>>> x[0]
1
>>> x[1]
'two'

>>> x[1]=2
>>> x
[1, 2]

>>> x.append(3)
>>> x
[1, 2, 3]
```

append() 出来るのはいわゆるリストだからか。

C.2 tuple

list とよく似ているが、immutable である。

```
>>> x=()
>>> x
()

>>> x=(1)
>>> x
1
>>> x=(1,)
>>> x
(1,)

>>> x=(1,'two')
>>> x
(1, 'two')
>>> x[0]
1
>>> x(1)
'two'

>>> x[1]=2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

>>> x=(1,2)
>>> x
(1, 2)

>>> x.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

C.3 ndarray

numpy の ndarry はいわゆる配列である。リストと比べてどれくらい速度に差が出るのか興味があるが、わざわざリストを使うのも馬鹿馬鹿しいので、当面 ndarray でプログラムを書くのだろう。

```
>>> import numpy as np

>>> x=np.array([])
>>> x
array([], dtype=float64)

>>> x=np.array([1])
>>> x
array([1])

>>> x=np.array([1,2.0])
>>> x
array([1., 2.])
>>> x[0]
1.0
>>> x[1]
2.0
>>> x[1]=3
>>> x
array([1., 3.])
>>> x.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'numpy.ndarray' object has no attribute 'append'
```

D 複素数

(しばらく工事中)

```
z=3+4j
z.real
z.imag
z.conjugate()
abs(z)
```

```
3+4j.real
3+4j.imag
3+4j.conjugate()
abs(3+4j)
```

実部・虚部は float らしい。

統一感がなくて、正直好きになれない。

超越関数は、math, cmath にはいっているけれど、ndarray に適用することを考えると、numpy のを使うべきか？

参考文献

[1] 桂田祐史：Poisson 方程式に対する差分法, https://m-katsurada.sakura.ne.jp/labo/text/poisson.pdf (2000 年?〜).