Python 3 覚書

桂田 祐史

2016 年 2 月 23 日, 2016 年 2 月 25 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…
- 引用符は、" でも ’ でも良い。三重クォート """, ’’’ というのもある。
- 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。
- 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、

[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

```
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。

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

とする，と書いてあるものが多い。何だろう?検索したら何か凄いことになっている。#!/opt/local/bin/pyt
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。

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。

prog2.py

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

```
% 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。

2.3

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
%
```

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

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 数値計算

3.1 Numerical Python (NumPy)

せっかちな MATLAB 使いには，NumPy for Matlab Users[8] という WWW ページが分りやすいかも。でも…

3.1.1 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

[8] http://www.scipy.org/NumPy_for_Matlab_Users

```
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。)

3.1.2 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]])
```

3.1.3 zeros(), ones(), 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.]])
```

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.]])
```

3.1.4 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。

3.1.5 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)
```

3.1.6 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]])
```

3.1.7 これはどうやる？

- a\b は

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

3.2 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。

3.2.1 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」[9] にある。
MacPorts の場合は，NumPy のインストールと同様に，次の 1 行コマンドで OK.

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

MacPorts は簡単でよろしい。

3.2.2 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

```
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。

3.2.3 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

[9] http://nalab.mind.meiji.ac.jp/~mk/labo/text/python/node24.html

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)。

4 matplotlib

ドキュメントはどこですか？

User Guide[10] を読んでいたけれど，The Matplotlib FAQ の Usage[11] を読んで初めて腑に落ちたことが多い。

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

4.1 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()
```

4.2 pylab

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

4.3 メモ

Matplotlib Examples[12] 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

[10] http://matplotlib.org/users/index.html
[11] http://matplotlib.org/faq/usage_faq.html
[12] http://matplotlib.org/examples/index.html

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

と書いておくものなのかな？

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[13] 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[14] に色々書いてある (これはなかなか良さそうな説明)。

5　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

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

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

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

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」[15])。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。

5.1　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。

[13] http://matplotlib.org/users/shell.html
[14] http://matplotlib.org/faq/usage_faq.html\#what-is-a-backend
[15] http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat1d-everywhere/

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。

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
```

5.2 陰解法

いわゆる θ 法 (「発展系の数値解析」[16]) 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」[17] 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。

[16] http://nalab.mind.meiji.ac.jp/~mk/labo/text/heat-fdm-0.pdf
[17] 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。)

5.3 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」[18] 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」[19] を読んでやり直し。無事に出来た。

5.3.1 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 */
};
```

[18] http://d.hatena.ne.jp/ousttrue/20091205/1260035679
[19] 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);
}
```

5.3.2 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; ← 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);
}
```

5.3.3 setup.py

「Numpy の配列を利用する C モジュールを作る」[20] 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=[]
        )
    ]
)
```

5.3.4 ビルド&インストール

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

5.3.5 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
```

[20] 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])
```

5.4 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。

5.4.1 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。

5.4.2 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()
```

5.4.3 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()
```

5.4.4 heat1d-i-v4.py

意味は分からないけれど，Matplotlib Animation Tutorial[21] 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)
```

[21] 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()
```

6　2次元熱方程式を試す

(工事中)

6.1　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)。

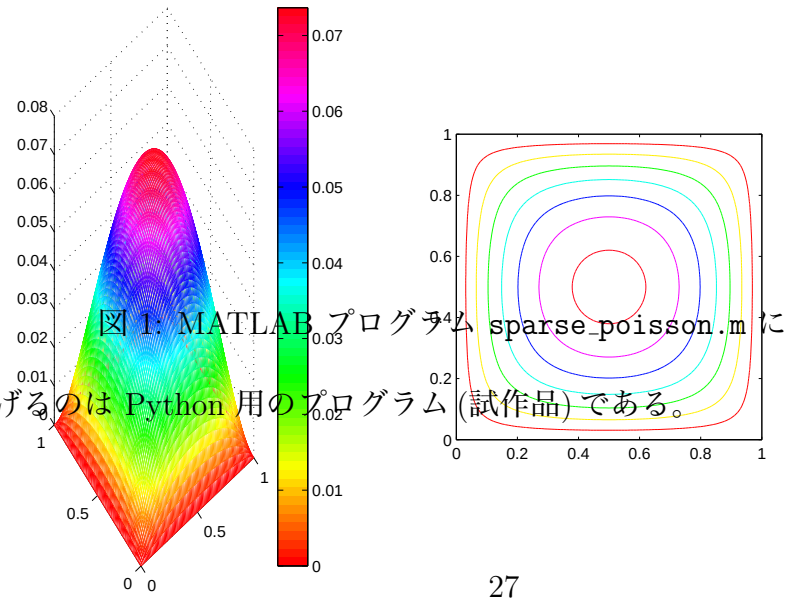

図 1: MATLAB プログラム sparse_poisson.m による計算

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()
```

7 misc

7.1 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。

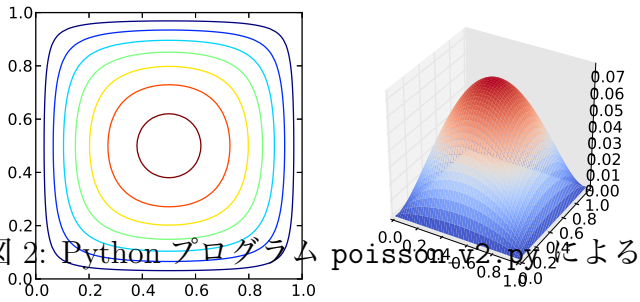

図 2: Python プログラム poisson_v2.py による計算

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)
>>>
```

7.2 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。)
```

8 出来たら良いな

- 疎行列、特に ARPACK の利用
- 多倍長演算

8.1 疎行列をちょいと試す

まあ，6 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」[22]

```
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。

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

8.2 bigfloat で MPFR を利用する

Bigfloat[23]

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。

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

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()
```

参考文献

[1] 桂田祐史：Poisson 方程式に対する差分法, http://nalab.mind.meiji.ac.jp/~mk/labo/text/poisson.pdf (2000 年?〜).