# Python 覚書

桂田 祐史

2012 年 12 月 24 日, 2017 年 12 月 10 日

## 1 はじめる

### 1.1 Python を始める理由

一言で言うと，アンテナを張っていて，面白そう，数値計算環境として便利そう，と感じていたところ，最後のひと押しのきっかけがあったからである。

2010 年度の卒研では，Ruby で数値計算したのだが，その際は，どうせ常微分方程式の初期値問題で，計算のスピードは大して必要ではないだろう，ネタ本 (とんでる力学) で使われていて，ベクトルを気軽に使えそうで，Runge-Kutta 法なども見た目スマートに書けそうだなあ，一つ試してみよう，と気楽に考えたのだが，色々調べていて，結構大きい計算をしている人もいるらしいことが分った。考えてみれば，最近使っている MATLAB も基本はインタープリターで，ただ大きめの仕事をする LAPACK のルーチンなどを高い効率で実現して，そういうものを呼び出し効率を稼いでいることに気がついた。インタープリターであることは必ずしも実効速度が遅いことを意味しない。Ruby や，それと競っている Python のような言語での数値計算は，やりようによっては使い物になるかも？と考えるようになった。

一方で MATLAB を使い込むにつれ，その言語としての拡張性に疑問を持つようになった。INTLAB のような成功例はあるものの，自分で拡張をしようとして調べだすと，正直嫌気がさすところがあった。

そんなときに，実際に Python で数値計算をしている人を見た。やはり百聞は一見に如かずで，面白そうに感じた。試してみようという気持ちが一気に高まった。

これを書いているのは，触り始めてから 3 週間が経過した時点である。実に色々な数値計算ライブラリィが Python の上で利用できるようになっていることが分った。

まだまだ分からない部分も多いし，現時点では効率で MATLAB に及ばないところも確かにあるようだ。それでも，これからの数値計算環境として有望そうという感触は変わっていない。これで得られる経験は，Python 自身が実を結ばなくても後々役に立つのではないか？と思っている。

### 1.2 Mac での利用 (1)

Mac OS X では，最初からインストールされている。

でも SciPy のサイトでは新しいのをインストールしろとおっしゃる。http://www.python.org/download/ から `python-2.7.3-macosx10.6.dmg` を入手してやるのかな？他に MacPorts で入れるという手もある (次項で解説)。

この項では，最初からインストールされている Python を利用する方法を説明する。

### 1.2.1 readline

元々 Python は GNU readline を使うように作られているらしいが，Mac では GPL 汚染を嫌って，GNU readline の代わりに NetBSD の editline がインストールされているらしい。これだと日本語の入力が出来ない。

Stand-alone readline module[1] から `readline-6.2.4.1-py2.6-macosx-10.6-universal.egg` (使っているのが OS X 10.6 だったから) を入手して、次のようにインストールした。

これだけで済みました

```
easy_install readline-6.2.4.1-py2.6-macosx-10.6-universal.egg
```

簡単で良いですね。

## 1.3 Mac での利用 (2)

MacPorts にも色々用意されている。これはこれで簡単で良い。python2 という名前で呼び出すことが出来るので，OS X に含まれているのと区別して使うのは簡単だ。

```
port search python
port variants python27
```

```
sudo port install py27-numpy
sudo port install py27-scipy
sudo port install py27-matplotlib
```

(最後の Numeric は不要かも知れない。)

## 1.4 情報

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 v3.2.3 documentation[2]
- Python 2.7ja1 documentation[3] (2012 年 12 月現在 Stable な 2.7 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)

## 1.5 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。

[1] http://pypi.python.org/pypi/readline
[2] http://docs.python.org/3.2/
[3] http://docs.python.jp/2/
[4] http://docs.python.jp/2/tutorial/index.html
[5] http://www.python.org/dev/peps/pep-0008/

- 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."
```

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

あるいは先頭に

```
#!/usr/bin/python
```

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。

```
#!/usr/bin/env python
```

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。#!/usr/bin/python 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

prog2.py

```
# -*- coding: utf-8 -*-

#import sys
#import codecs

#sys.stdout = codecs.getwriter('utf_8')(sys.stdout)

print u"こんにちは、パイソン。"
```

日本語の入出力は大変で，こうせい、ああせい，と細かい処方箋が書いてあって，大変そうと思ったのだけど、それはどうも古い話のようで，コメント・アウトしても動いた。要点は次の二つ。

1. エンコーディングを注釈で指定する。
   coding:　なんとか あるいはcoding=なんとか というパターンを読むらしい。次のようにするものだとか。
   - Emacs 使いは `# -*- coding:  utf-8 -*-`
   - vim 使いは `# fileencoding=utf8`
2. 文字列リテラルを表す引用符 " の前に u つける。

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

## 2.3

prog3.py

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

print u"これは常に実行される"

def test():
    print u"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
%
```

## 2.4　とにかく for

```
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。

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

の方が速いという説 (少なくとも v2.7 では真かも) とか，version 3 ではそうでないという説とか。

# 3 数値計算

## 3.1 Numerical Python (NumPy)

せっかちな MATLAB 使いには，NumPy for Matlab Users[6] という WWW ページが分りやすいかも。でも…

### 3.1.1 インストール

MacOS 付属の Python には最初から含まれている。
MacPorts の Python 用の Numpy は，例えば次のようにインストールできる。

```
sudo port install py27-numpy
```

### 3.1.2 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

[6] 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')
```

### 3.1.3 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[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]])
```

### 3.1.4 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.]])
```

### 3.1.5 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。

### 3.1.6 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)
```

### 3.1.7 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]])
```

### 3.1.8 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

(当然 Mac での話です。そのうち Linux やったりするだろうけど。)
ソースでインストールするには，http://sourceforge.net/projects/scipy/files/scipy/ から scipy-0.11.0.tar.gz のようなのを入手する。

```
% tar xzf scipy-0.11.0.tar.gz
% cd scipy-0.11.0
% python setup.py build
% sudo python setup.py install
```

とするみたいだけど，途中で失敗したりする。あ，gfortran がいるんですね…

ひとり言 (2012/12/31)

WWW ページでお勧めの gfortran がやけに古いなあ。あんまり信用出来ないや。MacPorts で既に入っていた gfortran を使ってみる。

```
% ln -s /opt/local/bin/gfortran-mp-4.5 ${SOMEWHERE}/bin/gfortran
% rehash
```

気を取り直して build やり直し。警告出まくりで気持ちが悪いけれど，これは各種数値計算ライブラリィのソースプログラムのお行儀のせいかな。fft がらみとか ARPACK とか。特殊関数らしいのとか。色々入っているようでワクワクする。あ，build できた。

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 py27-scipy
```

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ました，というストーリーです。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)
```

これを Scipy でやってみる。

```
>>> 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)
>>> 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.]])
```

さて大きい行列でやってみよう。

```
>>> n=10000
>>> a=mat(random.random((n,n)))
>>> P,L,U=sci.linalg.lu(a)
>>> P=mat(P)
>>> L=mat(L)
>>> U=mat(U)
>>> x=mat(ones((n,1)))
>>> b=a*x
>>> x2=linalg.solve(U,linalg.solve(L,P.T*b))
  (遅いね。もう一度解いているのかも。)
>>> linalg.norm(x-x2)
7.1042857390400037e-09
```

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だ。(MATLAB では，三角行列による割算 (L\ , U\ ) は高速に出来るので，この計算手順がまっとうなものとなる。)

(翌日)

はて？sci.linalg.lu_factor(), sci.linalg.lu_solve() を使うのか？

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

a=sci.mat([[1,2],[3,4]])
lu,piv=sci.linalg.lu_factor(a)
x=sci.mat([1,2]).T
b=a*x
x2=sci.linalg.lu_solve((lu,piv),b)

n=10000
a=sci.mat(np.random.random((n,n)))
lu,piv=sci.linalg.lu_factor(a)
x=np.ones((n,1))
b=a*x
x2=sci.linalg.lu_solve((lu,piv),b)
sci.linalg.norm(x-x2)
```

どうもそうみたい。こちらはずっと速い。

# 4 matplotlib

(2017/12/10 追記: この節で例にあげたプログラムは、確かに動いていたのだけど、久しぶりに動かそうとして、動かないものが多くなっていた。記憶が曖昧だけれど、この文書の Python3 バージョン[7] では、色々と書き方を変えていて、そちらのプログラムは現在でも動くし、Python2 でも小修正で動く。)

ドキュメントはどこですか？
User Guide[8] を読んでいたけれど，The Matplotlib FAQ の Usage[9] 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

[7] http://nalab.mind.meiji.ac.jp/~mk/labo/text/python3/
[8] http://matplotlib.org/users/index.html
[9] http://matplotlib.org/faq/usage_faq.html

## 4.1 pyplot

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

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[10] 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？

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[11] 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[12] に色々書いてある (これはなかなか良さそうな説明)。

[10] http://matplotlib.org/examples/index.html
[11] http://matplotlib.org/users/shell.html
[12] http://matplotlib.org/faq/usage_faq.html\#what-is-a-backend

# 5　1 次元熱方程式を試す

(2017/12/4 加筆: この節のプログラムは、作った当時は*もちろん*全部動いていたのだけど、今では動かなくなったものが多い。Python3 バージョンの方のプログラム http://nalab.mind.meiji.ac.jp/~mk/labo/text/python/ はこれより新しくて今でも動くようだ。次を参照: https://stackoverflow.com/questions/11874767/real-time-plotting-in-while-loop-with-m

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」[13])。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://nalab.mind.meiji.ac.jp/~mk/labo/text/heat1d-everywhere/

heat1d-v0.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# heat1d-v0.py

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

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)

plot.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:
        plot.plot(x,newu)
    u=newu.copy()

plot.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，`newu[]` を省略するとか，計算しながら表示するとか，`range()` の代わりに `xrange()` を使うとか。

heat1d-e.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# heat1d-e.py
# http://d.hatena.ne.jp/Megumi221/20080306/1204770689 を参考にした

import numpy as np
import matplotlib.pyplot as plot

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)

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

for n in xrange(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)
    plot.draw()
```

(細かい話ですが，某マシンで range() では 84.822 秒，xrange() ではユーザー時間 80.218 秒, 経過時間 2 分 38.58 秒。)

## 5.2 陰解法

いわゆる $\theta$ 法 (「発展系の数値解析」[14]) 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」[15] 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。

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

heat1d-i.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# 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 plot

def trilu(n, al, ad, au):
    for i in xrange(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 xrange(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)
# 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)

plot.ion()
line,=plot.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)
trilu(N-1,al,ad,au)

for n in xrange(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)
    plot.draw()
```

(ユーザー時間 82.188 秒, 経過時間 2 分 39.32 秒というわけで，あまり遅くなっていない。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 (?)

「三次元日誌 numpy の配列を受け取る C モジュールを作る」[16] を参考にして C モジュールを作ってみた。高速化のためというよりは，C モジュールはどうやって作るかを知りたかった，というのが本当の目的である (1 次元問題のシミュレーションで，計算の速さはあまり問題になりそうもないが，C モジュールを作るノウハウは後で必要になる可能性が高い。)。

### 5.3.1 trid.c

「公開プログラムのページ」[17] にある trid-lu.c が叩き台。C モジュールを実現するための方法は,「numpy の配列を受け取る C モジュールを作る」[18] を読んで理解しました。

```
/*
 * trid.c --- 三重対角行列の LU 分解をする Python 用 C モデュール
 *
 *  written by mk, on 5 Januaray 2013.
 *  http://www.math.meiji.ac.jp/~mk/program/linear/trid-lu.c
 *  http://d.hatena.ne.jp/ousttrue/20091205/1260035679
 *  http://codeit.blog.fc2.com/blog-entry-9.html
 *
 */

/* 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];
}

void trid(int n, double *al, double *ad, double *au, double *b)
{
  trilu(n,al,ad,au);
  trisol(n,al,ad,au,b);
}

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

PyObject *trid_trilu(PyObject *self, PyObject *args)
```

[16] http://d.hatena.ne.jp/ousttrue/20091205/1260035679
[17] http://nalab.mind.meiji.ac.jp/~mk/program/
[18] http://d.hatena.ne.jp/ousttrue/20091205/1260035679

```
{
  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; も OK?
}

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 trid_methods[] = {
  {"trilu",  trid_trilu,  METH_VARARGS, "LU factorize a tridiagonal matrix"},
  {"trisol", trid_trisol, METH_VARARGS, "solve linear eq."},
  {NULL,NULL,0,NULL}
};

// trid モデュールなので、inittrid という名前でないといけない
PyMODINIT_FUNC inittrid()
{
```

```
  (void)Py_InitModule("trid", trid_methods);
  import_array();
}
```

### 5.3.2 setup.py

「Numpy の配列を利用する C モジュールを作る」[19] 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=[
            'trid.c'
            ],
            include_dirs=[] + get_numpy_include_dirs(),
            library_dirs=[],
            libraries=[],
            extra_compile_args=[],
            extra_link_args=[]
        )
    ]
)
```

### 5.3.3 ビルド&インストール

```
python setup.py build
sudo python setup.py install
```

### 5.3.4 heat1d-i-v2.py

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# 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 plot
import trid

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)
```

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

```
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)

plot.ion()
line,=plot.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 xrange(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)
    plot.draw()
```

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])
```

ユーザー時間 81.806 秒, 経過時間 2 分 32.75 秒で陽解法に迫る。まあ，でもここまでやると明白だけれど，実行時間のボトルネックは描画にある。下手をすると十進 BASIC より遅い (笑)。

## 5.4 animation.FuncAnimcation() で高速化？

`animation.FuncAnimcation()` を使うと速くなる ??

### 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。

```
#!/opt/local/bin/python2
# -*- 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。

```
#!/opt/local/bin/python2
# -*- coding: utf-8 -*-
# heat1d-i-v3.py

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

def update(i):
```

```
    global n
    if n<nmax:
        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])
        n=n+skip
        line.set_ydata(u)
    else:
        sys.exit(0)
    return line,

ani=animation.FuncAnimation(fig, update, interval=100)
plot.show()
```

### 5.4.4 heat1d-i-v4.py

意味は分からないけれど，Matplotlib Animation Tutorial[20] 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)
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([], [])
```

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

```
    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)。

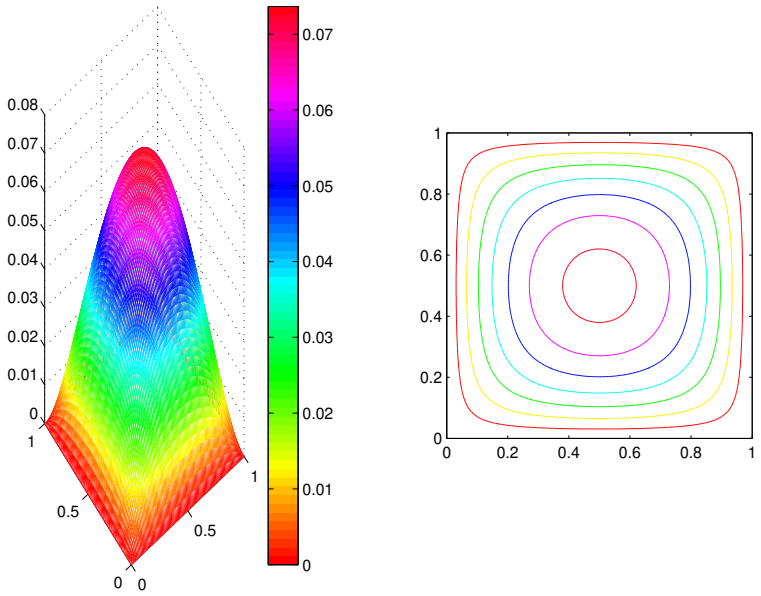


図 1: MATLAB プログラム sparse_poisson.m による計算

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。

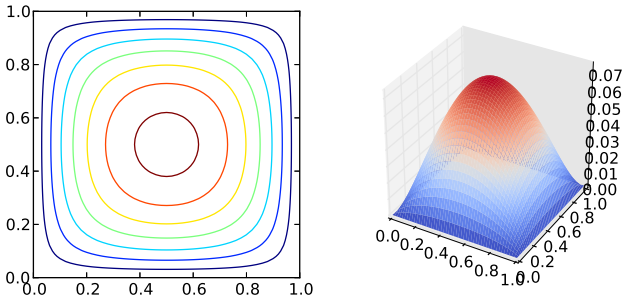


図 2: Python プログラム poisson_v2.py による計算

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。)
```

## 7.3 コマンドライン引数

C のプログラムの argc, argv が欲しい。
sys モジュールを使えば良い。

```
import sys

...

argv = sys.argv
argc = len(argv)
```

# 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」[21]

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

```
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。

## 8.2 bigfloat で MPFR を利用する

Bigfloat[22]

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。

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

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。

# 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?〜).