Pythonを利用した数値計算プログラミングの教育について

箕原 辰夫*1
Email: minohara@cuc.ac.jp

*1：千葉商科大学政策情報学部政策情報学科

◎Key Words: Python, 数値計算, プログラミング

1. はじめに

社会系は無論、理工系でも昨今は数値計算の科目がプログラミング教育課程の中に含まれていないことがある。情報処理学会の標準カリキュラム J07 でも必修科目としては組み込まれていなかったが、数値計算の教育の重要性は大学の教員の間でも再認識されるようになってきた。モデリング・シミュレーション解析や物理シミュレーションを行なうエンジンもあるが、それらのエンジンが様々な用途全てに対応できるとは限らず、学生が数値計算を自分で記述しなければならない状況もある。

これまで、FORTRAN や C 言語は、コンパイラ系の言語なのでネイティブのバイナリコードに変換することができ、高速に計算ができるので数値計算のプログラミング教育でも、これらの言語が利用されてきた。近年、Python はインタープリタ系の言語であるにも拘わらず、Num.py や Sci.py といった高精度な数値計算ライブラリも用意されて、Parallel Python や MPI for Python といった並列計算を行なうような環境を用いて、数値計算を高速に実行することが可能になってきた。取扱いの簡単さも考えて、今後は Python での数値計算の流れが大きくなるだろう。本稿ではそのときに、Python を用いた数値計算・数値解析のプログラミングの教育課程はどのようにすべきかについて考察する。

2. Python での数値計算の利点

Python は、数値計算あるいは数値解析の分野で近年目覚ましく発展し、よく使われるようになってきた。インタープリタですぐに計算が求まり、米国の大学でも初期プログラミング教育の言語として広く普及してきたことがその要因である。加えて、コンパイラ系の言語に対しての処理速度の遅さを改善すべく、より高速な処理系、ネイティブコードまでコンパイルを伴う処理系、並列に実行する処理系などが開発されたこともその普及を助けている。そのため、コンパイラ系の言語に対して遜色のない実行速度を確保するに至っており、スーパーコンピュータ上でも利用することが可能になっている。

2.1 標準の多桁整数型

Python では、整数型については、2.7 版以降は標準で多桁整数型になっている。そのため、階乗やべき乗などを計算しても、値が大きいからといって桁溢れすることなく計算結果が正確に求めることができる。

```
def factorial( n ):
    return n if n <= 1 else n * factorial(n-1)

>>> factorial( 100 )
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000L
```

2.2 ライブラリによる多精度実数

実数型については、Python は IEEE-754 の倍精度の浮動小数点実数で実現されている。これは、仮数部が 15～17 桁の精度しかない。整数型と同様に多桁の仮数部を実現するものとして、標準のライブラリに decimal モジュールが用意されている。仮数部の精度を指定して、Decimal クラスの実数を使った場合、その精度での計算結果が保証される。

```
from decimal import *
getcontext().prec = 80   # 精度を 80 桁で指定

>>> print( Decimal( "2.0").sqrt( ) )
1.4142135623730950488016887242096980785696718753769480731766797379907324784621070
```

2.3 Num.py と Sci.py

Python およびその派生の言語処理系のほとんどで Num.py(1) と Sci.py(2) という数値計算用のライブラリを用いることができる。これは、標準ではないので、専用のサイトからダウンロードする必要がある。Num.py は、ベクタ・行列格納用のライブラリになっており、Python 自体にもリストなどがあるが、行列演算については、Python のプログラムで実行するよりも、C 言語がコンパイルされた Num.py の Array ベースで演算させるほうが高速に計算することが可能となっている。

Sci.py は、Num.py のライブラリを利用して行列計算、クラスタリング、高速フーリエ変換、微分方程式の求積、

積分、補完、疎行列計算などの一般的に必要な数値計算に加え、多項式展開、画像解析、統計解析、回帰解析、信号解析、最適化などの数値解析ライブラリが含まれている。以下のスクリプトの例は、Num.py と Sci.py の線形代数演算を用いて、正方行列の行列式の値と逆行列を計算するものである。

```
import numpy as np
from scipy import linalg

arr = np.array( [ [1, 2], [3, 4] ] )
print( linalg.det( arr ) )
print( linalg.inv( arr ) )
```

2.4 高速計算・並列計算のサポート

Python のオリジナルの処理系（CPython と呼ばれる）はインタープリタであるが、これをコンパイルして高速に実行する言語処理系が用意されている。Cython[3]は型注釈のある Python のプログラムをネイティブコードに変換するコンパイラである。途中で生成される変換された C 言語のプログラムもみることが可能になっている。単純なプログラムでも 150 倍も高速になる。Cython では、OpenMP[4]にも対応しているので、マルチコアの並列計算にも展開することが可能となっている。Numba[5]は Num.py に対応した実行時にコンパイルされる処理系である。コンパイルしたい関数の定義に@jit という装飾子をつける。これも Cython と同様の高速性があり、OpenMP にも対応している。PyPy[6]は、Python のプログラムをそのまま走らせることができる処理系で、コンパイラを内蔵している。CPython に比べて、7.5 倍実行速度があがっている[7]。PyPy の問題点は、Num.py や Sci.py には充分対応していないことで、それぞれの数値計算について、独自のライブラリを用いる必要がある。

OpenMP をベースとするマルチコアによる並列計算は、Cython や Numba でも利用可能であるが、処理系としては、Parallel Python[8]が用意されている。メモリ共有型以外の並列計算の処理系については、まず標準でも、multiprocessing モジュールがあり、並列計算をサポートしている。Open MPI[9]などのメッセージ・パッシング・ベースでの並列計算のライブラリを Python で用いるための移植として、mpi4py[10]や PyMPI[11]などが用意されている。GPGPU を用いた計算をするために、Numba の拡張モジュールとして Anaconda Accelerate[12]という市販のパッケージが用意されている。これで、CUDA[13]などのライブラリを用いることが可能となっている。また、フリーのライブラリとして、PyCUDA[14]も用意されている。一方、OpenCL[15]をベースとする並列計算については、PyCUDA と同様に PyOpenCL[16]が用意されている。

2.5 計算結果の視覚化

計算結果の視覚化については、Python と親和性のある Matplotlib[17]が主に使われており、Num.py および Sci.py と Matplotlib の三者を利用することが多い。非常に短いコードでグラフ描画や画像処理が行なえる。

3. 数値計算教育の重点項目

Python を用いた数値計算のカリキュラムについては、Num.py を使った行列への演算からはじめて、Sci.py の各ライブラリの使い方を教えていけば、1 つの半期の科目としては成立するのではないかと思われる。しかし、その手前の数値計算のポイントをカリキュラムの内容に入れる必要があると思われるので、いくつかのポイントを指摘したい。

3.1 収束について

反復法や分割法を用いた場合など、真数に近づくための収束の速さは、どのような数値計算を行なう場合でも教育上では問題にすべきである。たとえば、ニュートン法を用いて平方根を計算する場合を考える。この場合、ほぼ 10 回程度の反復で、IEEE-754 の倍精度実数の仮数部の精度で値を求めることが可能となる。ニュートン法は収束が速い数値計算の例として教えるべきで、このような優れた方法は少ないことを別の例題を用いて比較した方が良い。また、Decimal クラスを利用して、100 桁〜1000 桁の精度で計算をさせてみると良い。

```
def squareroot( n ):
    x = 1.0
    for i in range( 10 ):
        x = x / 2.0 + n / (2.0 * x)
    return x

print( "prog", squareroot( 2.0 ) )
print( "math", math.sqrt( 2.0 ) )
```

実行結果>>>
('prog', 1.414213562373095)
('math', 1.4142135623730951)

分割法の例として、数値積分で四分の一の円を用いて、円周率の値を求める例題を用いて、四角形で近似させる場合、台形公式で近似する場合、シンプソンの公式で近似する場合の三者を比較させるのも良い。

```
def f( x ):
    y = 1.0 - x ** 2
    return math.sqrt( y ) if y >= 0 else 0

n = 10000000 # division number
delta = 1.0 / n # one step
```

```
# 四角形での近似、台形公式での近似
x, y = 1.0, 0.0    # initial value
rs = 0.0 # rectangle での面積
ts = 0.0 # trapezoid での面積
for i in range( n ):
    rs += f( x ) * delta
    nx = x - delta
    ny = f( nx )
    ts += ( y + ny ) * delta / 2.0
    x, y = nx, ny

# シンプソンの公式での近似
x = 1.0
ss = f( 1.0 ) # simpson での面積
for i in range( 1, n ):
    x -= delta
    if i % 2 == 0: ss += 2 * f( x )
    else: ss += 4 * f( x )
ss += f( 0.0 )
ss = ss * delta / 3.0
```

図 1 は、上記の結果を、分割数 n を 10 から 1000 万まで変えて、math.pi の値と異なる桁数をプロットしたものである。台形公式やシンプソンの公式を使った方が、より少ない分割数で精度が高くなっているのがわかる。

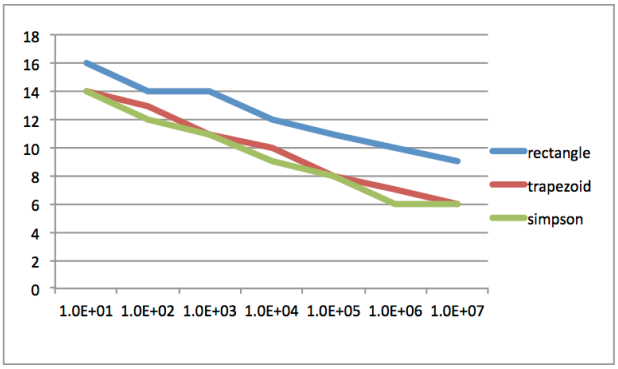

図 1　仮数部 17 桁中、math.pi と異なる桁数

π の値を求める方法は、昔からいろいろと提案されているが、数値計算のプログラミング教材としては、いかに速く、より多くの桁数を求められるかという数値計算の性能評価という目的に使える。たとえば、一番悪い例として、モンテカルロ法を用いた場合、1 億回の繰返しで実行しても 5 桁ぐらいの桁数しか求められない。オイラーのζ関数で求める方法や、上記の数値積分で求める方法などもあるが、あまり早くは収束しない。高速により多くの桁数を求める方法としては、arctan 関数のグレゴリー・ライプニッツ級数展開を用いて、マチンの公式の派生形で求められるのが一般的である。以下にこの級数展開を第 100 項まで求める仮数部の精度 200 桁の arctan 関数を Python で定義する。

```
from decimal import *
getcontext().prec = 200

def arctan( x ):
    y = Decimal( x ** Decimal( 2 ) )
        / Decimal( Decimal( 1 ) + x **Decimal( 2 ) )
    result = Decimal( 1 )
    num = 2
    numprod = 1
    div = 3
    divprod = 1
    for i in range( 1, 101 ):
        numprod *= num
        divprod *= div
        result += y ** Decimal( i ) * Decimal( numprod )
        / Decimal( divprod )
        num += 2
        div += 2
    return result * y / x
```

さて、マチンの公式の派生形として、以下のオイラーの公式、ストーマーの公式、ガウスの公式の 3 つを用いて仮数部が何桁まで合っているか求めた。

オイラーの公式：

$$\frac{\pi}{4} = 5\arctan\frac{1}{7} + 2\arctan\frac{3}{79}$$

ストーマーの公式：

$$\frac{\pi}{4} = 6\arctan\frac{1}{8} + 2\arctan\frac{1}{57} + \arctan\frac{1}{239}$$

ガウスの公式：

$$\frac{\pi}{4} = 12\arctan\frac{1}{18} + 8\arctan\frac{1}{57} - 5\arctan\frac{1}{239}$$

オイラーの公式では 172 桁まで、ストーマーの公式では 184 桁まで、ガウスの公式では 198 桁まで合ったものが求まった。このように π の値自体を求めることには意味がないが、その収束の速度を考慮すること、および異なる方式を用いて速度の比較をすることは重要な教育ポイントであると言える。

3.2　予測手段としての数値積分

物理法則など、瞬間的な力が微分方程式で解析的に得られるが、実際の状況に合わせてある一定時間後の物理的な結果が変わることが多い。たとえば、弾道を計算するのにも、弾道空間の中の風速が異なれば結果が異なってくる。これは、ゴルフボールの弾道のシミュレーションを想定しても明らかであろう。そのために、風速の分布やベクトルなどを考慮した形での予測をするためには、微分方程式を数値積分する必要がある。一般に用いられるのは、修正オイラー法やルンゲクッタ法である。これも、単純なオイラー法、修正オイラー法、ルンゲクッタ法のそれぞれを用いて、どの程度の差分がでるのかを数値計算の教育の中で体験しておく必要がある。

3.3　誤差の回避

前に示した数値積分のプログラムでも、計算結果を足し込むのに、小さい数同士から計算させるようにしている。

これは加減算のオーダーを合わせるためのものである。数値計算をするときには、数値の桁合わせで仮数部が重なるような形で加減算をする必要性を強調すべきである。また、同じ数値積分のプログラムで示した通り、実数用の平方根の関数 sqrt を呼ぶときには、引数が誤差の関係からマイナスになっている可能性を考えてプログラミングする必要がある。

単純な三角関数を呼ぶ際にも、倍精度実数の円周率の仮数部の精度が 16 桁程度しかないことを考慮するべきであり、最終的な三角関数の仮数部の精度を 15 桁程度に絞った方が理論値に近くなる。誤差を回避することと、理論上必要な適切な有効桁数を選ぶことは、数値計算の基礎と考えることができる。

```
th60 = math.radians( 60 )
th180 = math.radians( 180 )
print( math.cos( th60 ), math.sin( th180 ) )
print( round( math.cos( th60 ), 15 ),
       round( math.sin( th180 ), 15 ) )
実行結果>>>
(0.5000000000000001, 1.2246467991473532e-16)
(0.5, 0.0)
```

また、ガウスの消去法のピボットのように、どの数を軸とするかを選ぶのは重要となってくる。ピボットの取り方によっては、計算が容易に発散する可能性も否めない。

3.4 特徴抽出のための信号処理

Sci.py を用いた高速フーリエ変換では、信号に対しての周波数特性を得ることができる。フーリエ変換を用いることによって、見えてこなかったなんらかの特性が発見される可能性は高い。数値計算の応用としてもフーリエ変換および逆フーリエ変換をライブラリの関数を呼び出して行なわせることは是非教えたいものである。

4. ACM の Computer Curricula での扱い

ACM の 2001 版の Computer Curricula[18]では、数値計算の分野はその応用も含めて、Computational Science としてまとめられている。この分野の副項目は、以下のようにすべて選択の項目となっている。

CN1. Numerical analysis [選択]
CN2. Operations research [選択]
CN3. Modeling and simulation [選択]
CN4. High-performance computing [選択]

これを受けた情報処理学会の J07[19]のカリキュラムでも必修の項目としては入っていない。しかしながら、2013 年版[20]では、以下のように導入の部分が必修 1 時間となっている。

CN/Introduction to Modeling and Simulation [必修 1 時間]
CN/Modeling and Simulation [選択]
CN/Processing [選択]
CN/Interactive Visualization [選択]
CN/Data, Information, and Knowledge [選択]
CN/Numerical Analysis [選択]

導入の部分については、プログラミングの実習の中で数値積分や誤差なども扱えるので、数値解析の講義を置く必要はない。しかしながら、理工学だけでなく、社会科学での応用も考えれば、数値解析の重要項目を抑えておくための講義はカリキュラムの中に必要なのではないかと考えることができる。特に、Python および Sci.py を使っての応用計算の体験と、本稿で述べたような重要項目については、時間を割く必要があるのではないかと考えられる。

5. おわりに

本稿では Python を使った数値計算の教育の優位点、数値解析用のライブラリ、高速・並列処理系の環境、およびカリキュラムの必修科目としての重要性を述べた。今後、Python を使った数値計算・数値解析の実習・講義が日本の大学教育で増えていくことを望みたい。

参考サイト

(1) Num.py, http://www.numpy.org/
(2) Sci.py, http://www.scipy.org/
(3) Cython, http://cython.org/
(4) OpenMP, http://openmp.org/
(5) Numba, http://numba.pydata.org/
(6) PyPy, http://pypy.org/
(7) PyPy Speed Center, http://speed.pypy.org/
(8) Parallel Python Software, http://www.parallelpython.com/
(9) Open MPI, https://www.open-mpi.org/
(10) mpi4py, https://pypi.python.org/pypi/mpi4py
(11) pyMPI, http://pympi.sourceforge.net/
(12) Anaconda Accelerate, https://docs.continuum.io/accelerate
(13) NVIDIA, Developing with CUDA, http://www.nvidia.co.jp/object/cuda-parallel-computing-platform-jp.html
(14) PyCUDA, https://mathema.tician.de/software/pycuda/
(15) OpenCL, https://www.khronos.org/opencl/
(16) PyOpenCL, https://mathema.tician.de/software/pyopencl/
(17) Matplotlib, http://matplotlib.org/

参考文献

(18) ACM: Computing Curricula 2001 Computer Science, https://www.acm.org/education/curric_vols/cc2001.pdf (2001).
(19) 情報処理学会: 情報専門学科カリキュラム標準 J07, https://www.ipa.go.jp/files/000024071.pdf (2008).
(20) ACM: Computer Science Curricula 2013, http://www.acm.org/education/CS2013-final-report.pdf (2013).
(21) Micha Gorelick, Ian Ozsvald: “High Performance Python,” O’Reilly Japan Inc. (2015).
(22) Gaël Varoquaux, Emmanuelle Gouillart, and Olav Vahtras editors, “Scipy Lecture Notes - One document to learn numerics, science, and data with Python,” http://www.scipy-lectures.org/ (2015).