GNU Octave
(後編)

my recommendations on research & development tools.

鷲 沢 嘉 一†

キーワード：数値計算，数値解析，MATLAB，MEX

1. ま え が き

前編に引き続き，数値計算ソフトウェアGNU Octaveの紹介とその高速化に関するTipsについて紹介する．今回は，C言語との連携や，クラスタ環境での実行例などを中心に解説する．

2. C言語との連携

前編で解説した通り，MATLABやOctaveの反復文(forやwhile)は処理が遅い．前編では，for文やwhile文をベクトル演算に置き換えることで，高速化ができることを解説した．しかし，すべての反復文をベクトル・行列演算に置き換えられるわけではない．また，CやC++，Fortranで開発したソフトウェア資源を使用したいこともある．

MATLABでは，CおよびFortranのプログラムをMexファイルへコンパイルして呼び出すことができる．Octaveでは，C，C++，Fortranのプログラムをコンパイルし，Octfileという形式として，Octaveから呼び出すことができる．また，MATLABのMex形式で書かれたプログラムもコンパイルすることができる．OctaveでもMATLABでもMex形式のプログラムをコンパイルし，呼び出すことができるが，コンパイルしたバイナリーファイルに互換性はない．

本章では，Octave/MATLABの両方で使えるMex形式のCプログラミングについて解説を行う．

2.1 Mex

前回使用した入力ベクトル $\mathbf{a}$ に対して，

$$b_i = \begin{cases} 1 & (a_i > 0.3) \\ 1 & (a_i \leq 0.3) \end{cases} \quad (1)$$

となるようなベクトル $\mathbf{b}$ を出力するプログラムを考えよう．プログラムを図1に示す．

ファイル名をsamplemex.cとすると，コンパイルを行うためにはLinuxなどのコマンドプロンプトまたはOctaveのコマンドプロンプトから

† 独立行政法人理化学研究所 脳科学総合研究センター

"GNU Octave: Part 2" by Yoshikazu Washizawa (RIKEN Brain Science Institute, Saitama)

```
01: #include "mex.h"
02:
03: void mexFunction(int nlhs,mxArray *plhs[ ],
04:                   int nrhs,const mxArray *prhs[ ])
05: {
06:     double *a;
07:     double *b;
08:     double th;
09:     int m,n,i;
10:
11:     if ( (nlhs != 1) || (nrhs != 2) ){
12:         mexErrMsgTxt("Usage:  b = samplemex1(a, th);");
13:     }
14:
15:     m = mxGetM(prhs[0]); // number of row
16:     n = mxGetN(prhs[0]); // number of column
17:     a = mxGetPr( prhs[0] );
18:     th = *(mxGetPr( prhs[1] ));
19:
20:     mexPrintf("m=%d, n=%d th=%f\n", m,n,th);
21:
22:     plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
23:     b = mxGetPr( plhs[0] );
24:
25:     for (i=0; i<n; i++){
26:         if ( a[i] > th )
27:             b[i] = 1;
28:         else
29:             b[i] = 0;
30:     }
31: }
```

図1 samplemex.cサンプルプログラム
(説明のために行番号を付してある)

mkcofile --mex samplemex.c

と入力する．コンパイルが正常にできるとMexファイルsamplemex.mexが作成される．Octaveから呼び出すには，

> b = samplemex(a, 0.3);

のように入力する．

Octaveから呼び出される関数は，mexFunctionという名前にし，第一引数のnlhs (number of left hand sideの意味)は，Octaveから呼び出したときの戻り値の数を表す．この例の場合は，戻り値は $\mathbf{b}$ の一つであるので，nlhsの値は1である．第二引数のplhs (pointer of ... の意味)は，mxArray型の配列のポインタであり，Octaveから呼び出したときの戻り値のポインタを表す．この例ではplhs[0]

が**b**へのポインタを表している。第三引数nrhsは、Octaveから呼び出したときの引数の数であり、この例では、**a**と0.3の二つが渡されるので、nrhsの値は2である。第四引数のprhsは、Octaveから呼び出したときの引数が渡される。prhs[0]は**a**へのポインタを表し、prhs[1]は、0.3へのポインタを表す。もちろん、nlhs、plhs、nrhs、prhsの名前は、任意に設定して良い。

11行目から13行目では、引数の数をチェックし、規定の引数以外のときには、mexErrMsgTxtでエラーを出力する。15、16行目のmxGetMとmxGetNは、それぞれ行数と列数を返す関数である。17行目のmxGetPrは、mxArray型の変数からその実体の格納されているdouble配列の先頭ポインタを得るための関数である。Octaveでの行列形式は、実行列または複素実行列であり、mxGetPrは実部の配列、mxGetPiは、虚部の配列の先頭ポインタを返す関数である。18行目は、要素が一つしかないため、配列の中身を直接double型の変数thへ代入している。20行目、mexPrintfは、C言語のprintfと同様の形式で、Octaveの実行画面に表示を行う。22行目mxCreateDoubleMatrixは、 $m \times n$ の大きさの実行列を作成し、メモリの確保を行う。23行目でその先頭ポインタを**b**に代入している。25行目からは通常のC言語の文法で**b**に値を代入している。基本的にMATLABのMex形式と同じであるため、詳しくはMATLABのMexの解説を参照されたい。

3. その他の高速化

前回から高速化の手法として、

- (1) BLAS, LAPACKの入れ替え
- (2) 反復文の置き換え
- (3) Mexの利用

について解説を行った。その他の高速化に関するTipsについて、解説を行う。

3.1 メモリーの確保

For文やWhile文などの反復文において、行列やベクトルの大きさを少しずつ加えて大きくしていくと、仮想メモリー空間におけるメモリーアドレスが大きく分断され、メモリーアクセスやSIMD命令などが効率的に働かなくなる。下の例のように、ベクトル**a**の大きさが反復によって大きくなっていく場合、実行速度の低下を招く。

```
for ii = 1:100
    a(ii) = log(ii);
end
```

これを防ぐには、あらかじめzerosなどを用いてメモリー領域を確保しておけばよい。

```
a = zeros(1,100);
for ii = 1:100
    a(ii) = log(ii);
end
```

3.2 行列積の順序

数値計算における、基本的な注意点であるが、行列積の順序を入れ替えることで、速度が大きく変わることがある。明示的に指定しないとき、Octaveは左から順番に積を行う。下の場合には、**a * b**が先に計算され、そのあとで*** c**の部分が計算される。

a * b * c

a, b, cの行数、列数をそれぞれ、 $a_m, a_n, b_m, b_n, c_m, c_n$ とする ($a_n = b_m, b_n = c_m$)。 **a * b**の計算には、 $a_m a_n b_n$ 回、(その結果) *** c**の計算には $a_m b_n c_n$ 回の乗算が必要であるため、合計で $a_m b_n (a_n + c_n)$ 回の乗算が必要である。一方、明示的に **b * c**の演算を先に計算する場合、

a * (b * c)

乗算回数は、 $a_n c_n (a_m + b_n)$ 回である。

どちらの場合も演算結果は同じであるが、計算量は異なる。特に、**a, b**が行列で**c**がベクトルになるような演算の場合には、後者の計算量が小さくなる。手元のPCでは、**a, b**が3000×3000の実行列で**c**が3000次元の実ベクトルの場合、**a * b * c**で、4.1秒(乗算回数 2.7×10^{10} 回)、**a * (b * c)**では、0.015秒(乗算回数 1.8×10^7 回)と、270倍も実行時間の差が生じた。

3.3 逆行列

Octaveでは、逆行列の計算はinv()で計算ができるが、この手法は、数値計算の観点からよいとは言えない。詳しい議論は、数値計算の教科書、例えば、文献1)の第5章「逆行列よさようなら」を参照されたい。行列**A**の逆行列の計算方法として、以下の方法を考えてみる。ここでは、行列(あるいはベクトル)**B**に対して、最終的に $C = A^{-1}B$ を計算することを考える。

(1) $C = \text{inv}(A) * B$ を用いる。

(2) $C = A \setminus B$ を用いる。

(3) LU分解 (Aが対称行列の場合はCholesky分解) を用いる。

(3)の手法は、行列Aに対して、LU分解 ($[L \ U \ P] = \text{lu}(A)$) を行うと、

$$A = P^T L U \quad (2)$$

となる置換行列**P**、下三角行列**L**、上三角行列**U**が得られる。これより、 $A^{-1}B = U^{-1}L^{-1}PB$ が得られる。置換行列の積と、上下三角行列の逆は簡単に計算ができる。Octaveのコードは、

```
[L D P] = lu(A);
C = U \ (L \ (P * B));
```

のようになる。Aが対称行列の場合には、Cholesky分解を用いて

```
R = chol(A);
C = R \ (R \ B);
```

とする。(1)の方法は、 $A^{-1} = U^{-1}L^{-1}P$ を始めに計算することと同値である。

まず、実行時間を比較する。今、Aを3000×3000の実行列、Bを3000×100の実行列としよう。手元のPCでの実行時間は

- (1) inv(A)の計算に4.7秒、行列積に0.15秒
- (2) 1.8秒
- (3) LU分解に2.1秒、U\L\ (P*B))に0.15秒

であり、(2)が最も速く、(1)が最も遅かった。Aが対称行列の場合には、

- (1) inv(A)の計算に3.5秒、行列積に0.15秒
- (2) 1.4秒
- (3) Cholesky分解に1.0秒、R\'(R\'B)に0.15秒

であり、(3)が最も高速であった。

(2)の場合には、逆行列を掛ける行列Bが他にもある場合、例えば、 $A^{-1}B_1$ と $A^{-1}B_2$ の両方を計算するような場合に、(1)、(3)に比べて計算時間が掛かる。 B_1 、 B_2 があらかじめわかっているならば、 $C=A\backslash[B_1\ B_2]$ のようにしてからCを分割すればよいが、あらかじめわからない場合には(3)の方法の方が高速な場合もある。

次に精度を比較する。2乗誤差を

$$e = \|AC - B\|_F^2 \quad (3)$$

と定義する。ここで、 $\|\cdot\|_F$ はFrobeniusノルムを表す。(1)、(2)、(3)の2乗誤差をそれぞれ、 e_1 、 e_2 、 e_3 とする。問題によって誤差の絶対量が変わるため、 e_2 を基準として e_1/e_2 と e_3/e_2 を比較する。先ほどと同様に、Aを3000×3000、Bを3000×100として、一様乱数で作成し、10回の誤差の平均と標準偏差を求めた。 e_1/e_2 の値は、 15.3 ± 3.7 、 e_3/e_2 の値は 1 ± 0 であった。すなわち、 e_2 と e_3 の値は一致し、 e_1 の誤差はその15倍程度であった。

以上の結果より、(1)の手法はコードはシンプルであるが、計算時間、精度の観点から効率の良い方法であるとは言えない。これは、Octaveに限らず、数値計算一般に言えることである。ただし、Octaveでは、 n^2 次正方行列の上下三角行列の場合も、 n^2 のメモリーを消費してしまう。このため、消費メモリーの観点では(3)は(1)に劣る。

4. 計算資源の最適化

4.1 引数の取り込みとコマンドラインからの実行

前編では、OpenMPなどの共有メモリー型並列処理に対応したBLASに入れ替えることにより、Octaveを高速化できることを紹介した。行列積や連立方程式の解法など並列化しやすい演算を含むプログラムでは、簡単に並列化による高速化が実現できる。

別の高速化の方向性として、異なるデータやパラメータ、初期値、乱数種などを用いて、同じプログラムを何度も実行するような状況において、同じプログラムを同時に実行させ、CPUの各コアにプロセスを割り振り、実行時間の最適化を図ることもできる。実際に、こちらの並列化の方が、簡単に実現でき、さらに効率が良いことが多い。一つのプ

ログラムを並列化すると、どうしてもオーバーヘッドに時間が掛かり、1スレッドのプログラムを2スレッド並列にしても、実行時間は1/2になることは少ない。一方、メモリー帯域が充分であれば、複数コアを持つCPUで、同じプログラムを二つ同時に走らせても実行時間はほとんど変わらない。後者の方が実行効率は高い。この最適化は、Octaveをコマンドラインから実行することで容易に実現できる。本節では、Linuxシステムを前提とし、この高速化に関する解説を行う。

手動でOctaveをいくつも立ち上げ、それぞれのコマンドプロンプトに、異なるデータファイルやパラメータを指定して実行の終了を待つという方法もある。しかし、ジョブが数十、数百を超えると面倒になってくる。利用できるコア数が実行したいジョブの数よりも少ない場合には、同時実行数がコア数を下回ってから次のジョブを投入しなければ、リソースの競合が発生し、処理が遅くなる。

そこで、このようなジョブの投入を自動的に行うことを考えよう。同じプログラムで異なるデータファイルやパラメータを扱うときには、プログラムに引数を与えて起動し、引数にデータファイルやパラメータの値を指定すればよい。

MATLABにはないOctaveの特長の一つに、引数処理がある。Octaveを起動するときに、引数に実行するスクリプトファイルを指定し、その後に任意の引数を続けると、スクリプトファイル内からはnarginで引数の数、argvでその引数を取り込むことができる。引数を取り込むサンプルプログラムを図2に示す。argvで返される変数はセル変数と呼ばれる変数であり、中括弧{}でインデックスを囲って内部を参照する。この場合、セル変数の各要素は文字列であるため、数値への変換にはstr2num、数値かどうかのチェックにはisnumericを使えばよい。Linuxコマンドラインからの実行結果は、

```
% octave -q hikiuu.m abc def
```

```
Hikiuu 1: abc
```

```
Hikiuu 2: def
```

となる。オプション-qは、起動メッセージを表示しないオプションである。また、スクリプトファイルの先頭にフルパスのOctave

```
#!/usr/bin/octave -q
```

のように書いて、ファイルに実行権限を与えれば、Linuxの実行ファイルとして実行できる。

```
./hikiuu.m abc def
```

4.2 複数プロセスの実行

複数のOctaveプロセスを同時に立ち上げるためには次

```
hiki = argv;
for ii=1:nargin
    fprintf('Hikiuu %d: %s\n', ii, hiki(ii));
end
```

図2 引数取り込み (hikiuu.m)

```
#!/bin/sh

ncore=4 # コア数
export OMP_NUM_THREADS=1 # 1 ジョブ当たりコア数
for param in `seq 1 10`
do
    octave -q script.m $param > log.$param &
    # ジョブ数がコア数以上の場合、5秒待つ
    while [ `jobs -p | wc -l` -ge $ncore ]
    do
        sleep 5
    done
done
```

図3 ジョブ投入シェルスクリプト

の方法がある。

- (1) ジョブスケジューラを用いる
- (2) シェルスクリプトを用いる
- (3) atd+batchコマンドを用いる

ジョブスケジューラは初期設定が面倒であるが、一度設定してしまえば、ノードPCが複数ある時に効率的にジョブを実行することができる。各機関のスーパーコンピュータや大規模並列計算機などには、あらかじめジョブスケジューラが用意されている場合が多い。筆者はPBS (Portable Batch System) ベースのTorqueを使用している。

簡単なシェルスクリプトからジョブを投入することもできる。単一のPC上で動作させる例を図3に示す。chmodで実行権限を付して実行すれば、paramに1から10までの値が入りながら、最大4プロセスまで同時に実行される。簡単なシェルスクリプトなので、ncoreの値やparamの範囲を変えるだけで、カスタマイズができる。

シェルスクリプトでも、他のPCにジョブを投入することが可能である。そのためには、パスフレーズなしのsshログインを設定し、

```
ssh hostname octave -q /path/script.m var
```

のようにすればよい。ただし、ファイルサーバなどを利用して、実行ノードからもスクリプトファイル/path/script.mがアクセスできるようにしておく必要がある。さらに、ジョブ数の管理も自分で行う必要がある。例えば、4コアを持つプロセッサ上で、シングルスレッドのジョブを5つ以上同時に走らせると、リソースの競合が発生し、実行速度は遅くなる。順番に4並列で実行させることが望ましい。sshによる実行とシェルスクリプトを組合せれば、他のPCにジョブを投入できるように改良できる。

batchコマンドは、通常Linuxにインストールされており、簡単なジョブスケジューラとして用いることができる。また、大学や研究所の中規模計算機のジョブ管理にもよく用いられており、例えば、講義や演習のない夜間や休日に研究用のジョブを走らせるときに、batchコマンドとatd

(atコマンドを制御するdaemon)によるシステムがよく用いられる。シェルスクリプトによるジョブの投入は

```
batch -f script.sh
```

のように行う。直接Octaveのジョブを投入するには、

```
echo "octave -q script.m" | batch
```

のように行う。

デフォルトの設定では、CPU負荷率が0.8以下のとき(1.5の場合もある)に投入されたジョブを実行する。最近のディストリビューションでは、atdデーモンが立ち上がっていないことが多いので、サービスの設定でatdを立ち上げる必要がある。また、設定ファイルは、opensuseの場合は、/etc/sysconfig/atdにあり、このファイルを書換えることで、CPU負荷率の閾値などを調整することができる。あるいは、コマンドatrunのオプション-lで閾値を変更できる場合もある。詳しくは、それぞれのディストリビューションのドキュメントやmanページを参照されたい。4コアを持つCPUならば、閾値3.5程度にしておけば、batchコマンドで投入されたジョブが4並列で順次実行される。実行キューの確認はatq、ジョブの削除はatrmで行うことができる。

5. む す び

数値計算ソフトのOctaveとその高速化に関するTipsについて解説を行った。行列積の順序や逆行列の話題などは、数値計算において基礎的な事項であるが、見落としがちであり、初学者にとっても重要な事項であるため、解説を行った。

Windows環境でのジョブスケジューラは、HPC Serverなどに含まれている。本稿では触れなかったが、ドキュメントなどを参照されたい。Windows OSの場合は、無料で使用できるBLASが限られていたり、コンパイルが難しいことが多い。計算用のサーバやPCのOSがWindowsであるときも、Virtual PCやVMwareなどでLinux環境を構築し、仮想PCのLinux上で最適化されたBLASを使用した方が高速な場合も多い。本稿で紹介したスケジューラやシェルスクリプトも使用できるため、筆者はWindows OSを利用する場合でも、仮想Linux上でOctaveを利用することを勧める。

(2011年3月25日受付)

〔文 献〕

- 1) 伊理正夫、藤野和建：“数値計算の常識”，共立出版（1985）

鷺沢 嘉一 2008年、東京工業大学大学院博士課程修了。現在、理化学研究所脳科学総合研究センター研究員。機械学習、パターン識別、脳信号処理などの研究に従事する。工学博士。