他の言語と比較したMATLABの特徴

2019/12/26 T. Hirano

1. はじめに

プログラミング言語は変数および代入操作、論理・算術演算、繰り返し（ループ）、条件分岐があればどの言語でも同じ機能が実現できる。ただし、内部での実装方法により、計算時間やメモリ使用効率が異なるので、特に大規模な問題を扱う場合は内部実装が気になることがある。開発者でないと内部の実装まではわからないから仕方ないのだが（実際に、販売代理店の方に質問してもわからないという回答である）、本稿では、外部から確認できるものをできる限り確認することを目的とする。

2. MATLABの特徴

MATLABはMatrix Laboratoryの略で、行列計算を多用する科学技術計算用のインタープリタ言語である。インタープリタとは言っても、行列方程式を解く、固有値問題を解くなどのルーチンは、入力データを受け渡したらひたすら実行形式のルーチン内で処理をし、長い時間ルーチン内での処理になるので、このような用途ではコンパイル方式に対するインタープリタ方式の遅さは問題にならない。MATLABが目指したのは、FORTRANでLINPACKの行列ルーチンを呼び出すときの手間を省くためであったと言われる。数学・計算用の言語でも、Mathematicaは記号数式処理で可能な限り厳密な計算をすることが目的であり、数値計算はそのおまけのようなものであり、ターゲットが異なっている。実際、数値計算はMATLABが圧倒的に速い。MATLABでも記号数式処理ができるが(Symbolic Math Toolbox)、記号数式処理の能力はMathematicaの方が圧倒的に高い。

3. 演算について

演算はすべて行列を想定している（スカラーは1 x 1の行列）。（FORTRANではcomplex型があり、複素数は自然に使えたが、それがさらに行列にまで拡張された形である）

```
>> a = [1+1i*2 2+1i*3; 3 4]
a =
   1.0000 + 2.0000i   2.0000 + 3.0000i
   3.0000 + 0.0000i   4.0000 + 0.0000i
>> b= [5; 6]
b =
     5
     6
>> a*b
ans =
```

```
   17.0000 +28.0000i
   39.0000 + 0.0000i
```

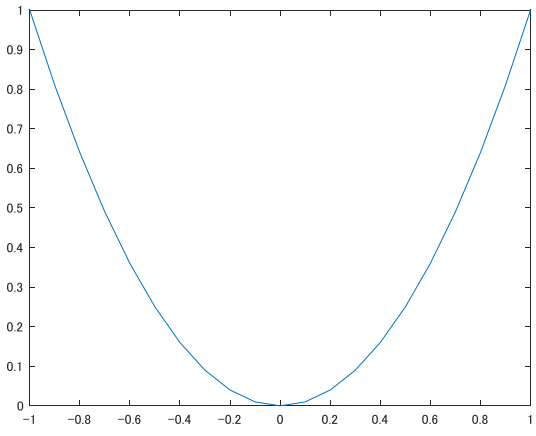
行列・ベクトル演算、またメモリ保存形式を利用してハードウェアでも可能な限り並列計算（ベクトル演算）を利用している。例えば、次のグラフの描画(plot コマンドなど)もその一例であり、最初に MATLAB を使うユーザーはこの仕組みが理解しにくいと思われる（筆者もそうであった）。

4. グラフの描画(polt コマンド)について

数式$y = x^2$のグラフ$-1 \leq x \leq 1$の範囲で描く。

```
>> x=-1:0.1:1
x =
  1 列から 13 列
   -1.0000   -0.9000   -0.8000   -0.7000   -0.6000   -0.5000   -0.4000   -0.3000   -0.2000   -0.1000         0
0.1000    0.2000
  14 列から 21 列
    0.3000    0.4000    0.5000    0.6000    0.7000    0.8000    0.9000    1.0000
>> y=x.^2
y =
  1 列から 13 列
    1.0000    0.8100    0.6400    0.4900    0.3600    0.2500    0.1600    0.0900    0.0400    0.0100
0    0.0100    0.0400
  14 列から 21 列
    0.0900    0.1600    0.2500    0.3600    0.4900    0.6400    0.8100    1.0000
>> plot(x,y)
```

上のコードの説明をする。「x=-1:0.1:1」で-1 から始まり、1（を越えない）まで 0.1 ずつ増やした行ベクトルを生成している。次の「y=x.^2」では、前に生成した行ベクトルの各成分を 2 乗している。ここで、「y=x^2」と書かずに、「y=x.^2」としているのは、行列演算ではなく『各成分を』2 乗するからである。実際、行ベクトル x の次元は 1×21 なので、x^2 は 1×21 の行列の左から 1×21 の行列をかける演算となり、間の列と行の次元が合わないので行列の積は実行不可能である。




```
>> x^2
エラー:  ^  (line 51)
行列をべき乗にするには次元が正しくありません。行列が正方行列で、べき指数がスカラーであることを確認してください。行列を要素ごとにべき乗するには、'.^' を使用してください。
```

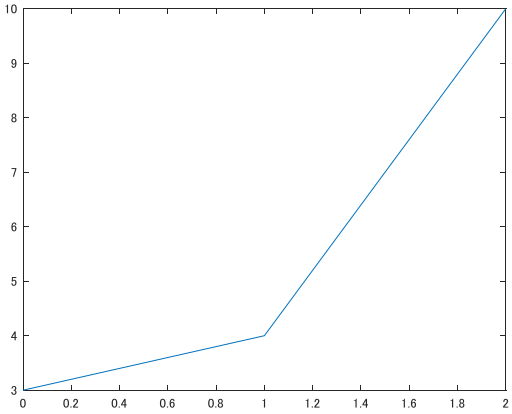
同様にベクトルの乗算*は.*で要素ごとの乗算、除算は行列では定義されていないが（あえて言うなら逆行列にその意味を持たせることができるが）./で要素ごとの乗算の意味にできる。他にも

```
>> x=[1 2 3];
>> sin(x)./x
ans =
    0.8415    0.4546    0.0470
```

など、少々複雑な表現があるが、あくまで上の表現では sin は x の各要素に適用され sin(x)は行ベクトルになっている。そして、./x によって、要素ごとに割り算が実行されることになる。もっと複雑な関数になっても同じであり、各要素の計算は同じ計算を違う値にたいして行うだけの構造であり、後述の SIMD による処理が向いているのである。

最後に、「plot(x,y)」の plot コマンドは、次元の一致した行ベクトル、または列ベクトルの x,y のそれぞれの要素を x,y 座標としてつないだ線を描くコマンドである。

```
>> plot([0 1 2], [3 4 10])
>> plot([0; 1; 2], [3; 4; 10])
>> plot([0; 1; 2], [3 4 10])
>> plot([0 1 2], [3; 4; 10])
```

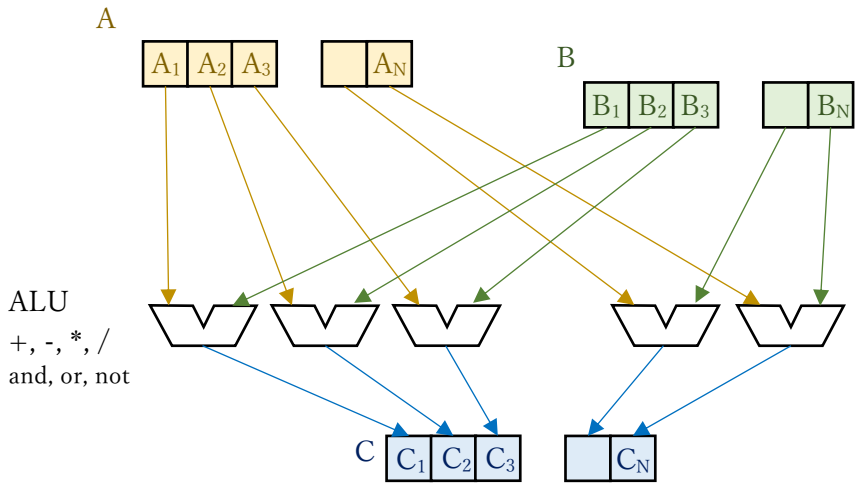


上のコマンドを実行するとすべて右図のようなグラフが描かれる。x, y は同じ行ベクトル同士、または列ベクトル同士である必要はなく、次元が一致した行または列ベクトルならばいいようである。

Mathematica などを利用していると、グラフを描くのにまず x,y のベクトルを準備しなければならないのが面倒に思えるが、次で説明するように、計算の効率性のためであると思われる。

5. ベクトル演算について

前説の「効率」について説明を続ける。ベクトル演算（入力データが異なるだけで、並列に同じ演算を行う）が可能だからである。ベクトル演算は 1980～1990 年代のスーパーコンピュータの中心技術であった。仕組みとしては CPU 内に被演算データを保存するレジスタが複数あり、各レジスタの内容に対して演算を行う ALU(Arithmetic Logic Unit)が複数あり、同時に違うデータに対する同じ演算を出力するものである。（下図のイメージ）



現在ではポリゴンの座標変換などを目的として、ベクトル演算を行う GPU (Graphics Processing Unit) がほぼ全てのコンピュータに搭載されており、余力を一般のベクトル演算用に用いることができる。一般のベクトル計算も行える GPU を GPGPU(General-Purpose Computing on Graphics Processing Units) と言う。また、CPU でもベクトル演算を行うための SIMD(Single Instruction, Multiple Data)命令をサポートするようになり、並列計算ができるようになっている。

さて、MATLAB が内部でどのように処理しているかはわからないが、次のコードでテストしてみる。

```
>> n=1000;
a=rand(n);
b=rand(n);
tic; c1=a*b; toc
経過時間は 0.018958 秒です。

>> c2=zeros(n,n);
tic;
for i=1:n
  for j=1:n
    sum=0;
    for k=1:n
      sum=sum+a(i,k)*b(k,j);
    end
    c2(i,j)=sum;
  end
end
toc;
経過時間は 3.551649 秒です。
```

上のプログラムは行列 a と b のかけ算を行うプログラムである。c1 はベクトル演算で計算されている(実

際にどのようにしているのか、つまり GPU を使っているのかなどはわからないが・・)。c2 は普通に for ループを使って計算（スカラー演算）している。計算時間はベクトル演算の方が 187 倍も速いことがわかる。

■ベクトル演算での積和演算の速度(FLOPS)を計算すると、

積和演算の回数（つまり浮動小数点演算 2 回）：1000^2

時間: 0.019 sec (GNU Octave で 0.016 sec)

速度: $(1000^2/0.019)/10^9 = 0.053$ GFLOPS

■スカラー演算(for ループ)での積和演算の速度(FLOPS)を計算すると、

積和演算の回数（つまり浮動小数点演算 2 回）：1000^2

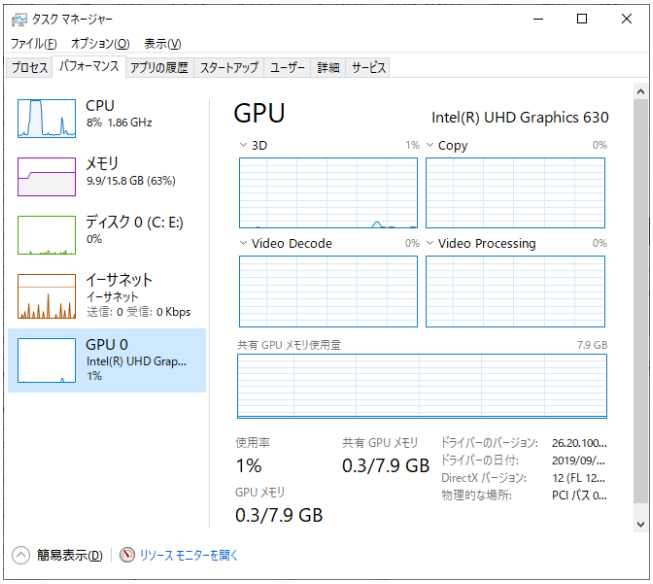
時間: 3.55 sec (GNU Octave ではなかなか終わらない。n=100 でやっと 9.7 sec)

速度: $(1000^2/3.55)/10^9 = 0.000282$ GFLOPS

となっている。余談だが、スカラー演算(for ループ)において、一番外側の i でのループを自動並列化する parfor(並列 for ループ)に置き換えると（単に for を parfor に書き換えるだけ)、計算時間は 1.5 sec 程度と高速化される。parfor は入れ子にできないので、一番外側に使った。内側(j,k)で使うと依存関係のせいでずっと遅くなる。

使用マシンは Dell Inspiron 3470 である。

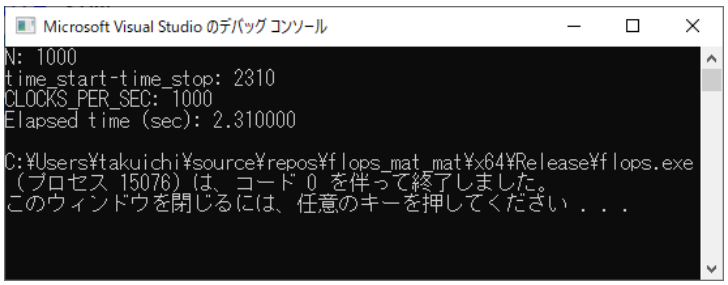
ベクトル演算（？）の速さは何によって実現されているのか、GPU は使われているのか気になるので、タスクマネージャで確認したが、使用しているのは GPU ではなく、CPU であることがわかった。CPU の SIMD 命令で実現されているものと思われる。



C 言語との速度の違いが気になるので、同じマシンを使って C 言語(Visual Studio 2017)で次のコードを書いて比較した。

```
#include "pch.h"
#include <stdio.h>
#include <stdlib.h>  // malloc, free, rand関数用
#include <time.h> // 時間計測clock()用

#define N 1000

int main()
{
	int i,j,k;
	//double a[N][N], b[N][N], c[N][N];  // この確保はN=1000だとできなかった。
	double **a, **b, **c;
	double sum;
	clock_t time_start, time_stop;

	printf("N: %d¥n", N);

	// メモリ確保
	a = (double **)malloc(N * sizeof(double *));
	b = (double **)malloc(N * sizeof(double *));
	c = (double **)malloc(N * sizeof(double *));
	for (i = 0; i < N; i++) {
		a[i] = (double *)malloc(N * sizeof(double));
		b[i] = (double *)malloc(N * sizeof(double));
		c[i] = (double *)malloc(N * sizeof(double));
	}
	// 初期値代入
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++) {
			a[i][j] = rand() / (double)RAND_MAX;
			b[i][j] = rand() / (double)RAND_MAX;
		}
	}
```

```
        time_start = clock();       // 時間計測開始時刻

        // 行列の積の計算
        for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                        sum = 0.0;
                        for (k = 0; k < N; k++) {
                                sum += a[i][k]*b[k][j];
                        }
                        c[i][j] = sum;
                }
        }
        time_stop = clock();        // 時間計測終了時刻

        printf("time_start-time_stop: %d¥n", time_stop - time_start);
        printf("CLOCKS_PER_SEC: %d¥n", CLOCKS_PER_SEC);
        printf("Elapsed time (sec): %f¥n", (time_stop - time_start) / (double)CLOCKS_PER_SEC);

        // メモリ開放
        for (i = 0; i < N; i++) {
                free(c[i]);
                free(b[i]);
                free(a[i]);
        }
        free(c);
        free(b);
        free(a);
}
```



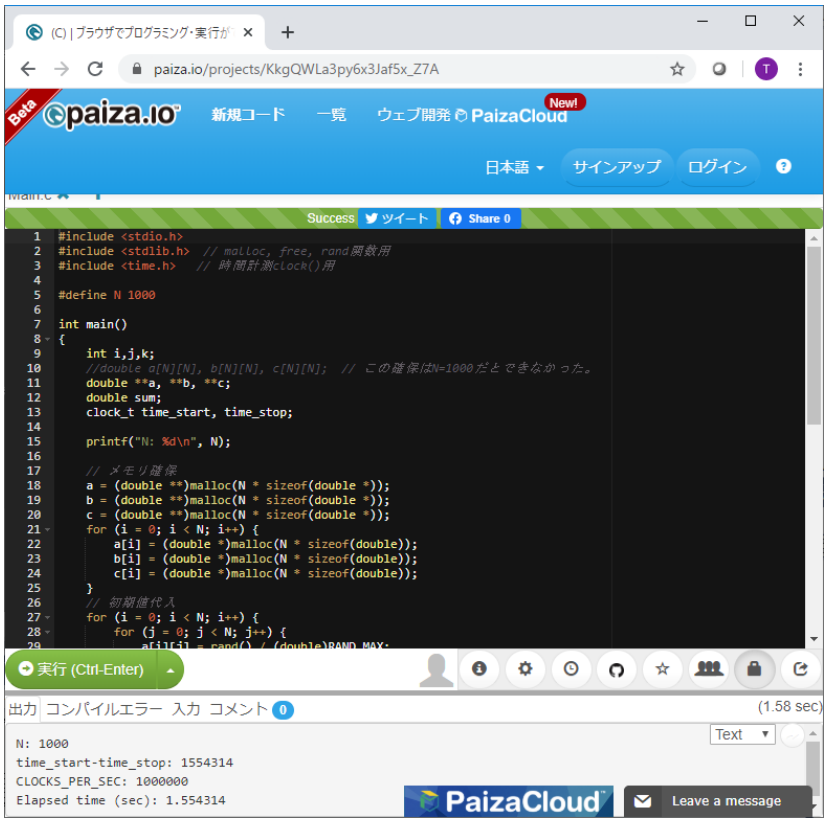
結果は

積和演算の回数（つまり浮動小数点演算 2 回）：1000^2

時間: 2.31 sec

速　度　：　$(1000^2/2.31)/10^9 =$ 0.000433 GFLOPS

となった。MATLAB の速度は C 言語の 65%とかなり速いことがわかる。上の C 言語のコードを WEB 上のプログラミング環境(paiza.io)で実行すると、時間は 1.55 sec であった。このマシンの Visual Studio 2017 の 150%の速度が出ている。



さらに驚くべきことに、MATLAB のベクトル演算を用いると C 言語のコードと比較しても 122 倍も速いことがわかる。必ずしも C 言語で書い

たほうが速いとは言えないことがわかる。(C 言語で書いた方が速いという意味は、ベクトル演算も明示的に書いた場合ということになる)

6. 行列（配列）の行と列の入れ替えなどについて

大規模な問題では、行列（配列）の格納効率に気を配らなければならない。C 言語ではポインタを活用して行と列を入れ替えるなどの操作は効率よく実装する手段がある。MATLAB ではどうなっているのか、内部の実装は見えないが、外部から whos で使用メモリを調べて確認する。

```
>> whos;              % 各変数の消費メモリを表示
>> memory;            % MATLAB アプリの Windows での使用メモリを表示(Windows 限定)
利用可能な最大配列:          10583 MB (1.110e+10 bytes) *
すべての配列に利用可能なメモリ:          10583 MB (1.110e+10 bytes) *
MATLAB で使用されるメモリ:          1681 MB (1.763e+09 bytes)
物理メモリ（RAM):          16208 MB (1.700e+10 bytes)

*  利用可能なシステムメモリ（物理 + スワップ ファイル）により制限されます。
```

```
>> n=11200;
>> a=rand(n);        % 乱数が入った n x n の行列を生成
>> whos; memory;
  Name             Size                      Bytes  Class     Attributes

  a            11200x11200               1003520000  double
  n                1x1                          8  double

利用可能な最大配列:          9650 MB (1.012e+10 bytes) *
すべての配列に利用可能なメモリ:          9650 MB (1.012e+10 bytes) *
MATLAB で使用されるメモリ:          2631 MB (2.759e+09 bytes)
物理メモリ（RAM):          16208 MB (1.700e+10 bytes)
```

double 型は 8Byte のメモリを使用し、行列 a は約 1GB のメモリを使用することがわかる。実際に「MATLAB で使用されるメモリ」は約 1GB 増加した。

```
>> b=rand(n);
>> whos; memory;
  Name             Size                      Bytes  Class     Attributes

  a            11200x11200               1003520000  double
  b            11200x11200               1003520000  double
  n                1x1                          8  double

利用可能な最大配列:          8672 MB (9.094e+09 bytes) *
すべての配列に利用可能なメモリ:          8672 MB (9.094e+09 bytes) *
```

```
MATLAB で使用されるメモリ:          3586 MB (3.760e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

さらに、同サイズの行列 b を定義すると、さらに使用メモリは 1GB 増加した。

```
>> c=b;
>> whos; memory;
  Name            Size                      Bytes  Class     Attributes

  a           11200x11200             1003520000  double
  b           11200x11200             1003520000  double
  c           11200x11200             1003520000  double
  n               1x1                         8  double

利用可能な最大配列:          8684 MB (9.106e+09 bytes) *
すべての配列に利用可能なメモリ:          8684 MB (9.106e+09 bytes) *
MATLAB で使用されるメモリ:          3585 MB (3.759e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

行列 b を行列 c にコピーした。この場合使用メモリは増えない。C 言語で言うと、実体の内容は今のところ同じなので、ポインタで c は b の内容を指し示すように表現していると思われる。また、使用メモリは若干減っているが、ガーベッジコレクションを行っているものと思われる。

```
>> c(1,2)=2;
>> whos; memory;
  Name            Size                      Bytes  Class     Attributes

  a           11200x11200             1003520000  double
  b           11200x11200             1003520000  double
  c           11200x11200             1003520000  double
  n               1x1                         8  double

利用可能な最大配列:          7566 MB (7.934e+09 bytes) *
すべての配列に利用可能なメモリ:          7566 MB (7.934e+09 bytes) *
MATLAB で使用されるメモリ:          4550 MB (4.771e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

先ほど生成した c の 1 行 2 列に 2 の値を代入。使用メモリは約 1GB 増加した。たった 1 要素ではあるが、b と c は異なるので、全体をコピーして c の内容も実体として生成されたようである。

```
>> a(1,:)=[];          % 行列 a の第 1 行を削除する
>> whos; memory;
  Name            Size                      Bytes  Class     Attributes

  a           11199x11200             1003430400  double
  b           11200x11200             1003520000  double
```

```
  c              11200x11200                1003520000  double
  n                  1x1                           8  double

利用可能な最大配列:          7596 MB (7.965e+09 bytes) *
すべての配列に利用可能なメモリ:          7596 MB (7.965e+09 bytes) *
MATLAB で使用されるメモリ:          4548 MB (4.769e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

行列 a の第 1 行を削除した。大規模な問題では、行列を新しく生成してコピーすることなく、削除したことにしたい。C 言語ではポインタの変更(NULL にするなど)で対応可能である。MATLAB でも使用メモリは増加していないので、同じ変数に再度代入しているような表現ではあるが、効率的に行や列の削除を行えるような内部表現で実装されているようである。

```
>> d=a(:,[1:n-1]);  % 行列 d に a の第 n 列を削除した値を代入
>> whos; memory;
  Name            Size                       Bytes  Class     Attributes

  a          11199x11200                1003430400  double
  b          11200x11200                1003520000  double
  c          11200x11200                1003520000  double
  d          11199x11199                1003340808  double
  n                  1x1                           8  double

利用可能な最大配列:          6248 MB (6.551e+09 bytes) *
すべての配列に利用可能なメモリ:          6248 MB (6.551e+09 bytes) *
MATLAB で使用されるメモリ:          5514 MB (5.782e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

d に行列 a の第 n 列を削除した値を代入した。使用メモリは約 1GB 増加するので、新たに実体を生成しているのがわかる。

```
>> nr1=2; nr2=10;
>> b([nr1 nr2],:)=b([nr2 nr1],:);          % 行列 b の第 nr1 行と第 nr2 行を入れ替える
>> whos; memory;
  Name            Size                       Bytes  Class     Attributes

  a          11199x11200                1003430400  double
  b          11200x11200                1003520000  double
  c          11200x11200                1003520000  double
  d          11199x11199                1003340808  double
  n                  1x1                           8  double
  nr1                1x1                           8  double
  nr2                1x1                           8  double

利用可能な最大配列:          6270 MB (6.574e+09 bytes) *
すべての配列に利用可能なメモリ:          6270 MB (6.574e+09 bytes) *
```

```
MATLAB で使用されるメモリ:          5489 MB (5.755e+09 bytes)
物理メモリ (RAM):          16208 MB (1.700e+10 bytes)
```

行列 b の第 nr1 行と第 nr2 行を入れ替えた。この表現「b([nr1 nr2],:)=b([nr2 nr1],:)」でも再代入しているのかどうかわからないが、使用メモリを見ると増えていないので、C 言語でいうポインタの変更、つまり内部の参照の変更で済ませ、効率的に表現しているようである。