第 4 章

C プログラミングと HPC の基礎

本章からコンピュータ言語を使用してプログラミングの基本を学んでいく。但し，最低限のプログラミング経験はあるものとし，必要となる言語要素だけを具体例から学び取って欲しい。

ここでは C 言語を用いたプログラミングの方法と，行列計算を用いたベンチマークテストを通じて，HPC(High Performance Computation, 高性能計算) の基礎を学ぶ。最初に述べたように，コンピューター資源には限りがあり，OS はそれを適切に管理するために不可欠である。そのため，コンピューターの能力を最大限発揮させるためには，なるべく OS 等のソフトウェアを介さず，CPU 等のハードウェアを直接操作することが望ましい。従って，アセンブラ言語 (機械語をほぼ 1 対 1 でテキスト化したもの) でプログラムを書くことが出来ればそれに越したことはない。しかしそれは大規模化するソフトウェアの世界にあっては限定的なものに留めるべきで，普通は，無駄の少ない機械語に変換できるコンパイラ言語を用いて全体の開発を行う。UNIX の世界では OS 開発用に作られた C 言語が標準的に使われており，現在でも科学技術計算の世界でも，ハードウェアに組み込まれたソフトウェアにも，広く使われている。是非ともこの C 言語によるプログラミング (C プログラミング) を通じて，コンピューターの性能を測るためのベンチマークテストの方法を学び，コンピューター資源の有限性を感じ取って欲しい。

4.1 C プログラムの初歩

2 次方程式を解く C プログラムを書いてみよう。

実数 a, b, c を定数として与えた時，

$$ax^2 + bx + c = 0$$

の解は，$a \neq 0$ であれば，解の公式を用いて表現することが出来る。判別式

$$d = b^2 - 4ac$$

の値を計算しておけば，

$d \geq 0$ の場合: 実数解 x_1, x_2 が次のようにして得られる。

$$x_1 = \frac{-b + \sqrt{d}}{2a}$$
$$x_2 = \frac{-b - \sqrt{d}}{2a}$$

$d < 0$ の場合:　複素数解 x_1, x_2 が次のようにして得られる。

$$x_1 = \frac{-b}{2a} + \frac{\sqrt{-d}}{2a} \cdot \sqrt{-1}$$
$$x_2 = \frac{-b}{2a} - \frac{\sqrt{-d}}{2a} \cdot \sqrt{-1}$$

となることは高校で習ってきた筈である。

従って，このプログラムの流れは次のようになる。

1. a, b, c の値をキーボードから入力して取り込む。
2. 判別式 d を計算し，その値の正負を判断して実数解，複素数解のどちらかを計算する。
3. 計算された解 x_1, x_2 を画面に表示する。

これらの処理を順次実装していく。C プログラムのファイル名は “quadratic_eq.c”とする。

4.1.1　C プログラムのコンパイル方法

C 言語に限らず，プログラムは普通，テキストファイルとして生成し，それをソフトウェアを解してコンピューターに実行させる。この大本のテキストファイルをソースプログラム (source program)，略してソースとも呼ぶ。C 言語のソースプログラムをここでは C ソースと呼ぶことにする。

2 次方程式を解く C ソース “qudratic_eq.c”のうち，まず係数 a, b, c を入力して表示するだけの部分を書いてみると，次のようになる。

```
 1: // 2次方程式を解くプログラム: quadratic_eq.c
 2: #include <stdio.h>
 3:
 4: int main()
 5: {
 6:     // a * x^2 + b * x + c = 0
 7:     double a, b, c;
 8:
 9:     // input coefs
10:     printf("a * x^2 + b * x + c = 0¥n");
11:     printf("a = "); scanf("%lf", &a);
12:     printf("b = "); scanf("%lf", &b);
13:     printf("c = "); scanf("%lf", &c);
14:
15:     printf("Solve %f * x^2 + %f * x + %f = 0...¥n", a, b, c);
16:
17:     return 0;
18: }
```

行頭の行番号 “14: ”は説明のために便宜上書いてあるだけなので，C ソースには入力しないこと！

このソースは各行で次のような処理を行っている。

1, 6, 9 行目:　“//”で始まる文字列は無視される。ここにコメントを書いておき，後日の自分用メモとして活用する。

2 行目: ヘッダファイル (header file) の読み込み。必要なライブラリ (機能) を呼び出すために必要な宣言が “*.h”というファイル (ヘッダファイル) に書き込まれている。ここでは標準入出力 (printf, scanf 関数) を使うために必要な宣言が記入されている “stdio.h”を読み込んでいる。

4, 5 行目・18 行目: main 関数に書かれている内容が上から順に実行される。関数の記述方法は後述するが，その中身は中括弧 “{” ... “}”内部に記入しておく必要がある。一つの処理をセミコロン “;”で区切って書く。

7 行目: 倍精度実数型 (double 型) の変数として a, b, c を宣言し，格納に必要なメモリ領域を確保しておく。

10 行目: これから入力する値が 2 次方程式の係数であることを出力 (画面に表示) している。“¥n”は改行を意味する。printf 関数で標準出力先 (画面) に"..."で囲まれた文字列を書式指定に基づいて表示を行う。

11〜13 行目: 入力する変数名を出力し，その後で scanf 関数で入力された値を実数型 (“%lf”と指定) として受け取り，変数に格納している。セミコロンで区切れば 1 行に複数の処理を書くことが出来る。

15 行目: 入力された値を 10 進小数として標準出力先に表示し，確認を行う。

このコンパイルする際には cc あるいは gcc コマンドを用いる。エラーがなければ実行可能ファイル”a.out”が生成されるはずである。

```
$ gcc quadratic_eq.c ←gccコンパイラでコンパイル
・・・(エラーがなければ何も表示されない)
$ ls -ld a.out ←実行可能フラグが立っていることを確認せよ！
-rwxr-xr-x   1 tkouya   cs          5124  5月 10日  17:04 a.out
```

カレントディレクトリにはパスが通っていない (コマンドを探索するためのディレクトリのリストに登録されていない）ので，a.out を実行する際には必ずカレントディレクトリを意味する./を接頭辞とする。

```
$ ./a.out ← 実行ファイル"a.out"を実行
a * x^2 + b * x + c = 0
a = 3 ← "3"を入力して[ENTER]キーを押す
b = 2 ← "2"を入力
c = 1 ← "1"を入力
Solve 3.000000 * x^2 + 2.000000 * x + 1.000000 = 0...
```

入力された係数が最後の行できちんと表示されていることを確認すること！

下記のように，-o オプションを付加することで，生成する実行ファイル名を指定することもできる。この場合はカレントディレクトリに”quadratic_eq”という実行ファイルができる。

```
$ gcc quadratic_eq.c -o quadratic_eq
$ ls -ld quadratic_eq
-rwxr-xr-x   1 tkouya   cs          5124  5月 10日  17:04 quadratic_eq
$ ./quadratic_eq
(・・・省略・・・)
```

4.1.2　計算と判別式 (if 文)

四則演算のうち，加法・減法は +, -と数学記号と同じだが，乗法は省略せずに*(アスタリスク, asterisk) と書き，除法は/(スラッシュ, slash) と書く。これを用いて判別式 d の計算式を書くと

```
d = b * b - 4.0 * a * c
```

となる。4.0 は 4 と書いても良いが，実数型 (浮動小数点数) として表現されるべき数は小数点を付ける癖を付けておくと良い。勿論，判別式の値を格納するための変数 d も適切なデータ型で宣言しておく必要がある。

また，解の計算で必要となる平方根 $\sqrt{d}$ は sqrt 関数を用いて行うことが出来る。但し，次のような追加手順が必要となる。

1. ヘッダファイルとして “math.h”(数学関数のためのヘッダファイル) を読み込む。
2. コンパイル時に-lm オプションを付けて libm.a ライブラリ (数学関数ライブラリ) をリンクする必要がある。

判別式の値が求められれば，これに基づいて解の判別を行うことが出来る。実数解，複素数解の判断は，if〜else 文を用いて次のように行えばよい。

```
// 実数解
if(d >= 0)
{
    printf("Real solutions: ¥n");
    printf("x1 = %f¥n", (-b + sqrt(d)) / (2.0 * a));
    printf("x2 = %f¥n", (-b - sqrt(d)) / (2.0 * a));
}
// 複素数解
else
{
    printf("Complex solutions: ¥n");
}
```

以上の処理を追加すれば，実数解については正しく判別して計算し，表示できるようになる。例えば $x^2 - 2x + 1 = 0$ の場合，$x_1 = x_2 = 1$ となるが，実際

```
$ gcc quadratic_eq.c -o quadratic_eq -lm ← libm.a(sqrt関数)をリンク
$ ./quadratic_eq
a * x^2 + b * x + c = 0
a = 1
b = -2
c = 1
Solve 1.000000 * x^2 + -2.000000 * b + 1.000000 = 0...
Real solutions:
x1 = 1.000000
x2 = 1.000000
```

となって正しく計算できていることが分かる。

課題 A

以上の解説を理解し，次の機能を順次実装せよ。

1. 実数解を計算して表示する。この時，次の 2 つの方程式の解が正しく計算できていることを確認せよ。
   (a) $2x^2 + 12x + 18 = 0$ ($x_1 = x_2 = -3$)
   (b) $32x^2 - 732160x - 516544800 = 0$ ($x_1 = 23565,\ x_2 = -685$)
2. 複素数解であることを判別するだけでなく，実数部，虚数部の計算をそれぞれ行って次のように表示できるよう “quadratic_eq.c”を改良し，例えば次のように複素数解になる 2 次方程式の解の計算が正しく出来ることを確認せよ。

$$3x^2 + 6x + 10 = 0 \tag{4.1}$$

$$x_1 = -1 + \frac{\sqrt{21}}{3}\mathrm{i},\ \mathrm{x}_2 = -1 - \frac{\sqrt{21}}{3}\mathrm{i} \tag{4.2}$$

3*. $a = 0$ の時も正しく 1 次方程式 $bx + c = 0$ の解を計算して表示出来るようにせよ。
4*. どんな a, b, c の値が入っても正しく処理して計算，あるいはエラー表示が出るようにせよ。

4.2 ソースファイル分割と Makefile

現在の HPC では，過去のソフトウェア資産を引き継ぎつつ，大規模化が進展しており，プログラムは機能別に小分けにして多人数で作成することが多い。そのためにもソースファイルの分割機能は不可欠であり，更新されたソースファイルだけを必要に応じてコンパイルし直すという効率化も必要である。これらを行う UNIX コマンドとして make というものがあり，これは Makefile に分割された C ソースをどのようにコンパイルし，どのように一つの実行ファイルにまとめ上げるのかを記述して使用するものである。この節では C ソースの分割化の実例として，行列・ベクトル計算を行うプログラムを作成し，Makefile の書き方と make コマンドの使い方を学んでいく。

4.2.1 行列とベクトルの乗算

実数を要素として持つ 3 次正方行列 A は

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

という形で表現される。また 3 次元のベクトル $\mathbf{b}$ は

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

と表現される。この時，行列 A とベクトル $\mathbf{b}$ の積 $= A$ は 3 次元ベクトルとなり

$$\mathbf{c} = A\mathbf{b} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + a_{13}b_3 \\ a_{21}b_1 + a_{22}b_2 + a_{23}b_3 \\ a_{31}b_1 + a_{32}b_2 + a_{33}b_3 \end{bmatrix}$$

と計算される。今，A の要素 $a_{ij}(i, j = 1, 2, ..., n)$ と $\mathbf{b}$ の要素 b_i $(i = 1, 2, ..., n)$ を

$$a_{ij} = i + j - 1,\ b_i = 3 - i$$

とすると，この計算を行う C ソース “matmul.c”は次のようになる。

```
 1: #include <stdio.h>
 2:
 3: int main()
 4: {
 5:     double mat_a[3][3], vec_b[3], vec_c[3];
 6:
 7:     // mat_a[i][j] = i + j + 1
 8:     mat_a[0][0] = 1.0;
 9:     mat_a[0][1] = 2.0;
10:     mat_a[0][2] = 3.0;
11:     mat_a[1][0] = 2.0;
12:     mat_a[1][1] = 3.0;
13:     mat_a[1][2] = 4.0;
14:     mat_a[2][0] = 3.0;
15:     mat_a[2][1] = 4.0;
16:     mat_a[2][2] = 5.0;
17:
18:     // vec_b[i] = 3 - i
19:     vec_b[0] = 3.0;
20:     vec_b[1] = 2.0;
21:     vec_b[2] = 1.0;
22:
23:     // vec_c := mat_a * vec_b
24:     vec_c[0] = mat_a[0][0] * vec_b[0] + mat_a[0][1] * vec_b[1]
+ mat_a[0][2] * vec_b[2];
25:     vec_c[1] = mat_a[1][0] * vec_b[0] + mat_a[1][1] * vec_b[1]
+ mat_a[1][2] * vec_b[2];
26:     vec_c[2] = mat_a[2][0] * vec_b[0] + mat_a[2][1] * vec_b[1]
+ mat_a[2][2] * vec_b[2];
27:
28:     // print vec_c
29:     printf("vec_c[0] = %f¥n", vec_c[0]);
30:     printf("vec_c[1] = %f¥n", vec_c[1]);
31:     printf("vec_c[2] = %f¥n", vec_c[2]);
32:
33:     return 0;
34: }
```

第 5 行目で宣言しているのは全て配列 (array) である。`mat_a[3][3]` を 2 次元配列 (3 行 3 列)，`vec_b[3]`, `vec_c[3]` を 2 次元配列 (3 行) と呼び，指定されたデータ型 (double 型) の要素を一度に確保し，一つの変数名に添字を大括弧 `[ ]` でくくって使用するための機能である。どのように配列要素を使用しているかは 8 行目以降の記述をから読み取って欲しい。これを実行すると

```
$ ./a.out
vec_c[0] = 10.000000
vec_c[1] = 16.000000
vec_c[2] = 22.000000
```

という結果を得る。つまり

$$\mathbf{c} = \begin{bmatrix} 10 \\ 16 \\ 22 \end{bmatrix}$$

である。

4.2.2 マクロとループ

3 次元と次元数が決まった行列とベクトルを計算するだけなら前述のように次元数を固定して計算式を書き並べることで結果が得られる。しかし，任意の自然数 n を次元数とする正方行列 A と n 次元ベクトル $\mathbf{b}$ が

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

のように与えられた時，この積 $A\mathbf{b}$ を

$$A\mathbf{b} = \begin{bmatrix} a_{11}b_1 + a_{12}b_2 + \cdots + a_{1n}b_n \\ a_{21}b_1 + a_{22}b_2 + \cdots + a_{2n}b_n \\ \vdots \\ a_{n1}b_1 + a_{n2}b_2 + \cdots + a_{nn}b_n \end{bmatrix}$$

と計算できるようにプログラムを作っておくことが本来は望ましい。

そこで，マクロという機能を使って，任意の次元数に対応できるように改変してみよう。

```
 1: #include <stdio.h>
 2:
 3: #define DIM 3
 4:
 5: int main()
 6: {
 7:     int i, j;
 8:     double mat_a[DIM][DIM], vec_b[DIM], vec_c[DIM];
 9:
10:     // mat_a[i][j] = i +  j + 1
11:     for(i = 0; i < DIM; i++)
12:     {
13:         for(j = 0; j < DIM; j++)
14:             mat_a[i][j] = (double)(i + j + 1);
15:     }
16:
17:     // vec_b[i] = 3 - i
18:     for(i = 0; i < DIM; i++)
19:         vec_b[i] = (double)(DIM - i);
20:
21:     // vec_c := mat_a * vec_b
22:     for(i = 0; i < DIM; i++)
23:     {
24:         vec_c[i] = 0.0;
25:         for(j = 0; j < DIM; j++)
26:             vec_c[i] += mat_a[i][j] * vec_b[j];
27:     }
28:
29:     // print vec_c
30:     for(i = 0; i < DIM; i++)
31:         printf("vec_c[%d] = %f¥n", i, vec_c[i]);
32:
33:     return 0;
34: }
```

なお，26 行目は

```
26:             vec_c[i] = vec_c[i] + mat_a[i][j] * vec_b[j];
```

と書いたのと同じ意味である。

4.2.3　オプション処理とポインタを用いた動的メモリ取得

マクロ機能とループ機能を使うことで，任意の次元数の行列・ベクトル積計算プログラムは作成できた。しかしこれでは次元数を変えるたびにコンパイルし直す必要がある。そしてコンパイルして生成されたネイティブアプリ（実行ファイル）は，ソースコードで指定した次元数の計算しかできないものになる。

そこで，実行時に次元数をオプション (option) を与えることがきるようにプログラムを改変してみよう。

UNIX の C 言語におけるオプション機能は次のように，main 関数の引数として与えることで使用することが出来る。

```
          int main(int argc, char *argv[])
        オプションの個数^~~~~~~~  ^^^^^^^^^^^^オプションの文字列
```

例えば次のプログラム (print_option.c) をコンパイルして実行してみよう。

```
 1: #include <stdio.h>
 2:
 3: int main(int argc, char *argv[])
 4: {
 5:     int i;
 6:
 7:     printf("オプションの数(argc): %d¥n", argc);
 8:
 9:     for(i = 0; i < argc; i++)
10:         printf("argv[%d] = %s¥n", i, argv[i]);
11:
12:     return 0;
13: }
```

これをコンパイルして実行する。オプションは

```
$␣実行ファイル名␣オプション1␣オプション2
```

のように，半角スペースで区切って実行ファイル名の後に付加する。

```
$ gcc print_option.c
$ ./a.out
オプションの数(argc): 1 ←オプションなしの時は実行ファイル名のみ(オプション1個)
argv[0] = ./a.out
$ ./a.out op1 op2 op3
オプションの数(argc): 4 ←実行ファイル名も含めてオプション数を返す
argv[0] = ./a.out
argv[1] = op1
argv[2] = op2
argv[3] = op3
```

実行時には必ず実行ファイル (コマンド) を入力するので，それが最初のオプションとなる。従って，argv[0] は実行ファイル名が文字列として与えられる。実行ファイル名以

降のオプションは argv[1], argv[2], ... に与えられる。

この機能を用いて，行列の次元数を argv[1] に与え，次元数に応じて行列とベクトルの格納領域を自動的に確保するプログラムに改変してみよう。

```
 1: #include <stdio.h>
 2: #include <stdlib.h>
 3:
 4: int main(int argc, char *argv[])
 5: {
 6:     int dim; // dimension
 7:     int i, j;
 8:     double *mat_a, *vec_b, *vec_c;
 9:
10:     // check dimension
11:     if(argc <= 1)
12:     {
13:         printf("Usage: %s [dimension]¥n", argv[0]);
14:         return 0;
15:     }
16:
17:     // input dimension
18:     dim = atoi(argv[1]);
19:     printf("Dimension = %d¥n", dim);
20:
21:     // initialize
22:     mat_a = (double *)calloc(dim * dim, sizeof(double));
23:     vec_b = (double *)calloc(dim, sizeof(double));
24:     vec_c = (double *)calloc(dim, sizeof(double));
25:
26:     // mat_a[i][j] = i +  j + 1
27:     for(i = 0; i < dim; i++)
28:     {
29:         for(j = 0; j < dim; j++)
30:             *(mat_a + i * dim + j) = (double)(i + j + 1);
31:     }
32:
33:     // vec_b[i] = 3 - i
34:     for(i = 0; i < dim; i++)
35:         *(vec_b + i) = (double)(dim - i);
36:
37:     // vec_c := mat_a * vec_b
38:     for(i = 0; i < dim; i++)
39:     {
40:         *(vec_c + i) = 0.0;
41:         for(j = 0; j < dim; j++)
42:             *(vec_c + i) += *(mat_a + i * dim + j) * *(vec_b + j);
43:     }
44:
45:     // print vec_c
46:     for(i = 0; i < dim; i++)
47:         printf("vec_c[%d] = %f¥n", i, *(vec_c + i));
48:
49:     // free
50:     free(mat_a);
51:     free(vec_b);
52:     free(vec_c);
53:
54:     return 0;
55: }
```

配列バージョンと異なる部分だけ解説しよう。まず

```
 8:     double *mat_a, *vec_b, *vec_c;
```

のアスタリスク (*) はポインタ (pointer) を意味する。配列を格納するメインメモリ領域の先頭アドレスを表現しているとイメージして貰えばよい。従って，ここではまだ double 型のメモリ領域を作るぞと言う宣言をしているだけで，格納したデータの実体は存在しない。それを確保するのは

```
17:     // input dimension
18:     dim = atoi(argv[1]);
19:     printf("Dimension = %d¥n", dim);
20:
21:     // initialize
22:     mat_a = (double *)calloc(dim * dim, sizeof(double));
23:     vec_b = (double *)calloc(dim, sizeof(double));
24:     vec_c = (double *)calloc(dim, sizeof(double));
```

の 22〜24 行目の calloc 関数である。オプションで与えられた次元数を整数 (dim) に変換して表示し (18, 19 行目)，行列 mat_a は dim×dim 個の，ベクトル vec_b, vec_c は dim 個の double 型のデータを保存する領域がメインメモリ上に動的に確保される。

配列と同様に確保されたメモリ領域にアクセスする時には

```
*(ポインタ名 + 先頭からのデータの個数)
```

とする。例えば 30 行目では，i 行 j 列目の行列の値を

```
30:             *(mat_a + i * dim + j) = (double)(i + j + 1);
```

というように代入先を指定している。

ポインタと結びつけられたメモリ領域が不要になれば

```
49:     // free
50:     free(mat_a);
51:     free(vec_b);
52:     free(vec_c);
```

と free 関数を用いて解放する。

このように，C 言語にはポインタに結びつけられたメモリ領域をプログラム実行中，不要になった時点で自動的に解放する機構（ガベージコレクション）が存在しない。この点は PHP のようなスクリプト言語 (後述) と異なり，あまり使い勝手が良くないと言われている。

4.2.4　関数化とソースファイルの分割

最後に，長くなったプログラムの一部を使い回しできるよう，別ファイルに分割してみよう。これは行列・ベクトル積の計算部分を

```
// vec_b[i] = 3 - i
for(i = 0; i < dim; i++)
    *(vec_b + i) = (double)(dim - i);
```

```
    // vec_c := mat_a * vec_b
    mat_vec_mul(vec_c, mat_a, vec_b, dim);

    // print vec_c
    for(i = 0; i < dim; i++)
        printf("vec_c[%d] = %f¥n", i, *(vec_c + i));
```

というように，新たに作成した mat_vec_mul 関数として独立させたものである。この独立した関数を別ファイル “mat_vec_mul.c”に保存しておこう。この時，ファイルには

```
// Matrix x Vector
// vec_ret := mat * vec
void mat_vec_mul(double *vec_ret, double *mat, double *vec, int dim)
{
    int i, j;

    // vec_c := mat_a * vec_b
    for(i = 0; i < dim; i++)
    {
        *(vec_ret + i) = 0.0;
        for(j = 0; j < dim; j++)
            *(vec_ret + i) += *(mat + i * dim + j) * *(vec + j);
    }
}
```

という記述がなされる。残りの main 関数は “matmul_main.c”というファイル名にしておこう。

分割されたソースファイルを同時にコンパイルして一つの実行ファイルにコンパイルするには

```
$ gcc matmul_main.c mat_vec_mul.c -o matmul
```

と指定する。

4.2.5 Makefile と make コマンドの利用

複数の C ソースファイルに分割された実行ファイルを生成する際には make コマンドを使うと便利である。make コマンドは，Makefile(あるいは指定されたテキストファイル)にコンパイル方法を記述し，それに沿って必要な時に必要なファイルだけを生成したり削除したりすることを可能にするものである。

例えば

```
CC = gcc

all: mat_mul

mat_mul: matmul_main.c mat_vec_mul.c
        $(CC) matmul_main.c mat_vec_mul.c -o matmul

clean:
        -rm *.o
        -rm matmul
```

という Makefile を書くと，次のような処理が可能になる。

■コンパイル: make のみ

```
$ make
gcc matmul_main.c mat_vec_mul.c -o matmul
$ ls -l ./matmul
-rwxrwxr-x 1 tkouya tkouya 8629  1月  7 21:15 ./matmul
```

■実行ファイル消去: make clean

```
$ make clean
rm *.o
rm: cannot remove ‘*.o’: そのようなファイルやディレクトリはありません
make: [clean] エラー 1 (無視されました)
rm matmul
$ ls -l ./matmul
ls: ./matmul: そのようなファイルやディレクトリはありません
```

課題 B

1. “quadratic_eq.c”のうち，2 次方程式を解く部分を関数化し，その部分を“quadratic_eq_solver.c”に分割せよ。残りのプログラムは ‘quadratic_eq_main.c”として保存し，これらを分割コンパイルする Makefile を作成せよ。
2. 上記の Makefile の機能を，行列ベクトル積プログラムをコンパイルするための Makefile と統合し，二つの実行ファイルが作成できるようにせよ。

4.3　ベンチマークテストの方法

ソフトウェアの実行性能を測りたい時には，処理時間がどの程度なのかを計測し，比較検討する。そのためにここではコマンド (実行ファイル) の時間計測の手法を 2 種類紹介する。

4.3.1　time コマンド (bash シェル固有) を用いた時間計測

コマンド (実行ファイル) 全体の処理時間を計測したい時には time コマンドを使用する。これは bash シェル (ユーザからの入出力を受け付け，カーネルとの橋渡しをする CUI) の機能である。

```
$ time ./matmul 100
Dimension = 100
Clock number per second: 100 (clocks/sec)
Run Time (Clock)    : 0
Run Time (Second)   : 0.000000
System Time (Second): 0.000000
User Time (Second)  : 0.000000

real    0m0.007s
user    0m0.001s
sys     0m0.006s
```

4.3.2 times 関数を用いた時間計測

プログラムの一部の処理を計測したい時には，プログラム内に時間計測関数を埋め込む。C 言語の場合は次のように times 関数を使うことが多い。

```
#include <sys/times.h> // for times
#include <unistd.h> // for sysconf

・・・・・・・・・

    struct tms start_time, end_time; // get run time
    clock_t start_clock, end_clock;

    ・・・・・・

    start_clock = times(&start_time);

    ・・・・時間計測を行いたい処理・・・・

    end_clock = times(&end_time);

    // 1秒あたりのクロック数: sysconf(_SC_CLK_TCK)で取得
    printf("Clock number per second: %d (clocks/sec)¥n", sysconf(_SC_CLK_T
CK));

    // 計測クロックの表示
    printf("Run Time (Clock)    : %d¥n", end_clock - start_clock);

    // 計測時間の表示(秒単位)
    printf("Run Time (Second)   : %f¥n", (double)(end_clock - start_clock)
 / (double)sysconf(_SC_CLK_TCK));

    // システム時間 : OSが使用した時間
    printf("System Time (Second): %f¥n", (double)(end_time.tms_stime - sta
rt_time.tms_stime) / (double)sysconf(_SC_CLK_TCK));

    // ユーザ時間 : ユーザのプログラムが使用した時間
    printf("User Time (Second)  : %f¥n", (double)(end_time.tms_utime - star
t_time.tms_utime) / (double)sysconf(_SC_CLK_TCK));
```

時間計測を何回か試行してみると，実行するたびに時間が異なることが分かる。複数のプロセスが平行して動作している環境では特に差が出やすい。従って，時間計測の差異には何度か繰り返し，その結果の平均値を使う必要がある。

課題 C

次元数を 100, 1000, 2000 として行列・ベクトル積プログラムの時間計測を行え。またその結果を 5 回, 10 回，15 回行った時の平均値を求め，その変化について考察せよ。(ヒント：「大数の法則」を検索してみよ。)

第 4 章の学習チェックリスト

- □ C プログラムの基本的な文法を理解し，単純な計算を行うプログラムを製作，コンパイルして実行することができる。
- □ コンパイルした C プログラムにオプションを与え，それを解釈してプログラムの挙動を変更させる方法を理解し，オプション機能付き C プログラムを製作，コンパイル，実行することができる。
- □ プログラムの実行時間を計測することができる。