第 6 章

線型計算におけるデータ構造

ここでは第一部の最後として，線型計算をコンピュータ上で実行するための準備を行う。コンピュータは例外なく，指定された手順書，すなわち「プログラム」に従って処理を行う。プログラムには，処理の対象となるデータがどのような性質のもので，どのように記憶装置に格納されているのか，という「データ構造」と，データを処理する方法と手順，即ち「アルゴリズム」が記述されている。線型計算を実行するプログラムも同様に，ベクトルや行列を格納するためのデータ構造やアルゴリズムを記述したものとなる。本章ではまずデータ構造について述べた後，基本線型計算を例にアルゴリズムとその計算量を確認していく。

6.1 行列の種類とその数学的性質

今まで見てきたように，行列は，実行列 (すべての要素が実数に限定) か複素行列 (複素数要素を含む) か，正方行列 (行数と列数が一致) か一般行列か，という分類ができる。このうち，本書では以降，特に断らない限り正方行列のみ扱うことにする。即ち，実正方行列 $\mathbb{R}^{n\times n}$ と複素正方行列 $\mathbb{C}^{n\times n}$ のみ扱う，ということである。

ここでは正方行列を次のように分類し，その数学的性質を確認する。

正方行列　行数と列数が同じ。

- **実正方行列**　全ての要素が実数。
  - **対称行列**　転置行列が元の行列と同じになる行列。
  - **直交行列**　転置行列が逆行列になる行列。
- **複素正方行列**　複素数要素を含む行列。
  - **エルミート行列**　転置共役行列が元の行列と同じになる行列。
  - **ユニタリ行列**　転置共役行列が逆行列になる行列。
- Hessenberg **行列**　下副対角要素より下の要素がすべてゼロの行列。
  - **三重対角行列**　上下副対角要素，対角要素以外の要素がすべてゼロの行列。
  - **上 (下) 三角行列**　対角成分より下 (上) の要素がすべてゼロの行列。
  - **対角行列**　対角成分以外の全ての要素がゼロの行列。

6.1.1　対称行列とエルミート行列

複素正方行列 $C \in \mathbb{C}^{n\times n}$ に対してすべての要素を共役複素数に置き換える操作を

$$\overline{C} = \begin{bmatrix} \overline{c_{11}} & \overline{c_{12}} & \cdots & \overline{c_{1n}} \\ \overline{c_{21}} & \overline{c_{22}} & \cdots & \overline{c_{2n}} \\ \vdots & \vdots & & \vdots \\ \overline{c_{n1}} & \overline{c_{n2}} & \cdots & \overline{c_{nn}} \end{bmatrix}$$

と書くことにする。この時，もし

$$\overline{C}^T = C$$

が成立する時，C をエルミート (Hermite) 行列と呼ぶ。

もし C が実正方行列であれば，$\overline{C} = C$ より

$$C^T = C$$

となる。この時，C は対称 (symmetric) 行列と呼ぶ。

6.1.2　直交行列とユニタリ行列

複素正方行列 $U \in \mathbb{C}^{n\times n}$ において

$$\overline{U}^T U = U\overline{U}^T = I_n$$

即ち，$U^{-1} = \overline{U}^T$ となる時，U をユニタリ (unitary) 行列と呼ぶ。

U が実正方行列であれば，$\overline{U} = U$ であるから

$$U^T U = UU^T = I_n$$

となる。この時，U を直交 (orthogonal) 行列と呼ぶ。

U がユニタリ行列であれば，これを列ベクトル形式

$$U = [\mathbf{u}_1\ \mathbf{u}_2\ ...\ \mathbf{u}_n]\ (\mathbf{u}_i \in \mathbb{C}^n, i = 1, 2, ..., n)$$

で表現すると

$$U\overline{U}^T = \left(\overline{U}^T U\right)^T = \left(\begin{bmatrix} \overline{\mathbf{u}_1}^T \\ \overline{\mathbf{u}_2}^T \\ \vdots \\ \overline{\mathbf{u}_n}^T \end{bmatrix} [\mathbf{u}_1\ \mathbf{u}_2\ \ldots\ \mathbf{u}_n]\right)^T$$

$$= \begin{bmatrix} (\mathbf{u}_1, \mathbf{u}_1) & (\mathbf{u}_1, \mathbf{u}_2) & \cdots & (\mathbf{u}_1, \mathbf{u}_n) \\ (\mathbf{u}_2, \mathbf{u}_2) & (\mathbf{u}_2, \mathbf{u}_2) & \cdots & (\mathbf{u}_2, \mathbf{u}_n) \\ \vdots & \vdots & & \vdots \\ (\mathbf{u}_n, \mathbf{u}_1) & (\mathbf{u}_n, \mathbf{u}_n) & \cdots & (\mathbf{u}_n, \mathbf{u}_n) \end{bmatrix}^T$$

$$= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix}$$

であるから,

$$\mathbf{u}_i \overline{\mathbf{u}_j}^T = (\mathbf{u}_i, \mathbf{u}_j) = \begin{cases} 1 & (i = j) \\ 0 & (i \neq j) \end{cases}$$

であることを意味する。つまり互いに直交する n 本の n 次元ベクトルによってユニタリ (直交) 行列が形成されている，とも言える。

6.1.3 対角行列，3 重対角行列，上 (下) 三角行列，ヘッセンベルグ行列

単位行列のように，対角成分 $a_{ii}(i = 1, 2, ..., n)$ 以外の成分がすべてゼロの n 次正方行列 D を対角行列 (diagonal) と呼ぶ。ゼロ要素は省略して書くこともある。

$$D = \begin{bmatrix} d_1 & 0 & \cdots & 0 \\ 0 & d_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & d_n \end{bmatrix} = \begin{bmatrix} d_1 & & & \\ & \ddots & & \\ & & & d_n \end{bmatrix}$$

対角成分の上の要素 $a_{i,i+1}(i = 1, 2, ..., n-1)$ を上副対角要素 (upper subdiagonal element), 対角要素の下の要素 $a_{i+1,i}(i = 1, 2, ..., n-1)$ を下副対角要素 (lower subdiagonal element) と呼ぶ。対角要素，上下副対角要素以外の要素がすべてゼロの行列 T を，三重対角 (tridiagonal) 行列と呼ぶ。

$$T = \begin{bmatrix} t_{11} & t_{12} & 0 & \cdots & \cdots & 0 \\ t_{21} & t_{22} & t_{23} & 0 & & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & t_{n-1,n} & t_{n-1,n-1} & t_{n-1,n} \\ 0 & \cdots & \cdots & 0 & t_{n,n-1} & t_{nn} \end{bmatrix} = \begin{bmatrix} t_{11} & t_{12} & & & \\ t_{21} & \ddots & \ddots & & \\ & \ddots & \ddots & t_{n-1,n} & \\ & & t_{n,n-1} & t_{nn} & \end{bmatrix}$$

下副対角要素より下の要素がすべてゼロの行列 H をヘッセンベルグ (Hessenberg) 行列と呼ぶ。

$$H = \begin{bmatrix} h_{11} & h_{12} & \cdots & \cdots & h_{1n} \\ h_{21} & h_{22} & \ddots & & \vdots \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & h_{n-1,n} & h_{n-1,n-1} & h_{n-1,n} \\ 0 & \cdots & 0 & h_{n,n-1} & h_{nn} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & \cdots & h_{1n} \\ h_{21} & \ddots & \ddots & \vdots \\ & \ddots & \ddots & h_{n-1,n} \\ & & h_{n,n-1} & h_{nn} \end{bmatrix}$$

対角成分より下の要素がすべてゼロの行列を上三角行列 U, 対角成分より上の要素がすべてゼロの行列を下三角行列 L と呼ぶ。

$$U = \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \ddots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{bmatrix} = \begin{bmatrix} u_{11} & \cdots & u_{1n} \\ & \ddots & \vdots \\ & & u_{nn} \end{bmatrix}$$

$$L = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n1} & \cdots & l_{n,n-1} & l_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & & \\ \vdots & \ddots & \\ l_{n1} & \cdots & l_{nn} \end{bmatrix}$$

問題 6.1

1. 単位行列 I_n，ゼロ行列 O はどのような性質を持つか？
2. 次の行列 $A, B, C \in \mathbb{C}^{3\times 3}$ はどのような行列か？

$$A = \begin{bmatrix} -4 & 0 & 4 \\ 0 & 5 & 1 \\ 4 & 1 & -3 \end{bmatrix}, B = \begin{bmatrix} \mathrm{i} & -\mathrm{i} & 3+2\mathrm{i} \\ \mathrm{i} & 2-4\mathrm{i} & -9 \\ 3-2\mathrm{i} & -9 & 1+4\mathrm{i} \end{bmatrix}, C = \begin{bmatrix} 2 & -1 & 0 \\ -2 & 3 & 1 \\ 0 & 5 & 4 \end{bmatrix}$$

6.2 基本線型計算のアルゴリズムと計算量

6.2.1 浮動小数点数の四則演算と Landau の O 記号

小学校で習う小数の加減乗除は整数のそれと本質的には同じものである。人間の感覚で言う「面倒くさい」計算は，そのままコンピュータについても当てはまる。面倒な計算は時間を要する。従って四則演算は前章のベンチマークテストからも分かるように，

$$T(\text{加算 (FADD)}) = T(\text{減算 (FSUB)}) \leq T(\text{乗算 (FMUL)}) < T(\text{除算 (FDIV)}) \tag{6.1}$$

という順に計算時間を要すると考えて良い。後で述べる初等関数はこれらの演算を組み合わせて実行されるため，更に時間を要するのが普通である。但し，現在の CPU は内部に積み込んだ高速転送可能なキャッシュ (cache) メモリを持っており，一度メインメモリから読み込んだ値をそこに記憶しておき，2 度目以降のアクセスはそれを取り出すだけで済む。従って，このキャッシュメモリをうまく利用できるよ

うにした線型計算プログラムは，素朴に組んだものより高速になる。よって単純に計算量だけでは計算時間を推定できないこともある。

従って，数値計算のアルゴリズムの計算時間は加減算の実行回数以上に，乗除算や初等関数の実行回数に左右される。「アルゴリズムの演算回数」という言葉がしばしば後者の意味で使われるのはそのためである。

計算回数に限らず，様々な場面で使用される言葉としてオーダ (order) がある。これは以下に示す Landau の O(ラージオー) と同義である。

定義 6.1 (Landau の O 記号)
ある一変数実関数 $f(x), g(x) \in \mathbb{R}$ に対して，$f(x) = O(g(x))$ とは

$$\lim_{x \to \alpha} \frac{f(x)}{g(x)} = \text{定数} \ (\neq 0)$$

となることを意味し，このような $f(x)$ は $g(x)$ のオーダ (order) であると呼ぶ。この $O(g(x))$ を Landau の O(ラージオー) 記号という。α としては 0 もしくは $\pm\infty$ がよく使用される。

また

$$\lim_{x \to \alpha} \frac{f(x)}{g(x)} = 0$$

であるときは特に $f(x) = o(g(x))$ と書き，これを o(スモールオー) 記号と呼ぶ。

以降，オーダという言葉は O(ラージオー) の意味で使用する。$g(x)$ としてよく使用されるのは x の多項式であり，特に $x^2, x^3, \ldots x^n$ である。$O(x^n)$ はほぼ x^n に比例していることを表しており，直感的に理解しやすいため，様々な場面で使用される。

6.2.2 複素数の四則演算

本書は実数の演算が主体であるが，複素数の演算が必要となる場面に遭遇することもある。ここで復習も兼ねて，複素数の演算とその演算量について若干の考察を行う。

任意の複素数 $c = \mathrm{Re}(c) + \mathrm{Im}(c)\sqrt{-1} \in \mathbb{C}$ は 2 つの実数の組 $(\mathrm{Re}(c), \mathrm{Im}(c))$ として表現できる。従って，複素数の四則演算は全て実数のそれを組み合わせることによって実現できる。

$$|a| = \sqrt{(\mathrm{Re}(a))^2 + (\mathrm{Im}(a))^2} \tag{6.2}$$

$$a \pm b = (\mathrm{Re}(a) \pm \mathrm{Re}(b)) + \sqrt{-1}(\mathrm{Im}(a) \pm \mathrm{Im}(b)) \tag{6.3}$$

$$ab = (\mathrm{Re}(a)\mathrm{Re}(b) - \mathrm{Im}(a)\mathrm{Im}(b)) + \sqrt{-1}(\mathrm{Im}(a)\mathrm{Re}(b) + \mathrm{Re}(a)\mathrm{Im}(b)) \tag{6.4}$$

$$a/b = \frac{a\overline{b}}{|b|^2} \tag{6.5}$$

ここで $\overline{b} = \mathrm{Re}(b) - \mathrm{Im}(b)\sqrt{-1}$ である。

表 6.1　複素数演算の計算回数

	加減算	乗算	除算	平方根
$\|a\|$ ((6.2) 式)	1	2	0	1
$\|a\|$ ((6.6) 式)	1	2	1	1
$a \pm b$	2	0	0	0
ab	2	4	0	0
a/b ((6.5) 式)	3	6	2	0
a/b ((6.7) 式)	3	3	3	0

但し，オーバーフローを防止するため，$|a|$ と a/b は次のように計算するのが良いとされている [2]。

$$|a| = \begin{cases} \mathrm{Re}(a) & (\text{if } \mathrm{Im}(a) = 0) \\ \mathrm{Im}(a) & (\text{if } \mathrm{Re}(a) = 0) \\ |\mathrm{Re}(a)| \sqrt{1 + \left(\dfrac{\mathrm{Im}(a)}{\mathrm{Re}(a)}\right)^2} & (\text{if } |\mathrm{Re}(a)| \geq |\mathrm{Im}(a)| > 0) \\ |\mathrm{Im}(a)| \sqrt{1 + \left(\dfrac{\mathrm{Re}(a)}{\mathrm{Im}(a)}\right)^2} & (\text{if } |\mathrm{Im}(a)| > |\mathrm{Re}(a)| > 0) \end{cases} \tag{6.6}$$

$$a/b = \begin{cases} \text{計算不能} & (\text{if } b = 0 \\ & \text{(即ち } \mathrm{Re}(b) = \mathrm{Im}(b) = 0)) \\ \dfrac{\mathrm{Re}(a) + \mathrm{Im}(a) \cdot \left(\dfrac{\mathrm{Im}(b)}{\mathrm{Re}(b)}\right)}{s} + \dfrac{-\mathrm{Re}(a) \cdot \left(\dfrac{\mathrm{Im}(b)}{\mathrm{Re}(b)}\right) + \mathrm{Im}(a)}{s} \sqrt{-1} & \\ & (\text{if } |\mathrm{Re}(b)| \geq |\mathrm{Im}(b)| \geq 0) \\ \text{ここで } s = \mathrm{Re}(b) + Im(b) \cdot \left(\dfrac{Im(b)}{\mathrm{Re}(b)}\right) & \\ \dfrac{\mathrm{Re}(a) \cdot \left(\dfrac{\mathrm{Re}(b)}{\mathrm{Im}(b)}\right) + \mathrm{Im}(a)}{s} + \dfrac{-\mathrm{Re}(a) + \mathrm{Im}(a) \cdot \left(\dfrac{\mathrm{Re}(b)}{\mathrm{Im}(b)}\right)}{s} \sqrt{-1} & \\ & (\text{if } |\mathrm{Im}(b)| \geq |\mathrm{Re}(b)| \geq 0) \\ \text{ここで } s = \mathrm{Re}(b) \cdot \left(\dfrac{\mathrm{Re}(b)}{\mathrm{Im}(b)}\right) + \mathrm{Im}(b) & \end{cases} \tag{6.7}$$

以上の複素数演算の計算回数を表 6.1 にまとめておく。対応する実数の演算と比べて，2〜3 倍の演算量を必要とすることが分かる。従って，複素数の演算は実数のそれに比べてかなり「高くつく」ことを認識しておく必要がある。当然のことながら，必要となるメモリ量も実数の 2 倍になる。

例題 6.1

1. 式 (6.6)，(6.7) がそれぞれ $|a|$ と a/b を計算していることを確認せよ。
2. 前章の多倍長浮動小数点数の四則演算のベンチマークテストの結果を用いて，複素数演算の性能評価を行え。また実際にベンチマークテストを行った結果と比較せよ。

表 6.2 ベクトル演算の計算回数

	加減算	乗算
$\alpha\mathbf{a}$	0	n
$\mathbf{a} \pm \mathbf{b}$	n	0
$(\mathbf{a}, \mathbf{b})$	$n-1$[a]	n

[a] 実際の計算ではあらかじめゼロクリアした変数に成分の積を足し込むため，n となることもある。

表 6.3 行列演算の計算回数

	加減算	乗算
$A\mathbf{b}$	$n(n-1)$[a]	n^2
αA	0	n^2
AB	$n^2(n-1)$[b]	n^3

[a] 実際の計算ではあらかじめゼロクリアした変数に成分の積を足し込むため，n^2 となることもある。
[b] 実際の計算ではあらかじめゼロクリアした変数に成分の積を足し込むため，n^3 となることもある。

6.2.3 基本線型計算

現在の数値計算は大規模化が進んでおり，そこではベクトル及び行列の基本線型計算 (linear computation) が多用される。基本線型計算は次元数が上がるにつれて莫大な計算量を必要とすることを認識しておく必要がある。ここではその一端に触れることにする。

実ベクトル $\mathbf{a} = [a_1\ a_2\ \cdots a_n]^T \in \mathbb{R}^n$ の基本線型計算，および，実正方行列 $A = [a_{ij}] \in M_n(\mathbb{R})$ $(i, j = 1, 2, ..., n)$ の基本線型計算の計算回数を表 6.2，6.3 にまとめておく。

Scilab スクリプトで，基本線型計算の演算量を確認してみよう。まず $A \in \mathbb{R}^{n\times n}$, $\mathbf{b} \in \mathbb{R}^n$

$$A = \begin{bmatrix} \sqrt{2}(n-2) & \sqrt{2}(n-3) & \cdots & \sqrt{2}(-1) \\ \sqrt{2}(n-3) & \sqrt{2}(n-4) & \cdots & \sqrt{2}(-2) \\ \vdots & \vdots & & \vdots \\ \sqrt{2}(-1) & \sqrt{2}(-2) & \cdots & \sqrt{2}(-n) \end{bmatrix} = [\ \sqrt{2}(n-(i+j)]_{i,j=1}^{n}$$

$$\mathbf{b} = \begin{bmatrix} \sqrt{2} \\ 2\sqrt{2} \\ \vdots \\ n\sqrt{2} \end{bmatrix}$$

を用いて行列・ベクトル積 $A\mathbf{b}$ の計算と，加法・乗法 1 回あたりの演算時間を計測するスクリプト (benchmark.sce) を以下に示す。

```
1: // 行列ベクトル積のベンチマークテスト
2:
3: // 見出し
4: printf("行列ベクトル積ベンチマークテスト\n");
5: printf("次元数,          秒数,      GFlops\n")
6:
7: // 次元数セット
8: index = 1;
```

```
graph_x = [];
graph_gflops = [];
for dim = 500:100:1000

    // ベクトル (vec) のセット
    vec = [];
    for i = 1:dim
        vec(i) = sqrt(2) * i;
    end;

    // 行列 (mat) のセット
    mat = [];
    for i = 1:dim
        for j = 1:dim
            mat(i, j) = sqrt(2) * (dim - (i + j));
        end
    end

    // 行列・ベクトル積の計算
    tic();
    vec_ret = mat * vec;
    mat_vec_mul_time = toc();

    // グラフ描画用
    graph_x(index) = dim;
    graph_sec(index) = mat_vec_mul_time;
    graph_gflops(index) = (dim * dim * 2) / mat_vec_mul_ti
me / 1024^3;
    printf("%6d, %10.3g, %10.3g\n", dim, graph_sec(index),
 graph_gflops(index));

    index = index + 1;
end; // ベンチマーク終了

// グラフ描画
plot(graph_x, graph_gflops);
```

これを実行すると, まず 29 行目の行列・ベクトル積の計算に要した計算時間が変数 mat_vec_mul_time

に格納される。これを使って加法と乗法の演算量を合計した $2n^2$ を割ると，1 秒間に何回の浮動小数点演算が実行できるか，即ち，Flops(FLoating-point Operations Per Second) 値が算出できる。桁が大きすぎるため，ここでは 1024^3 で割って，GFlops(Giga Flops) を算出してある。

算出された値を順次出力し，

行列ベクトル積ベンチマークテスト

```
次元数,        秒数,     GFlops
   500,       0.002,       0.25
   600,       0.002,       0.36
   700,       0.003,      0.327
   800,       0.004,       0.32
   900,       0.005,      0.324
  1000,       0.006,      0.333
```

のような出力結果を得たら，GFlops 値をプロットしたグラフを 42 行目で出力している。

例題 6.2

1. $\alpha, \beta \in \mathbb{R}$，$\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ であるとき，$\alpha\mathbf{a} \pm \beta\mathbf{b}$ の計算量を求めよ。
2. $\alpha, \beta \in \mathbb{R}, A, B, C \in M_n(\mathbb{R})$ である時，$(\alpha A \pm \beta B)C$ の計算量を求めよ。
3. $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$ である時，内積 $(\mathbf{a}, \mathbf{b})$ の計算量を求めよ。
4. benchmark.sce を参考にして，正方行列 A, B の積 AB の演算時間を計測し，GFlops 値を求める Scilab スクリプト，benchmark_mat.sce を作れ。行列 A, B は下記のように与えよ。

```
// 行列 a(mat_a) のセット
mat_a = [];
for i = 1:dim
    for j = 1:dim
        mat_a(i, j) = sqrt(2) * (i + j - 1);
    end
end;

// 行列 b(mat_b) のセット
mat_b = [];
for i = 1:dim
    for j = 1:dim
        mat_b(i, j) = sqrt(2) * (dim * 2 - (i + j - 1));
    end
end
```