乱数、連立１次方程式、固有値問題、フーリエ変換

乱数

乱数生成

計算機による乱数生成は、数値演算の結果であり厳密には**疑似乱数**というのが正しいのですが、実用上は問題ない程度に一様な乱数を生成するよう設計されています。

ここでは、Park-Miller の乱数生成式に従った関数を題材とします。

$$X_{k+1} = X_k \cdot g \bmod n$$

生成式自体は、剰余を次の乱数とする単純なものですが、乗算の際の桁あふれを考慮すると、こみ入ったものとなります。

$$m = a\,q + r,\ \text{つまり}\ q = [\,m / a\,],\ r = m \bmod a$$

ここで、[ ] は整数部分を表す

という関係が成り立っているものとします。すると、次の式が成り立ちます。

$$a\,z \bmod m = \begin{cases} a(z \bmod q) - r[z/q] & \text{この式の数値が } \geqq 0 \text{ のとき} \\ a(z \bmod q) - r[z/q] + m & \text{上の式の数値が } < 0 \text{ だったとき} \end{cases}$$

次の、プログラムコード 1 は、上記のロジックに従い、疑似乱数を生成するコードです。

演習課題との関係で、変数名をわざと分かりづらくしてあることに注意して下さい。

```
const long int a = 3399;
const long int b = 48271;
const long int c = 44488;
const long int d = 2147483647;
void rand0(long int *j)
{
    long int k = (*j) / c;
    *j = b * (*j - k * c) - a * k;
    if (*j < 0)
        *j += d;
}
```

プログラムコード 1

演習課題との関係から、rand0() では、生成された乱数値は関数の戻り値としてではなく、引数に渡された変数に直接セットされます。

また、乱数の計算に前回生成した乱数値を利用するので、乱数値を保持する変数を一貫して使用することを前提としています。また、rand0() に与える long int 値が 0 であると、原理的に生成される数値が 0 になってしまうので、初回時に渡す引数の値を 0 以外にセットして関数を呼び出す必要があります。

演習課題 6-1

演習で使用する GCC は、C99 で新たに導入された機能、long long int 型が利用可能で、long int 型の 2 倍の記憶容量を持っています。それにより、long int 同士の乗算を桁あふれなしに演算できるようになりました。したがって、上記のコードは、よりシンプルなものに書き直すことができます。

long long int 型を使って、プログラムコード 1 と同じ系列の疑似乱数を生成するプログラム rand2() を完成させて下さい。

- ファイル名を kadai_6_1.c として下さい。
- long long int 型の定数は、LL を後置します。例えば、1234LL と記述します。
- printf() で long long int 型を使う場合、型指定の記号 'll' と整数指定の記号 'd' の組み合わせを用います。例えば、"value=%lld" という風に書きます。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、

のように、枠で示されています。ただし、枠の大きさは、おおよそコード 1 行を表すものとします。

ヒント 関数の戻り値は、疑似乱数値とは違い、UnitTest でのみ使われるものです。

```
long long int rand2(long int *seed)
{
```

```
    long long int acc = (long long int)*seed [redacted];
    *seed = acc [redacted];
    return acc;
}
```

連立１次方程式

連立１次方程式の解法

次の連立１次方程式を考えてみます。

$$
\begin{aligned}
a_{11}\,x_1 + a_{12}\,x_2 + a_{13}\,x_3 + \ldots + a_{1N}\,x_N &= b_1 \\
a_{21}\,x_1 + a_{22}\,x_2 + a_{23}\,x_3 + \ldots + a_{2N}\,x_N &= b_2 \\
a_{31}\,x_1 + a_{32}\,x_2 + a_{33}\,x_3 + \ldots + a_{3N}\,x_N &= b_3 \\
&\ldots \\
a_{M1}\,x_1 + a_{M2}\,x_2 + a_{M3}\,x_3 + \ldots + a_{MN}\,x_N &= b_M
\end{aligned}
$$

ここで、係数 a_{ij} (i = 1, 2, ..., M; j = 1, 2, ..., N) と b_i (i = 1, 2, ..., M) が与えられており、未知の x_j (j = 1, 2, ..., N) が M 個の方程式で関連づけられています。

線型（線形）代数では、このような連立方程式を行列を用いて解く方法があります。係数と右辺値を行列の成分として計算をしますが、たとえば次のような 4 次の行列式を考えてみます。

$$\begin{vmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{vmatrix} \begin{vmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{vmatrix}$$

これを次のように変形させて、解を得るやり方を**ガウス・ジョルダン(Gauss-Jordan)法**といいます。

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix} \begin{vmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{vmatrix} = \begin{vmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \end{vmatrix}$$

この変形の過程で、行列Aの逆行列を求めることができます。上の変形は、以下の変形と見なすことができます。

$$\begin{aligned} A\,X &= B \\ A\,X &= E\,B \\ A^{-1}A\,X &= A^{-1}E\,B \\ E\,X &= A^{-1}B \end{aligned}$$

この変形は、右辺ベクトル B に逆行列を掛けていることになるので、単位行列に対して同じ操作を行うと、逆行列を求めることができます。

演習課題 6-2

下にある Gauss-Jordan 法に基づく関数 gauss_jordan() を完成させてください。関数の仕様は、以下の通りとします。

- ファイル名を kadai_6_2.c として下さい。
- 引数として与えられた配列の内容を直接書き換えることとします。
- 正常終了の場合、戻り値 0 を返します。
- 解が出ない場合など、処理が続行できない場合は、関数の戻り値として -1 を返します。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、

のように、枠で示されています。ただし、枠の大きさは、おおよそコード 1 行を表すものとします。

ヒント ピボット操作により、行列 a, b, inv_a に対して行の入れ替えが行われます。

ヒント fabs() は浮動小数点引数の絶対値を返す関数です。

ヒント 数値表現 1e-10 は浮動小数点で、1.0×10^{-10} のことです。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
#define SWAP(a,b) {temp=(a);(a)=(b);(b)=temp;}

/* 行列の最大次数 */
#define NEQ 16

typedef double ElemType;


/* ガウス・ジョルダン法で演算を行う関数 */
int gauss_jordan(
    int n,                      /* 行列の次数 */
    ElemType a[NEQ][NEQ],       /* 係数（n次正方行列）配列 */
    ElemType b[NEQ],            /* 右辺値（列ベクトル）配列 */
    ElemType inv_a[NEQ][NEQ]    /* 逆行列の結果を格納する配列 */
)
{
    /* カウンタ */
    int icol, i, j;
    /* SWAP 作業用変数 */
    ElemType temp;

    /* 逆行列演算用の単位行列を作成する */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            inv_a[i][j] = (i == j) ? [                                ];
        }
```

```
    }

    /* 各列を順番に処理する */
    for (icol = 0; icol < n; icol++) {
        int pivot_row;

        /* 列の最大値の成分を探索し、ピボット行候補とする */
        {
            ElemType big = 0.0;
            for (i = icol; i < n; i++) {
                if (fabs(a[i][icol]) > big) {
                    big = [                              ];
                    pivot_row = [                              ];
                }
            }
            /* ピボット成分が、演算丸め誤差の分を考慮して、
               きわめて 0 に近ければエラー終了とする */
            if (fabs(big) < 1e-10) {
                return -1;
            }
        }

        /* ピボット行の入れ替え処理 */
        if (icol != pivot_row) {
            for (i = 0; i < n; i++) {
                SWAP(a[icol][i], [                              ]);
                SWAP(inv_a[icol][i], [                              ]);
```

```
            }
            SWAP(b[icol], [redacted]);
        }

        /* 対角成分を 1 にする */
        {
            ElemType inv_pivot = 1.0 / [redacted];
            for (i = 0; i < n; i++) {
                a[icol][i] *= inv_pivot;
                inv_a[icol][i] *= inv_pivot;
            }
            b[icol] *= inv_pivot;
        }

        /* ピボット列の対角成分以外を 0 にする */
        for (i = 0; i < n; i++) {
            if (i != icol) {
                ElemType refvalue = a[i][icol];
                for (j = 0; j < n; j++) {
                    a[i][j] -= refvalue * [redacted];
                    inv_a[i][j] -= refvalue * [redacted];
                }
                b[i] -= refvalue * [redacted];
            }
        }
    }
```

```
    return 0;
}



/* 使用する行列の型宣言 */
typedef struct {
    /* 正方行列の要素数 */
    int n;
    /* 係数（正方行列） */
    ElemType mat[NEQ][NEQ];
    /* 右辺（列ベクトル） */
    ElemType vec[NEQ];
    /* 逆行列演算 */
    ElemType rev[NEQ][NEQ];
} Matrix;


/* 行列データの定義 */
Matrix testData[] = {
    {3, {{2, 1, 2}, {3, 2, 0}, {1, 2, 2}}, {6, -1, 3}},
    {3, {{2, 1, 2}, {2, 4, 4}, {1, 2, 2}}, {6, -1, 3}},
    {4, {{2,2,3,4}, {1,0,2,1}, {-1,2,1,2}, {2,1,4,0}}, {6,3,-1,0}}
};


/* データ数 */
```

```
const int dataSize = sizeof (testData) / sizeof (Matrix);

int main()
{
    int idx;

    for (idx = 0; idx < dataSize; idx++) {
        /* カウンタ */
        int i, j;

        /* 行列データ */
        Matrix data = testData[idx];

        /* ガウス・ジョルダン法による連立方程式の解法 */
        int retcode = gauss_jordan(data.n, data.mat, data.vec, data.rev);
        printf("retcode: %d¥n", retcode);

        if (retcode == 0) {
            /* 行列成分値の表示 */
            for (i = 0; i < data.n; i++) {
                for(j = 0; j < data.n; j++) {
                    printf("%8.3f", data.mat[i][j]);
                }
                printf("  %8.3f  ", data.vec[i]);
                for(j = 0; j < data.n; j++) {
                    printf("%8.3f", data.rev[i][j]);
                }
```

```
                printf("¥n");
            }
        }
    }

    return 0;
}
```

固有値問題

対称行列の Jacobi 変換

ヤコビ法とは、実数の対称行列 A の固有値、固有ベクトルを簡単な反復で求めるものです。 適当な直交行列 P_1 により、 $A \leftarrow P_1^T A P_1$ と変換し、A のどれか一つの非対角要素を 0 にします。 次に、また別の適当な直交行列 P_2 により、別の非対角要素を 0 にするということを繰り返すと、前に 0 にした要素は 0 ではなくなりますが、操作を反復して非対角要素をほぼ 0 に近づけることにより、対角化

$$P_m^T \ldots P_2^T P_1^T A P_1 P_2 \ldots P_m = \Lambda$$

を求める手法です。

A の非対角要素のなかで一番大きなものを、a_{pq}（ ただし p < q ） とすると、直交行列 P を次のように考えます。

$$\begin{vmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & cos\theta & \cdots & sin\theta & 0 \\ 0 & 0 & \cdots & 0 & 0 \\ 0 & -sin\theta & \cdots & cos\theta & 0 \\ 0 & 0 & \cdots & 0 & 1 \end{vmatrix}$$

ここで P は、単位行列の pp成分を cos に、pq 成分を sin に、 qp 成分を -sin に、qq 成分を cos に置き換えたものです。すると、P^T は、pq 成分と qp 成分が入れ替わったものとなり、P^TAP を計算すると、次のようになります。

$$a'_{ij} = a_{ij} \quad (i, j \neq p, q)$$
$$a'_{ip} = a'_{pi} = cos\theta\, a_{ip} - sin\theta\, a_{iq} \quad (i \neq p, q)$$
$$a'_{iq} = a'_{qi} = sin\theta\, a_{ip} + cos\theta\, a_{iq} \quad (i \neq p, q)$$
$$a'_{pp} = cos\theta\,(cos\theta\, a_{pp} - sin\theta\, a_{pq}) - sin\theta\,(cos\theta\, a_{pq} - sin\theta\, a_{qq})$$
$$a'_{pq} = a'_{qp} = sin\theta\,(cos\theta\, a_{pp} - sin\theta\, a_{pq}) + cos\theta\,(cos\theta\, a_{pq} - sin\theta\, a_{qq})$$
$$a'_{qq} = sin\theta\,(sin\theta\, a_{pp} + cos\theta\, a_{pq}) + cos\theta\,(sin\theta\, a_{pq} + cos\theta\, a_{qq})$$

ここで、非対角要素 a'_{pq}（ 対称行列なので、 $= a'_{qp}$ ） を 0 にしたいので、

$$sin\theta\, cos\theta\,(a_{pp} - a_{qq}) + (sin^2\theta - cos^2\theta)\, a_{pq} = 0$$
$$\frac{1}{2} sin2\theta\,(a_{pp} - a_{qq}) + cos2\theta\, a_{pq} = 0$$

を満たす θ を見つければよいということになります。ここで、

$$\alpha = \frac{1}{2}(a_{pp} - a_{qq}),\ \beta = -a_{pq}$$

とおくと、

$$\alpha \sin 2\theta = \beta \cos 2\theta$$
$$\alpha^2 \sin^2 2\theta = \alpha^2 (1 - \cos^2 2\theta) = \beta^2 \cos^2 2\theta$$
$$\alpha^2 = (\alpha^2 + \beta^2) \cos^2 2\theta$$

となるので、 $\cos^2 2\theta$ の平方根を求めて、γ とおきます。

$$\cos 2\theta = \frac{|\alpha|}{\sqrt{\alpha^2 + \beta^2}} = \gamma$$

すると、半角の公式

$$\sin^2 \theta = \frac{1}{2}(1 - \cos 2\theta)$$
$$\cos^2 \theta = \frac{1}{2}(1 + \cos 2\theta)$$

から、

$$\cos\theta = ((1+\gamma)/2)^{1/2}$$
$$\sin\theta = \pm((1-\gamma)/2)^{1/2}$$ ※ ($\alpha\beta$ の符号が負の場合、負となる)

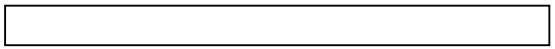
と導かれます。ただし、$\alpha\beta$ の値が負の場合、$\sin\theta$ の値も負となるので、注意して下さい。

演習課題 6-3

ヤコビ法により、実対称行列を引数にとり、その固有値と固有ベクトルを求めるプログラム `Jacobi()` を完成させて下さい。

- ファイル名を `kadai_6_3.c` として下さい。
- 行列の次数 `n` 、実対称行列 `A` と固有ベクトル格納領域 `X` を引数として受け取ります。
- 行列 `A[0..n-1][0..n-1]` のすべての固有値および固有ベクトルを計算します。
- `A` の対角要素に固有値が計算されるものとします。
- `X` には、固有ベクトルを列に並べて格納します。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、



のように、枠で示されています。ただし、枠の大きさは、おおよそコード `1` 行を表すものとします。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* 行列の最大次数 */
#define NEQ 16

typedef double ElemType;


int Jacobi(
    int n,                      /* 行列の次数 */
    ElemType A[NEQ][NEQ],       /* 入力値 （正方対称行列）配列 */
    ElemType X[NEQ][NEQ]        /* 固有ベクトル格納配列 */
)
{
    const int maxloop   = 1000;     /* 繰り返し処理上限回数 */
    const double eps  = 1.0e-6;     /* 収束判定の閾値 */
    ElemType A2[NEQ][NEQ];          /* 回転作業用行列 */
    ElemType X2[NEQ][NEQ];          /* 固有ベクトル作業配列 */

    int i, j, cnt = 0;      /* カウンタ */

    /* 固有ベクトル計算用単位行列のセット */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
```

```
            X[i][j] = (i == j) ? 1.0 : 0.0;
        }
    }

    /* 繰り返し処理 */
    for (cnt = 0; cnt < maxloop; cnt++) {
        /* p, q 変数 */
        int p = 0, q = 0;
        /* sin, cos 変数 */
        ElemType sn, cs;

        /* 非対角要素の最大要素の探索 */
        ElemType max = 0.0;
        for (i = 0; i < n; i++) {
            for (j = 0; j < n; j++) {
                if (i != j) {
                    if (fabs(A[i][j]) > max) {
                        /* 最大値、位置インデックスの保持 (p < q) */
                        max = fabs(A[i][j]);
                        q   = [                                    ];
                        p   = [                                    ];
                    }
                }
            }
        }
        /* 収束判定閾値より小さくなれば処理終了 */
        if (max < eps)
```

```
        [                                    ];

    {
        ElemType alpha = 0.5 * (A[p][p] - A[q][q]);
        ElemType beta = -A[p][q];
        ElemType gamma = [                                    ];
        cs = sqrt(0.5 * (1.0 + gamma));
        sn = sqrt(0.5 * (1.0 - gamma));
        if (alpha * beta < 0)
            sn = -sn;
    }

    /* 行列に回転演算を施して、要素を 0 にする処理 */
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if ((i == p || i == q) && (j == p || j == q)) {
                if (i == p && j == p)
                    /* App の計算 */
                    A2[i][j] = [                                    ];
                else if (i == q && j == q)
                    /* Aqq の計算 */
                    A2[i][j] = [                                    ];
                else
                    /* Apq, Aqp の計算 */
                    A2[i][j] = [                                    ];
            }
            else if (i == p || j == p) {
```

```
            /* Api, Aip の要素の計算 */
            int k = (j == p) ? i : j;
            A2[i][j] = [          ];
        }
        else if (i == q || j == q) {
            /* Aqi, Aiq の要素の計算 */
            int k = (j == q) ? i : j;
            A2[i][j] = [          ];
        }
        else
            A2[i][j] = A[i][j];
    }
}

/* 固有ベクトル値の計算  */
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (j == p)
            X2[i][p] = X[i][p] * cs - X[i][q] * sn;
        else if (j == q)
            X2[i][q] = X[i][q] * cs + X[i][p] * sn;
        else
            X2[i][j] = X[i][j];
    }
}
/* 作業領域からの書き戻し */
for (i = 0; i < n; i++) {
```

```
            for (j = 0; j < n; j++) {
                A[i][j] = [                                    ];
                X[i][j] = [                                    ];
            }
        }
    }

    return -1;
}



/* 使用する行列の型宣言 */
typedef struct {
    /* 正方行列の要素数 */
    int n;
    /* 正方行列 （対称行列） */
    ElemType mat[NEQ][NEQ];
    /* 固有ベクトル （列ベクトル） */
    ElemType vec[NEQ][NEQ];
} Matrix;


/* 行列データの定義 */
Matrix testData[] = {
    {3, {{1, 3, 0}, {3, 2, 1}, {0, 1, 3}}},
    {4, {{0,2,1,2}, {2,3,0,0}, {1,0,0,1}, {2,0,1,0}}},
```

```
    {4, {{0,1,0,1}, {1,2,0,1}, {0,0,0,0}, {1,1,0,0}}}
};


/* データ数 */
const int dataSize = sizeof (testData) / sizeof (Matrix);


int main()
{
    int idx;

    for (idx = 0; idx < dataSize; idx++) {
        /* カウンタ */
        int i, j;

        /* 行列データ */
        Matrix data = testData[idx];

        printf("--------------¥n");
        /* 入力行列の表示 */
        printf("Matrix¥n");
        for (i = 0; i < data.n; i++) {
            for(j = 0; j < data.n; j++) {
                printf("%8f ", data.mat[i][j]);
            }
            printf("¥n");
```

```
        }

        /* Jacobi 法による、固有値・固有ベクトルの導出 */
        int retcode = Jacobi(data.n, data.mat, data.vec);
        printf("retcode: %d¥n", retcode);

        if (retcode < 0)
            printf("Matrix Does Not Converge.¥n");
        else if (retcode == 0) {
            /* 対角要素の表示 */
            printf("Eigenvalue¥n");
            for (i = 0; i < data.n; i++)
                printf(" %8f ", data.mat[i][i]);

            /* 列ベクトルの表示 */
            printf("¥nEigenvector¥n");
            for (i = 0; i < data.n; i++) {
                for(j = 0; j < data.n; j++) {
                    printf("%8f ", data.vec[i][j]);
                }
                printf("¥n");
            }
        }
    }

    return 0;
}
```

フーリエ変換

高速フーリエ変換

離散フーリエ変換 (DFT) は、複素数 ω を

$$\omega = e^{-2\pi i/N}$$

と定義すると、次のように書くことができます。

$$H_n = \sum_{k=0}^{N-1} \omega^{nk} h_k$$

高速フーリエ変換 (FFT) は、上の離散フーリエ変換を高速に求めるアルゴリズムのことを言います。

まずは、高速フーリエ変換ではなく離散フーリエ変換を見てみることにします。離散フーリエ変換を行う関数 `dft()` を以下のように実装しました。

```
#include <math.h>

const int N_MAX = 64;

void dft(int N, float arr_r[], float arr_i[])
{
```

```
        /* 演算結果格納配列 */
        float res_r[N_MAX], res_i[N_MAX];

        /* カウンタ変数 */
        int n, k;

        /* Hn の演算 */
        for (n = 0; n < N; n++) {
                float val_r = 0.0;      /* 実数部 */
                float val_i = 0.0;      /* 虚数部 */

                /* Σ の演算 */
                for (k = 0; k < N; k++) {
                        /* w = e -2πi nk/N */
                        float wr =  cos(2 * M_PI * k * n / N);  /* 実数部 */
                        float wi = -sin(2 * M_PI * k * n / N);  /* 虚数部 */
                        /* Hn = wnk hk */
                        val_r += arr_r[k] * wr - arr_i[k] * wi; /* 実数部 */
                        val_i += arr_r[k] * wi + arr_i[k] * wr; /* 虚数部 */
                }
                /* Hn の格納 */
                res_r[n] = val_r;
                res_i[n] = val_i;
        }

        /* 結果の書き戻し */
        for (n = 0; n < N; n++) {
                arr_r[n] = res_r[n];
                arr_i[n] = res_i[n];
        }
}
```

単純な離散フーリエ変換では、要素数 N に対し、N 個の要素との積和をとるため、$O(n^2)$ の計算量を要することになります。よく用いられている高速フーリエ変換では、要素を 2 分割していくことで、計算量を $O(n \log_2 n)$ に近づけていきます。そのため、要素数が 2 の冪乗でなければならない、という制約があります。

本稿では、高速フーリエ変換の原理を詳しく説明することはできませんので、必要であれば関連する資料等により高速フーリエ変換の知識を補ってください。

高速フーリエ変換の基本的なアイディアは、ある系列の DFT に対し、逐次的に小さな部分系列に分解する、というもので、分解の仕方により、時間間引き、周波数間引き、というアルゴリズムがあります。分解の仕組みは、長さ N の離散フーリエ変換が、長さ N/2 の離散フーリエ変換同士の和で表される、ということにあります。

ここでは、周波数間引きのやり方を考えてみます。離散フーリエ変換を次のように定義し、

$$H_n = \sum_{k=0}^{N-1} e^{-2\pi i n k/N} h_k$$

$0 \leqq m \leqq N/2 - 1$ とおいて、フーリエ変換で得られる周波数成分を、偶数の要素 H_{2m} と奇数の要素 H_{2m+1} 、に分けます。

周期性に着目して、k と $k + N/2$ のとき、$e^{-2\pi i(2m)k/N}$ が同値になることを利用すると、

$$H_{2m} = \sum_{k=0}^{N-1} e^{-2\pi i(2m)k/N} h_k$$

$$= \sum_{k=0}^{N/2-1} e^{-2\pi i(2m)k/N} h_k + \sum_{k=N/2}^{N-1} e^{-2\pi i(2m)k/N} h_k$$

$$= \sum_{k=0}^{N/2-1} e^{-2\pi imk/(N/2)} (h_k + h_{k+N/2})$$

同様に、奇数についても、$e^{-2\pi i(2m+1)k/N} = -e^{-2\pi i(2m+1)(k+N/2)/N}$ に着目すると、

$$H_{2m+1} = \sum_{k=0}^{N-1} e^{-2\pi i(2m+1)k/N} h_k$$

$$= \sum_{k=0}^{N/2-1} e^{-2\pi i(2m+1)k/N} h_k + \sum_{k=N/2}^{N-1} e^{-2\pi i(2m+1)k/N} h_k$$

$$= \sum_{k=0}^{N/2-1} e^{-2\pi i(2m+1)k/N} (h_k - h_{k+N/2})$$

$$= e^{-2\pi ik/N} \sum_{k=0}^{N/2-1} e^{-2\pi i(2m)k/N} (h_k - h_{k+N/2})$$

$$= e^{-2\pi ik/N} \sum_{k=0}^{N/2-1} e^{-2\pi imk/(N/2)} (h_k - h_{k+N/2})$$

ということで、それぞれ、長さ N/2 系列の DFT に帰着されます。 このとき、 $h_k + h_{k+N/2}$ からなる配列を偶数成分として、 $e^{-2\pi ik/N} (h_k - h_{k+N/2})$ からなる配列を奇数成分として、再帰的にフーリエ変換を求め、結果を再構成することで、全体のフーリエ変換が得られることになります。

再帰的に分解を繰り返すことで、N/4、N/8、という具合に計算量を減らしていくことができ、それ以上分解ができなくなる長さ 1 になるまで続けられます。長さ 2 での演算を行列とベクトルの形式で表すと、次のようになります。

$$\begin{vmatrix} H_0 \\ H_1 \end{vmatrix} = \begin{vmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^2 \end{vmatrix} \begin{vmatrix} h_0 \\ h_1 \end{vmatrix}$$

$$= \begin{vmatrix} 1 & 1 \\ 1 & -1 \end{vmatrix} \begin{vmatrix} h_0 \\ h_1 \end{vmatrix}$$

演習課題 6-4

再帰的に実装された高速フーリエ変換プログラムを完成させて下さい。

- ファイル名を kadai_6_4.c として下さい。

下にある演習プログラムは、ところどころコードを埋める箇所があります。空欄部分は、

[ ]

のように、枠で示されています。ただし、枠の大きさは、おおよそコード 1 行を表すものとします。

ヒント オイラーの公式 $e^{ix} = \cos x + i \sin x$ を使います。

```
#include <stdio.h>
#include <math.h>

const int N_MAX = 64;



void fft(int n, float arr_r[], float arr_i[])
{
    float ev_r[N_MAX], ev_i[N_MAX]; /* 偶数成分 FFT 部分系列 */
    float od_r[N_MAX], od_i[N_MAX]; /* 奇数成分 FFT 部分系列 */

    /* 回転因子の角度 */
    float theta = 2 * M_PI / n;
    /* カウンタ */
    int nh, j;

    if (n <= 1)
        return;

    /* N/2 */
```

```
nh = n / 2;
for (              ) {
    float xr, xi, wr, wi;
    /* 偶数成分を計算し、格納する */
    ev_r[j] = arr_r[j] + arr_r[nh + j];
    ev_i[j] = arr_i[j] + arr_i[nh + j];
    /* 奇数成分を計算する */
    xr =             ;
    xi =             ;
    /* 係数を求める */
    wr =             ;
    wi =             ;
    /* 奇数成分に係数を乗じて、格納する */
    od_r[j] = xr * wr - xi * wi;
    od_i[j] = xi * wr + xr * wi;
}

/* 偶数成分の FFT を帰納的に求める */
fft(             );

/* 奇数成分の FFT を帰納的に求める */
fft(             );

/* 得られた FFT 部分系列を書き戻す */
for (j = 0; j < nh; j++) {
    arr_r[2 * j] = ev_r[j];         /* 偶数成分 実数部分 */
    arr_i[2 * j] = ev_i[j];         /* 偶数成分 虚数部分 */
```

```
            [          ];      /* 奇数成分 実数部分 */
            [          ];      /* 奇数成分 虚数部分 */
        }
    }



int main()
{
    float x[N_MAX], y[N_MAX];
    const int n = 64;

    int i;
    for (i = 0; i < n; i++) {
        x[i] = sin(4 * 2 * M_PI * i / n);
        y[i] = 0;
    }

    /* FFT */
    fft(n, x, y);

    for (i = 0; i < n; i++) {
        printf("%d %8f %8f¥n", i, x[i], y[i]);
    }

    return 0;
}
```

演習問題索引