I1 実験考察レポート

井上理璃子

2025 年 5 月 20 日

1　1 日目課題

1.1　本課題 1.7

■p00.c

```
1 // added header files
2 #include <stdio.h>
3 #include <stdlib.h>
```

ヘッダーファイルを追加した。

■p01.c

```
1  #include <stdio.h>
2  #include <math.h> // added math.h
3
4  // added other functions before main function
5  double cos2(double x)
6  {
7    double c = cos(x);
8    return c * c;
9  }
10
11 double sin2(double x)
12 {
13   double s = sin(x);
14   return s * s;
15 }
16
17
18 int main()
19 {
20   int i;
21   for (i = 0; i < 100; i++) {
22     double y = cos2((double)i) + sin2((double)i);
23     printf("cos^2(%d)+sin^2(%d) = %f\n", i, i, y);
24   }
```

```
25    return 0;
26  }
```

sin や cos などの関数を使うために math.h ヘッダーファイルを追加した。また、main 関数の後に自分で定義した関数を宣言すると main 関数内で呼び出せないので、cos2, sin2 を main 関数の前に持ってきた。

■p02.c

```c
// added header files
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char ** argv)
{
  double x = atof(argv[1]);
  double y = atof(argv[2]);
  printf("%f\n", x * y);
  return 0;
}
```

ヘッダーファイルを追加した。

■p03.c

```c
// 中略
vect3 * mk_point(double x, double y, double z)
{
  // added memory allocation
  vect3 * p = (vect3 *)malloc(sizeof(vect3));
  if (p == NULL) {
    fprintf(stderr, "メモリの確保に失敗しました\n");
    exit(1);
  }
  p->x = x;
  p->y = y;
  p->z = z;
  return p;
}

int main(int argc, char ** argv)
{
  vect3 * A = mk_point(atof(argv[1]), atof(argv[2]), atof(argv[3]));
  vect3 * B = mk_point(atof(argv[4]), atof(argv[5]), atof(argv[6]));
  double a = angle(A, B);
  printf("%f\n", a);

  // added memory release
  free(A);
  free(B);
  return 0;
```

```
27 }
```

vect3 関数内で p ポインタを宣言する時に、malloc を用いてメモリを確保した。また、main 関数内で A, B を使い終わった後に free 関数を用いてメモリを解放した。p04.c も同様に編集した。

■p05.c

```c
#include <assert.h>
#include <stdio.h>
int main()
{
  FILE * fp = fopen("p05.c", "rb"); // edited O to 0
  char buf[100];
  while (1) {
    int n = fread(buf, 1, 100, fp);
    if (n == 0) break;
    fwrite(buf, 1, n, stdout);
  }

  fclose(fp);
  return 0;
}
```

ファイル名の”pO5.c”の O はアルファベットなので、”p05.c”に変更した。

■p06.c

```c
// added header files
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>

int main()
{
  int fd = open("p06.c", O_RDONLY); // modified O to 0

  char buf[100];
  while (1) {
    int n = read(fd, buf, 100);
    if (n == 0) break;
    write(1, buf, n);
  }

  close(fd); // added close
  return 0;
```

ファイル名の O を 0 に変更し、ファイルデータを使用した後に close によって終了した。

■p07.c

```c
int main(int argc, char ** argv)
{
```

```
  /* try to read the whole contents of
     the file into s */
  FILE * fp = fopen(argv[1], "rb");
  /* s is initially small (10 bytes) */
  int begin = 0;
  int end = 10;
  char * s = (char *)malloc(end);
  while (1) {
    /* read as many bytes as s can store ((end - begin) bytes) */
    int r = fread(s + begin, 1, end - begin, fp);
    if (r < end - begin) {
      /* reached end of file */
      end = begin + r;
      break;
    }
    /* s is full and we have not reached end of file.
       we extend s by 10 bytes */
    begin = end;
    end += 10;
    char * t = (char *)malloc(end); /* new buffer */
    assert(t);
    bcopy(s, t, end);
    free(s);
    s = t; // added buffer update
  }
  /* print s from end to beginning */
  int i;
  for (i = end - 1; i >= 0; i--) {
    putchar(s[i]);
  }
  return 0;
}
```

s = t としてバッファーを更新する処理を加えた。これを省くと、新しく割り当てたメモリ（t）が使用されず、古いメモリ（s）が解放された後も無効なメモリを参照し続けることになる。

■p08.c

```
double number()
{
  errno = 0;
  // modified endptr to pointer and passed the address to strtod
  char * endptr;
  double x = strtod(p, &endptr);
  p = endptr;
  if (errno) {
    perror("strtod:");
    syntax_error();
```

```
  }
  return x;
}
```

endptr の修正をした。strtod 関数の第 2 引数は char 型（ポインタへのポインタ）であり、アドレスを渡す必要がある。

```
double H_expression()
{

  switch (*p) {
  case '0' ... '9': {
    return number();
  }
  case '(': {
    p++; // added pointer advance
    double x = E_expression();
    if (*p == ')') {
      p++; // added pointer advance
      return x;
    } else {
      syntax_error();
    }
  }
  default:
    syntax_error();
  }
  return 0; // added return 0
}
```

p++ を二箇所に追加した。開きカッコ'(' と閉じカッコ')' を読み取った後、ポインタを次の文字に進める必要がある。この処理がなければ同じ文字を繰り返し読むことになり、無限ループやパース失敗を引き起こす。

```
double E_expression()
{
  double x = F_expression();
  while (1) {
    if (*p == '+') {
      p++;
      x += F_expression(); // modified = and +
    } else if (*p == '-') {
      p++;
      x -= F_expression(); // modified = and -
    } else {
      return x;
    }
  }
```

```
15 }
```

=+,=-を +=,-=に修正した。

2　2 日目課題

■本課題 2.8

```
1  int main(int argc, char *argv[]) {
2      int fd = open(argv[1], O_RDONLY);
3      if (fd == -1) {
4          perror("open");
5          exit(1);
6      }
7      unsigned char byte;
8      int count = 0;
9      while (1) {
10         int n = read(fd, &byte, 1);
11         if (n == -1) {
12             perror("read");
13             exit(1);
14         }
15         if (n == 0) {
16             break;
17         }
18         printf("%d %d\n", count, byte);
19         count++;
20     }
21
22     close(fd);
23     return 0;
24 }
```

このプログラムはコマンドラインで指定されたファイルを open で開き、何バイト目がなんというバイトだったのかを 2 カラムで表示するプログラムである、read 関数を用いて、1 バイトずつ byte に読み込み、printf を用いて標準出力している。

```
% ./read_data my_data
  0 0
  1 1
  2 2
  3 3
  4 4
  5 5
  ...
```

準備課題 2.1 で作成した my_data を読み込ませると上のようになり、プログラムが正しく動作していること

が確認できた。

■本課題 2.13　与えられた条件で音データを”kadai2_13.raw”というファイルに録音するために、以下のコマンドを実行した。

```
% rec -b 8 -c 1 -e unsigned-integer -r 44100 kadai2_13.raw
```

その後、本課題 2.8 で作成した、read_data を下記のように用いて、バイナリデータをテキストデータに変換した。

```
% ./read_data kadai2_13.raw  > kadai2_13.txt
```

最後に gnuplot で波形を確認するために、下記のコマンドを実行した。

```
% gnuplot
gnuplot> plot "kadai2_13.txt" with linespoints
```

得られた波形データが下のようである。

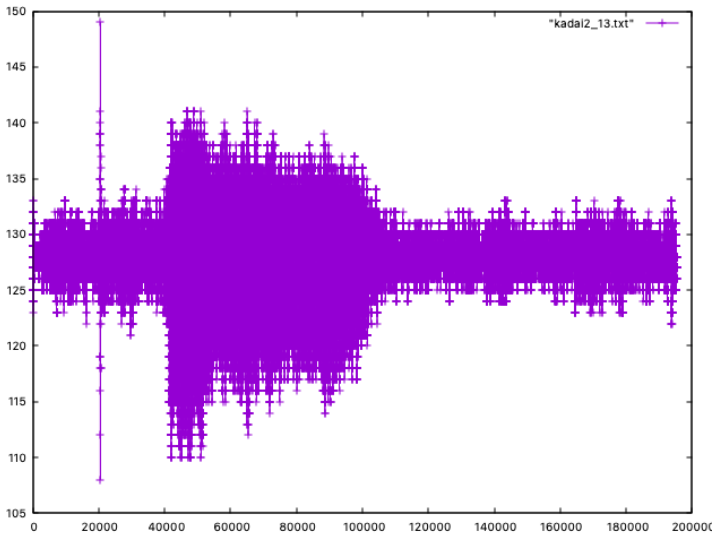

図 1: 本課題 2.13 の音声波形.

■本課題 2.14

```c
#define PI 3.14159265358979323846

int main(int argc, char *argv[]) {
    double A = atof(argv[1]);
    double f = atof(argv[2]);
    int n = atoi(argv[3]);

    // ビット符号付き整数の最大値 16
    // 振幅を正規化するために使用
    const double MAX_AMP = 32767.0;

```

```
12      // 各点を計算してデータとして出力 RAW
13      for (int i = 0; i < n; i++) {
14          // 時間（秒）
15          double t = (double)i / 44100.0;
16
17          // サイン波計算: x = A * sinπ (2ft)
18          double sample = A * sin(2.0 * PI * f * t);
19
20          // ビット符号付き整数に変換 16
21          short output;
22
23          // 振幅が最大値を超えないように調整
24          if (sample > MAX_AMP) {
25              output = 32767;
26          } else if (sample < -MAX_AMP) {
27              output = -32768;
28          } else {
29              output = (short)sample;
30          }
31
32          // バイナリデータとして出力
33          write(1, &output, sizeof(short));
34      }
35
36      return 0;
37  }
```

このプログラムでは、コマンドライン引数から振幅、周波数、サンプル数を取得した後、各サンプルの時間位置をサンプリングレート 44100Hz で計算する。次に正弦波の数式「A * sin(2 π ft)」を用いて波形を生成し、16 ビット整数の範囲（-32768〜32767）に収まるよう値を調整する。生成された波形データは write 関数を使って標準出力にバイナリデータとして書き出される。コンパイル後、

```
% ./sin 10000 440 88200 | play -t raw -b 16 -c 1 -e s -r 44100 -
```

のコマンドを実行したところ、2 秒分の音が鳴ることを確認できた。

■本課題 2.15

```
1  // read_data2.c
2  int main(int argc, char *argv[]) {
3      int fp = open(argv[1], O_RDONLY);
4      if (fp == -1) {
5          perror("open");
6          exit(1);
7      }
8      short data;
9      int count = 0;
10     while (1) {
```

```
11          int n = read(fp, &data, 2);
12          if (n == -1) {
13              perror("read");
14              exit(1);
15          }
16          if (n == 0) {
17              break;
18          }
19          printf("%d %d\n", count, data);
20          count++;
21      }
22
23      close(fp);
24      return 0;
25  }
```

16 ビットごとに一つの符号付き整数が並んでいるとみなして表示するために、read 関数の第三引数を 2 バイトに変更した。

```
% ./read_data2 sin.raw > sin.txt
% gnuplot
gnuplot> plot "sin.txt" with linespoints
gnuplot> set xrange [0:300]
gnuplot> replot
```

上のようにコマンドを実行し、gnuplot を用いて正弦波のデータを可視化した結果、図 2 のような波形が得られた。

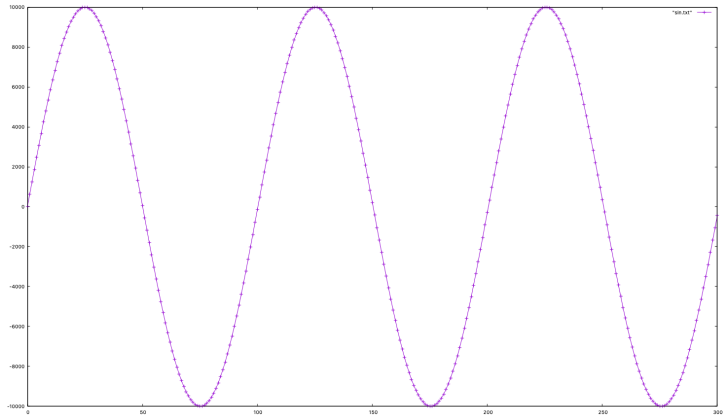

図 2: 本課題 2.15 の正弦波形.

■選択課題 2.16

```
1  // doremi.c
2  #define PI 3.14159265358979323846
3  #define SAMPLE_RATE 44100
```

```c
#define SAMPLES_PER_NOTE 13230

int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "使用法: %s 振幅 A 音の数 n\n", argv[0]);
        return 1;
    }

    double A = atof(argv[1]);
    int n = atoi(argv[2]);

    double f = 261.63;

    const double MAX_AMP = 32767.0;
    int count = 0;
    // 音分の音階を生成 n
    for (int note = 0; note < n; note++) {
        // 現在の音階のインデックス（ドレミファソラシドの周期）
        int note_index = note % 8;

        for (int i = 0; i < SAMPLES_PER_NOTE; i++) {
            double t = (double)i / SAMPLE_RATE;

            double sample = A * sin(2.0 * PI * f * t);

            // ビット符号付き整数に変換 16
            short output;

            // 振幅が最大値を超えないように調整
            if (sample > MAX_AMP) {
                output = 32767;
            } else if (sample < -MAX_AMP) {
                output = -32768;
            } else {
                output = (short)sample;
            }

            // バイナリデータとして出力
            write(1, &output, sizeof(short));
        }

        // 周波数を変更
        // ミとファ、シとドの間隔は半音で２の乗倍 1/12
        if (count % 7 == 2 || count % 7 == 6) {
            f = pow(2.0, 1.0/12.0) * f;
        } else {
```

```
50              f = pow(2.0, 1.0/6.0) * f;
51          }
52          count++;
53      }
54
55      return 0;
56  }
```

doremi.c は適当なドの音から始めて、ドレミファソラシド... と n 音順に出力するプログラムである。ドの音の周波数を基準として、各音の処理の最後で周波数の更新を行なった。ミとファの間、シとドの間は半音なので、$2^{1/12}$ 倍、それ以外は全音上がるので、$2^{1/6}$ 倍となる。

```
% ./doremi 10000 88200 | play -t raw -b 16 -c 1 -e s -r 44100 -
```

このコマンドを用いて実際に音を聞くことができた。

3　4 日目課題

■本課題 4.1

```
1  // downsample.c
2  int main(int argc, char *argv[]) {
3      if (argc != 2) {
4          fprintf(stderr, "使用法: %s ダウンサンプル率\n", argv[0]);
5          exit(1);
6      }
7      int downsample_rate = atoi(argv[1]);
8      if (downsample_rate <= 0) {
9          fprintf(stderr, "ダウンサンプル率は正の整数である必要があります\n");
10         exit(1);
11     }
12     short data;
13     int count = 0;
14
15     while (1) {
16         int n = read(0, &data, sizeof(short));
17         if (n == -1) {
18             perror("read");
19             exit(1);
20         }
21         if (n == 0) {
22             break;
23         }
24         if (count % downsample_rate == 0) {
25             write(1, &data, sizeof(short));
26         }
27         count++;
28     }
```

```
29      return 0;
30  }
```

上記が 16 ビットのモノラルの raw データを標準入力から受け取って、与えられた割合で間引いて標準出力に出力する downsample.c のプログラムである。データを読み込むごとにカウントをインクリメントしていき、カウントを downsample_rate で割ったあまりが 0 の時のみ、標準出力に書き込むようにしている。

■本課題 4.3

```
% ./sin 10000 3528 88200 > ../Day4/kadai4_3.raw
% ./read_data2 ../Day4/kadai4_3.raw > ../Day4/kadai4_3.txt
% gnuplot
> plot "kadai4_3.txt" with linespoints, "kadai4_3.txt" every 10 with line
> set xrange [0:50]
> replot
```

上記のコマンドを用いて、gnuplot で波形を確認した結果、図 3 のようなデータが得られた。エイリアシングが生じていることが確認できる。

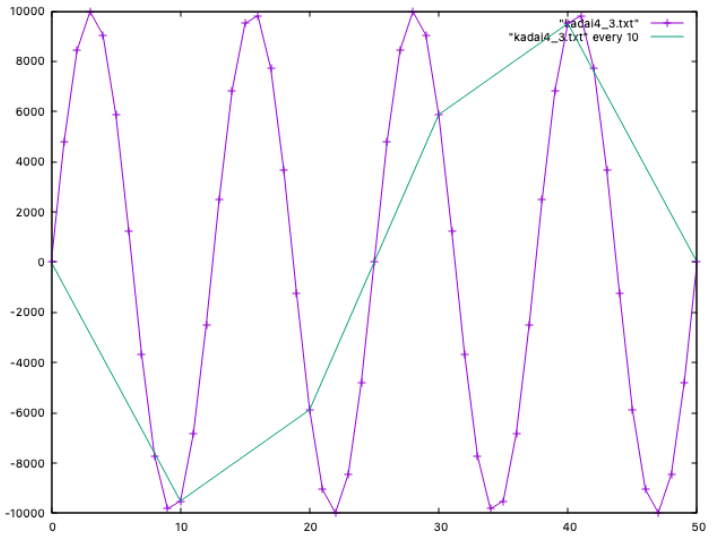

図 3: 本課題 4.3 の波形.

■本課題 4.5　以下は bandpass.c の main 関数部分である。

```
1  int main(int argc, char ** argv) {
2    (void)argc;
3    long sample_rate = 44100;
4    long n = atol(argv[1]);
5    long low_cut = atol(argv[2]);
6    long high_cut = atol(argv[3]);
7
8    if (!pow2check(n)) {
```

```
9       fprintf(stderr, "error : n (%ld) not a power of two\n", n);
10      exit(1);
11    }
12    FILE * wp = fopen("fft.dat", "wb");
13    if (wp == NULL) die("fopen");
14    sample_t * buf = calloc(sizeof(sample_t), n);
15    complex double * X = calloc(sizeof(complex double), n);
16    complex double * Y = calloc(sizeof(complex double), n);
17    while (1) {
18      /* 標準入力から個標本を読む n */
19      ssize_t m = read_n(0, n * sizeof(sample_t), buf);
20      if (m == 0) break;
21      /* 複素数の配列に変換 */
22      sample_to_complex(buf, X, n);
23      /* FFT -> Y */
24      fft(X, Y, n);
25
26      for (long i = 0; i < n; i++) {
27        long freq_idx = (i <= n / 2) ? i : i - n;
28        long freq = freq_idx * sample_rate / n;
29        if (freq < low_cut || freq > high_cut) {
30          Y[i] = 0;
31        }
32      }
33
34      print_complex(wp, Y, n);
35      fprintf(wp, "----------------\n");
36
37      /* IFFT -> Z */
38      ifft(Y, X, n);
39      /* 標本の配列に変換 */
40      complex_to_sample(X, buf, n);
41      /* 標準出力へ出力 */
42      write_n(1, m, buf);
43    }
44    fclose(wp);
45    return 0;
46  }
```

26-32 行目が bandpass フィルタの実装であり、ここではまず、配列インデックス i から実際の周波数インデックス freq_idx への変換を行っている。FFT 結果は 0 から n-1 までだが、周波数表現としては 0〜n/2 が正の周波数、n/2+1〜n-1 が負の周波数に対応する。条件分岐 `(i <= n / 2) ? i : i - n` でこの変換を実現している。次に、周波数インデックスから実際の Hz 単位の周波数値を計算している。計算式 `freq_idx * sample_rate / n` により、インデックスを実周波数に変換している。サンプリングレート（44100Hz）を使用して正確な周波数値を得ている。最後に、計算した周波数が指定範囲（low_cut〜high_cut）外であれば、

その周波数成分を 0 にして除去している。

```
% sox ./sound/doremi.wav -t raw -c 1 - | ./bandpass 8192 300 2000 | play -t raw -b 16 -c 1 -e s -r 4
```

このコマンドによりバンドパスフィルタに通した音を実際に聞いてみると、途中から音が聞こえなくなること
が確認できる。

```
# plot_spectrum.gp
# 変数の設定
sample_rate=44100
n=8192

# プロット設定
set xlabel 'Frequency (Hz)'
set ylabel 'Amplitude'
set logscale y
set xrange [0:5000]

# プロット実行
plot 'fft.dat' using (abs($1 <= n/2 ? $1*sample_rate/n : ($1-n)*sample_rate/n)):4 with lines title '
pause -1
```

上記のような gp ファイルを使用して、音の周波数を gnuplot で確認した結果、図 4 のようになり、しっかり
と指定した周波数間の音のみを通すフィルタになっていることが分かる。

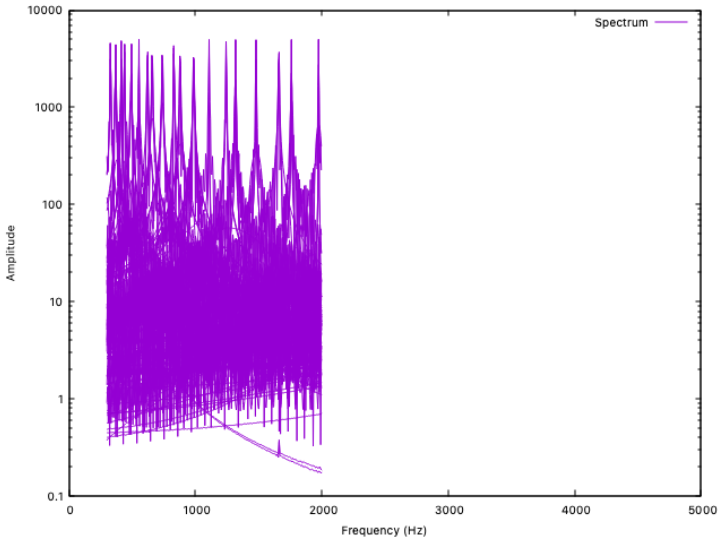

図 4: 本課題 4.5 の音の周波数を可視化したもの.