前期実験– I1 課題

高山乃綾

2025 年 7 月 18 日

1　はじめに

今回の I1 実験では, 最終的に I 実験で作成するインターネット電話に向けて C 言語の復習を行った. それに加えて, デバッグ方法などのプログラムを書く上での必須の作業の練習, およびファイル入出力や音の入出力, またデジタル信号処理の基礎を学んだ.

これが表示されたらうまく同期できている！これもうまく表示されるといいな

1 日目: 環境設定, C プログラミング
2 日目: ファイルの読み書きと, 音データの入出力
3 日目: 全時間実習
4 日目: デジタル信号処理

今回、コードについては全てを記載せず、課題に関して重要な部分のみを取り上げた.

2　実験結果と考察

2.1　1 日目

2.1.1　本課題 1.7

課題 HP（https://www.dropbox.com/scl/fi/4mzqn8k6f63vlu4ktojcn/problems.tar.gz?rlkey=c7dxh1wl1xzun7p99pcfub7ca&e=1&dl=0）には, 9 個の誤ったプログラムのコードが記載されている. それぞれに対し, 何が誤りなのかを順に調べた.

■2.1.1.1　p00.c のバグ修正

p00.c は, コンパイル時に ”p00.c” というファイルを 1 バイトずつ読み取り、標準出力として p00.c の全文がそのまま表示されるプログラムを想定している.

与えられたコード

Code 1: p00.c 修正前

```c
int main()
{
  FILE * fp = fopen("p00.c", "rb");
  while (1) {
    int c = fgetc(fp);
    if (c == EOF) break;
    fputc(c, stdout);
  }
  return 0;
}
```

問題となる行と理由

- 行 1：ヘッダ <stdio.h> が無いため FILE や fopen が宣言されずコンパイルエラーが起きてしまう.
- 行 3：fopen の戻り値 NULL チェックが無い → ファイルが開けないと segmentation fault が表示されてしまう.
- 行 10：fclose(fp); が無い → ファイルディスクリプタが解放されずリソースリークが起こってしまう.

修正版コード

Code 2: p00_fix.c 修正版

```c
#include <stdio.h>

int main(void)
{
    FILE *fp = fopen("p00.c", "rb");
    if (fp == NULL) {
        perror("p00.c");
        return 1;
    }

    while (1) {
        int c = fgetc(fp);
        if (c == EOF) break;
        fputc(c, stdout);
    }

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

修正ポイント

ヘッダ追加　`#include <stdio.h>` を 1 行目に追加した.

例外処理　行 6–9 に NULL 判定と `perror` を設けることで, fopen でエラーが起きた際の例外処理を行い, より丁寧なコードに変更した.

後始末　行 15 に `fclose(fp);` を追加し, データを全て確実に書き出すようにした. また, ”開いたら必ず閉じる” というコーディングの鉄則に, より従ったコードにした.

■2.1.1.2　p01.c のバグ修正

p01.c は、実行時に $i = 0, 1, 2, \ldots, 99$ について $\cos(i)$ と $\sin(i)$ を計算し、それぞれの二乗を加えた結果を各行で

$$\cos^2(i) + \sin^2(i) = 1.0$$

の形式で出力することを想定している.

与えられたコード

Code 3: p01.c 修正前

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

問題となる行と理由

- 行 1：<math.h> を #include していないため, cos, sin が宣言されずコンパイル時に警告やリンク時に未定義シンボルが出てしまう.
- 行 6：cos2, sin2 を定義前に呼び出しているため, エラーが出てしまう.「プロトタイプ宣言」が必要.

修正版コード (抜粋)

Code 4: p01_fix.c 修正版

```
1  #include <stdio.h>
2  #include <math.h> /* cos, sin の宣言 */
3
4  double cos2(double x);
5  double sin2(double x); /* プロトタイプ宣言をしておく */
6
7  int main(void)
8  {
9      int i;
10     for (i = 0; i < 100; i++) {
11     double y = cos2(i) + sin2(i);
12     printf("cos^2(%d)+sin^2(%d) = %f\n", i, i, y);
13     }
14     return 0;
15 }
16
17 /* 以下、cos2() と sin2() の実装は修正前と同一 */
```

修正ポイント

数学関数の宣言　1 行目直後に #include <math.h> を追加し, cos と sin を使えるようにした.

プロトタイプ宣言の追加　cos2, sin2 のプロトタイプを main 関数より前に置いて, ”implicit declaration” を起こさないようにした.

■2.1.1.3　p02.c のバグ修正

p02.c は数値 x と y を，プログラムを動かすときにいっしょに渡すことで，その積（$x \times y$）を計算し，その結果を画面に表示することを目的としたプログラムである．このように，プログラムを起動するときに渡す数値などの情報のことを「**コマンドライン引数**」と呼ぶ.

与えられたコード

Code 5: p02.c 修正前

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

問題となる行と理由

- 行 1-2：`<stdio.h>` と `<stdlib.h>` を `#include` していないため, `printf`, `atof` が宣言されずコンパイル時に警告やリンク時に未定義シンボルが出てしまう.
- 行 3：`int main(int argc, char **argv)` にもかかわらず `argc` の値を確認していない. もし引数が 2 つ未満で実行してしまうと, `argv[1]` が `NULL` になり, `atof(NULL)` で Segmentation fault などのエラーが出てしまう.

修正版コード (抜粋)

Code 6: p02_fix.c 修正版

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

int main(int argc, char **argv)
{
    /* 引数の個数のチェック */
    if (argc != 3) {
        fprintf(stderr, "Usege: %s <x> <y>/n". argv[0]);
        return 1;
    }
/* 以下、修正前と同様 */

}
```

修正ポイント

ヘッダ追加　先頭に `#include <stdio.h>` と `#include <stdlib.h>` を追加し, `printf` と `atof` の未定義を解決した.

引数の個数チェック　`if (argc != 3)` を追加し, 引数不足の場合は`"Usege: %s <x> <y>/n". argv[0]` と使い方を表示させるようにした.

■2.1.1.4　p03.c のバグ修正

p03.c は，コマンドライン引数として与えられた 2 つの 3 次元ベクトル $\vec{A} = (Ax, Ay, Az)$ と $\vec{B} = (Bx, By, Bz)$ のなす角 θ をラジアンで計算し，標準出力に表示するプログラムである．角度の計算には，次の関係式が用いられている：

$$\cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \cdot |\vec{B}|} \quad \Rightarrow \quad \theta = \cos^{-1}\left(\frac{\vec{A} \cdot \vec{B}}{|\vec{A}| \cdot |\vec{B}|}\right)$$

ここで，$\vec{A} \cdot \vec{B}$ は内積，$|\vec{A}|$ および $|\vec{B}|$ はベクトルの大きさ（ノルム）を表す．

与えられたコード

Code 7: p03.c 修正前

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


typedef struct vect3
{
  double x;
  double y;
  double z;
} vect3;


/* dot product (naiseki) */
double dot(vect3 * A, vect3 * B)
{
  return A->x * B->x + A->y * B->y + A->z * B->z;
}


double angle(vect3 * A, vect3 * B)
{
  return acos(dot(A, B) / sqrt(dot(A, A) * dot(B, B)));
}


vect3 * mk_point(double x, double y, double z)
{
  vect3 * p;
  p->x = x;
  p->y = y;
  p->z = z;
  return p;
```

```
34 }
35
36
37 int main(int argc, char ** argv)
38 {
39   vect3 * A = mk_point(atof(argv[1]), atof(argv[2]), atof(argv[3]));
40   vect3 * B = mk_point(atof(argv[4]), atof(argv[5]), atof(argv[6]));
41   double a = angle(A, B);
42   printf("%f\n", a);
43   return 0;
44 }
```

問題となる行と理由

- 行 29：ポインタ p に対して malloc によるメモリ確保がされておらず, 未定義のメモリを参照して代入しているため, 実行時エラー（Segmentation Fault）につながる.
- 行 30 – 32：メモリ確保をしていないため, p->x などの代入は未定義動作となる.

修正版コード（抜粋）

Code 8: p03_fix.c 修正版

```
1  #include <math.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  /* vect3 の定義，dot(), angle() の実装は修正前と同様 */
6
7  vect3 *mk_point(double x, double y, double z)
8  {
9      vect3 *p = malloc(sizeof(vect3));        /* ← 動的メモリ確保を追加 */
10     if (!p) { perror("malloc"); exit(1); }   /* ← malloc 失敗時のエラー処理を追加 */
11     p->x = x;
12     p->y = y;
13     p->z = z;
14     return p;
15 }
16
17 int main(int argc, char **argv)
18 {
19     vect3 *A = mk_point(atof(argv[1]), atof(argv[2]), atof(argv[3]));
20     vect3 *B = mk_point(atof(argv[4]), atof(argv[5]), atof(argv[6]));
21     double a = angle(A, B);
22     printf("%f\n", a);
23     free(A);  /* ← 動的確保したメモリを解放 */
24     free(B);  /* ← 同上 */
25     return 0;
```

```
26 }
```

修正ポイント

メモリの明示的な確保　`vect3 *p;` のみでは, ポインタが指す先の領域が存在しないため, `malloc(sizeof(vect3))` を用いて必要な構造体の分だけメモリを確保するように修正した.

メモリ確保の成否の確認　`malloc` によるメモリ確保が失敗する可能性があるため, `if (!p)` による確認を加え, 失敗時にはエラーメッセージを表示してプログラムを終了するようにした.

使用後のメモリの解放　動的に確保したメモリは最後に `free` によって解放しないとメモリリークにつながるため, `main` 関数の最後で `free(A); free(B);` を追加した.

■2.1.1.5　p04.c のバグ修正

p04.c は, p03.c と同様の実装を想定されている.

与えられたコード

Code 9: p04.c 修正前

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

/* vect3 の定義，dot(), angle() の実装は p03.の修正前と同様 c */

vect3 * mk_point(double x, double y, double z)
{
  vect3 p[1];
  p->x = x;
  p->y = y;
  p->z = z;
  return p;
}

/* 関数の定義も mainp03.の修正前と同様 c */
```

問題となる行と理由

p04.c は, `mk_point` 関数内で `vectt3 p[1];` を宣言し, そのアドレスを最後に返しているが, この `p` は関数内のローカル変数であるため, 関数終了後とともに破棄される. そのため, そのアドレスを返すことは未定義動作につながるため, エラーが発生する.

そのため, p04.c においても修正方法は p03.c と同様であり, 修正版コードは p03.c のものと同じである.

■2.1.1.6　p05.c のバグ修正

p05.c は, プログラム実行時に同じディレクトリ内のファイル 'p05.c' を「バイナリモード」で開き, その内容を最大 100 バイトずつ読み込んで標準出力に書き出すことで, ファイルの中身を丸ごと画面にコピーすることを目的としたプログラムである. 具体的には, 'fread()' で読み込んだデータをそのまま 'fwrite()' に渡すことで, テキスト/バイナリを問わずファイルの完全なコピーを行える仕様を想定している.

与えられたコード

Code 10: p05.c 修正前

```
1  #include <assert.h>
2  #include <stdio.h>
3  int main()
4  {
5    FILE * fp = fopen("p05.c", "rb");
6    char buf[100];
7    while (1) {
8      int n = fread(buf, 1, 100, fp);
9      if (n == 0) break;
10     fwrite(buf, 1, n, stdout);
11   }
12   return 0;
13 }
```

問題となる行と理由

- 行 5 : 指定したファイル名が `p05.c` となっており, 数字の '0' と英文字の 'O' を入れ違えている.
- 行 6 : `fopen()` の戻り値をチェックしていないため, ファイルが開けなかった場合に以降の `fread()` が未定義動作を起こす.

修正版コード

Code 11: p05_fix.c 修正版

```
1  #include <assert.h>
2  #include <stdio.h>
3  int main()
4  {
5    FILE * fp = fopen("p05.c", "rb"); // ファイル名の訂正
6    assert(fp != NULL); // ファイルオープンならここで止まるようにする
7    char buf[100];
8    while (1) {
9      int n = fread(buf, 1, 100, fp);
10     if (n == 0) break;
```

```
11      fwrite(buf, 1, n, stdout);
12    }
13    return 0;
14  }
```

修正ポイント

ファイル名の訂正　英文字や数字は混同しやすいため, 正しいファイル名を選択できているか十分に確認する.

ファイルオープン時のチェック　`assert(fp != NULL);` を追加し, ファイルが開けなかった場合に即座にエラーを報告してプログラムを停止するようにした.

ファイルのクローズ　読み取り終了後に `fclose(fp);` を呼び出し，開いたファイルを適切に閉じるようにした.

■2.1.1.7　p06.c のバグ修正

p06.c は，プログラム実行時に同じディレクトリ内のファイル ‘p06.c‘ をシステムコール ‘open()‘ で読み取り専用に開き，‘read()‘ / ‘write()‘ を用いて最大 100 バイトずつ読み込み・書き出すことで，ファイルの中身を丸ごと標準出力にコピーすることを目的としたプログラムである.

与えられたコード

Code 12: p06.c 修正前

```
1  int main()
2  {
3    int fd = open("pO6.c", O_RDONLY);
4    char buf[100];
5    while (1) {
6      int n = read(fd, buf, 100);
7      if (n == 0) break;
8      write(1, buf, n);
9    }
10   return 0;
11 }
```

問題となる行と理由

- 行 3：‘open()‘, ‘read()‘, ‘write()‘ を宣言するヘッダ `fcntl.h` と `unistd.h` が宣言されておらず，コンパイル時に未定義シンボルのエラーが発生する.
- 行 3：開こうとしているファイル名が ‘”pO6.c”‘ となっており，数字の ‘0’ と英文字の ‘O’ を入れ違えている.

修正版コード

Code 13: p06_fix.c 修正版

```c
#include <fcntl.h>
#include <unistd.h>   // open, read, write の宣言を使うために追加
#include <stdio.h>    //  （任意）エラーメッセージ出力用

int main()
{
  int fd = open("p06.c", O_RDONLY);   // ファイル名の訂正
  char buf[100];
  while (1) {
    int n = read(fd, buf, 100);
    if (n == 0) break;
    write(1, buf, n);
  }
  return 0;
}
```

修正ポイント

必要ヘッダの追加　‘open()‘, ‘read()‘, ‘write()‘ を正しく使うため，`fcntl.h` と `unistd.h` のインクルードを追加した．

ファイル名の訂正　‘”pO6.c”‘ を誤っていたため，正しい ‘”p06.c”‘ に修正した．

標準入出力ヘッダの追加　`stdio.h` を追加し，必要に応じて ‘perror()‘ などを使いやすくした．

■2.1.1.8　p07.c のバグ修正

p07.c は，コマンドライン引数で渡されたファイルをバイナリモードで開き，動的に拡張可能なバッファにその全内容を読み込んだうえで，最後にバッファ内を逆順に出力することを目的としたプログラムである．読み込みは最初 10 バイト分だけ確保したバッファに対し，あふれたら 10 バイトずつ拡張しながら続ける設計である．

与えられたコード

Code 14: p07.c 修正前

```c
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <assert.h>

int main(int argc, char ** argv)
{
  /* try to read the whole contents of the file into s */
```

```
 9   FILE * fp = fopen(argv[1], "rb");
10   /* s is initially small (10 bytes) */
11   int begin = 0;
12   int end = 10;
13   char * s = (char *)malloc(end);
14   while (1) {
15     int r = fread(s + begin, 1, end - begin, fp);
16     if (r < end - begin) {
17       /* reached end of file */
18       end = begin + r;
19       break;
20     }
21     /* buffer full, extend by 10 bytes */
22     begin = end;
23     end += 10;
24     char * t = (char *)malloc(end); /* new buffer */
25     assert(t);
26     bcopy(s, t, end);               /* ← コピー長が誤り */
27     free(s);
28     /* ← s = t; が抜けている */
29   }
30   /* print s from end to beginning */
31   int i;
32   for (i = end - 1; i >= 0; i--) {
33     putchar(s[i]);
34   }
35   return 0;
36 }
```

問題となる行と理由

- 行 17：`bcopy(s, t, end);` でバッファ全体（end バイト）をコピーしているが，実際に中身が入っているのは前回読み込んだ `begin` バイト分だけである．
- 行 18：`free(s);` のあとに `s = t;` が抜けており，拡張後のバッファを使わないまま古いポインタを参照し続けている．
- 最終行付近：動的に確保したバッファ `s` およびファイル `fp` を閉じる／解放する処理がないため，リソースリークが発生する可能性がある．

■2.1.1.9　p07.c 修正差分

以下では，修正前と異なる部分のみを示す．

バッファ拡張時の処理

Code 15: p07_fix_diff.c

```
// --- 修正前 ---
    bcopy(s, t, end);
    free(s);
// --- 修正後 ---
    bcopy(s, t, begin);    // 実際に読み込まれたバイト数だけをコピー
    free(s);
    s = t;                  // 新しく確保したバッファを使うようにする
```

リソース解放の追加

Code 16: p07_fix_diff2.c

```
// --- 修正前末尾 ---
  for (i = end - 1; i >= 0; i--) {
    putchar(s[i]);
  }
  return 0;
// --- 修正後末尾 ---
  for (i = end - 1; i >= 0; i--) {
    putchar(s[i]);
  }
  free(s);     // 動的バッファを解放
  fclose(fp); // ファイルを閉じる
  return 0;
```

修正ポイント

コピーサイズの修正　`bcopy(s, t, end)` を `bcopy(s, t, begin)` に変更し，実際に読み込まれたバイト数だけを複製するようにした．

バッファ更新の追加　`free(s)` のあとに `s = t` を入れ，拡張後のメモリ領域を正しく参照するようにした．

リソースの解放　ループ後に `free(s)` と `fclose(fp)` を追加し，動的バッファとファイル記述子を適切に解放するようにした．

■2.1.1.10　p08.c のバグ修正

p08.c は，標準入力から数式文字列を読み込み，再帰下降法により四則演算と括弧を評価して結果を出力するプログラムである．入力例としては `23 + 3 * 67 - (3 + 4)` のような式を想定する．

与えられたコード

Code 17: p08.c 修正前

```
…
  （省略） …

double H_expression()
{
  switch (*p) {
  case '0' ... '9': {
    return number();
  }
  case '(': {
    double x = E_expression();
    if (*p == ')') {
      return x;
    } else {
      syntax_error();
    }
  }
  default:
    syntax_error();
  }
}

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

問題となる行と理由

- H_expression の定義：引数を取らない関数は `double H_expression(void)` とすべきところ，`void` が抜けている．
- 行内の括弧処理：(を読み飛ばしていないため，開き括弧自身が再評価されて無限ループや不正な結果

を招く．

- 閉じ括弧の判定：条件が逆で，) が来たときにエラー扱いしてしまっている．
- 関数の最後：すべての分岐で return を保証しておらず，コンパイラの警告や未定義動作を誘発する．
- E_expression の加減算：x =+ F_expression() や x =- F_expression() は「符号付き代入」ではなく，「x を正しく更新しない」間違い表記である．

Code 18: p08_fix.c 修正差分

```
// --- 修正前 ---
double E_expression() {
  double x = F_expression();
  while (1) {
    if (*p == '+') {
      p++;
      x =+ F_expression();
    } else if (*p == '-') {
      p++;
      x =- F_expression();
    } else {
      return x;
    }
  }
}
// --- 修正後 ---
double E_expression(void) {
  double x = F_expression();
  while (1) {
    if (*p == '+') {
      p++;
      x = x + F_expression();
    } else if (*p == '-') {
      p++;
      x = x - F_expression();
    } else {
      return x;
    }
  }
}
```

修正ポイント

関数宣言の修正　引数を取らない関数は (void) を明示し，コンパイラ警告を防止した．

括弧の読み飛ばし　開き括弧および閉じ括弧を p++ で適切にスキップし，入出力ポインタ位置を正しく制御するようにした．

戻り値の保証　すべてのコードパスで return を記述し，未定義動作を防いだ．

加減算表現の修正　‘=+‘／‘=-‘ を用いた誤表記を，‘x = x + ...‘／‘x = x - ...‘ に修正した．

2.2 2 日目

2.2.1 本課題 2.8: バイナリファイルの読み込みと表示

本課題では, コマンドラインで指定されたファイルを open で開き, 何バイト目が何というバイトだったかを, 一行目にバイト数を, 二行目にその値を表示させるプログラム read_data.c を作成した. そして, その前に作成した my_data を読み込ませた. my_data はファイルの第 0 バイト目に 0, 第 1 バイト目に 1,..., 第 255 バイト目に 255 が入っている 256 バイトのファイルである.

read_data.c のコード

Code 19: read_data.c

```c
/* ファイルの読み込み練習 */

#include <fcntl.h>   // open(), O_CREAT など
#include <unistd.h>  // write(), close() など
#include <stdio.h>   // perror(), fprintf() など
#include <stdlib.h>  // exit(), EXIT_SUCCESS/EXIT_FAILURE など

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return EXIT_FAILURE;
    } /* コマンドライン引数が少ない場合に、使い方を示して終了してくれる部分*/

    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        perror("error");
        exit(EXIT_FAILURE);
    }

    unsigned char buffer[64]; // 一度に 64 バイトまで読み込む
    ssize_t n; // 実際に読み込んだバイト数. signed size_t ( 符号あり size_t )の略.
    int byte_index = 0; // ファイル内でのバイト位置を記録

    while (1) {
        n = read(fd, buffer, sizeof(buffer)); // 最大 64 バイト読み込む
        if (n == -1) {
            perror("read");
            close(fd);
            exit(EXIT_FAILURE); // ssize_t のおかげで、うまくいけば読み取ったバイト数を、間違っていたら -1 を返してくれる.
        }

        if (n == 0) break; // EOF ( ファイルの終わりという意味 )
```

```
33
34          for (ssize_t i = 0; i < n; i++) {
35              printf("%d %d\n", byte_index, buffer[i]);
36              byte_index++;
37          }
38      }
39
40      if (close(fd) == -1) {
41          perror("close");
42          exit(EXIT_FAILURE);
43      }
44
45      return EXIT_SUCCESS;
46  }
```

実行例

Code 20: 実行例

```
1  $ gcc -std=c11 -Wall -Wextra -o read_data read_data.c
2  $ ./read_data my_data
3  0 0
4  1 1
5  2 2
6  3 3
7  4 4
8  ...
9  252 252
10 253 253
11 254 254
12 255 255
```

read_data.c の挙動説明

- `open(argv[1], O_RDONLY)`：コマンドラインで指定されたファイルを読み取り専用で開き，失敗時は `perror` によってエラー報告して終了する．
- 読み込みバッファ `unsigned char buffer[64]`：一度に最大 64 バイトずつ `read()` で読み込む．戻り値 `n` は実際に読み込んだバイト数（EOF で 0，エラーで -1）を示す．
- ループ内部：`for (ssize_t i = 0; i < n; i++)` で 0 起点の `byte_index` とバッファ内の値 `buffer[i]` を `printf("%d %d\n", byte_index, buffer[i])` で出力し，バイト位置をインクリメントする．
- `close(fd)`：全読み込み後にファイル記述子を閉じ，リソースを解放する．失敗時は `perror` を呼んで終了する．

このように, `my_data` の内容が正しく表示されることを確認できた．また, この `read_data.c` は `my_data` に限らず, 任意のサイズのファイルを扱うことができることも確認した．

2.2.2 本課題 2.13: 生音声データの取得と可視化

本課題では, マイク入力を使って生音声データを取得し, 生のバイト列 (RAW 形式) として保存した後, 課題 (2.8) で作成した `read_data` プログラムを流用してバイト列を読み出し, gnuplot による波形表示を行った.

以下の条件で録音を行った.

- 形式：RAW（拡張子 `.raw`）
- 量子化ビット数：8bit
- チャンネル数：1（モノラル）
- 符号化：unsigned-integer
- サンプリング周波数：44100Hz

■2.2.2.1 (1) 録音とファイル準備

まず，sox の `rec` コマンドでマイク入力を以下の条件で録音し，RAW ファイル `sound2.13.raw` を作成した.

Code 21: 生音声の録音（RAW 形式）

```
$ rec -q -c 1 -b 8 -e unsigned-integer -r 44100 sound2.13.raw
#  （ 録音終了後は Ctrl + C ）
```

■2.2.2.2 (2) プログラムのコンパイル

以下のように, `read_data.c` を `read_data` としてコンパイルした.

Code 22: read_data のコンパイル

```
$ gcc -std=c11 -Wall -Wextra -o read_data read_data.c
```

■2.2.2.3 (3) バイト列の読み出し

`sound2.13.raw` を waveform2.13.txt として変換し読み出した.

Code 23: バイト列の読み出し実行例

```
$ ./read_data sound2.13.raw > waveform2.13.txt
```

■2.2.2.4 (4) プロットと画像出力

gnuplot を用いて, 波形の結果を画像表示した.

Code 24: GNUplot 課題 2.13

```
$ gnuplot
$ gnuplot> plot "waveform2.13.txt" with linespoints
```

波形の結果は，以下のようになった．

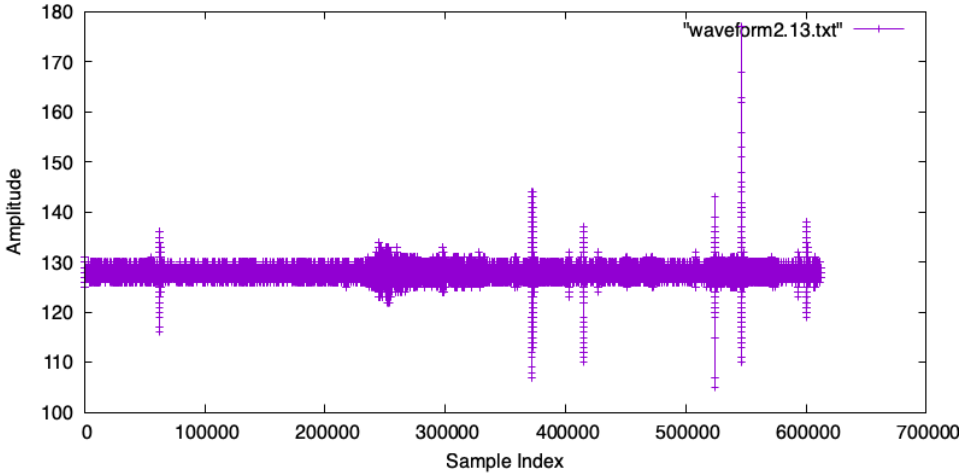

図 1: 生音声データの波形（8bit unsigned, 44100Hz）

■2.2.2.5　結果

波形（図 1）を見ると，振幅値が概ね 120〜135 の範囲で変動していることがわかる．音声を発声していない無音部分ではほぼ定常的に約 128 あたりに値が集中し，声を出した瞬間に上方向へピークが現れ，その後減衰して再び定常値へ戻っている様子が読み取れた．

■2.2.2.6　考察

- 8bit の量子化では，無音時の中心値（約 128）から ± 127 の振幅を扱うため，無音部分は量子化誤差により値が微細に揺らいでいる（量子化ノイズ）．
- 発声時のピークは声の強弱や周波数成分に依存し，図中で複数の山谷が見られるのは発音の抑揚や子音／母音の切替を反映していると考えられる．
- サンプリング周波数 44100Hz は音声帯域を十分カバーしているため，細かな振動もきれいに捉えられているが，8bit 量子化ではダイナミックレンジが狭いため，量子化ビット数を増やすとより滑らかな波形再現が期待できる．

2.2.3　本課題 2.14: 正弦波 sin.c の可視化

本課題では, 振幅 A，周波数 f，サンプル数 n を指定して

$$x(t) = A\sin(2\pi f t)$$

を 16bit・モノラル・44100Hz の RAW 形式で n 点出力するプログラム `sin.c` を作成した．

■2.2.3.1　sin.c のコード

Code 25: sin.c 本文

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>  // int16_t を使うため

#define SAMPLING_RATE 44100

int main(int argc, char *argv[]) {
    if (argc != 4) {
        fprintf(stderr, "Usage: %s <amplitude> <frequency> <samples>\n", argv[0]);
        return EXIT_FAILURE;
    }
    double A = atof(argv[1]);    // 振幅
    double f = atof(argv[2]);    // 周波数 (Hz)
    int n    = atoi(argv[3]);    // サンプル数

    for (int i = 0; i < n; i++) {
        double t = (double)i / SAMPLING_RATE;
        double value = A * sin(2 * M_PI * f * t);
        int16_t sample = (int16_t)value;
        fwrite(&sample, sizeof(sample), 1, stdout);
    }
    return EXIT_SUCCESS;
}
```

■2.2.3.2　補足：パイプによる直接再生

以下のようにパイプ（|）を用いて，sin の出力をそのまま play コマンドに渡すこともできる：

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

この方法では，一時的な中間ファイル（sin.raw など）を作成せずに済むため，プログラムと音声再生の接続がスムーズであり，繰り返しの確認や実験に便利である．最後の - は，play に対して「標準入力から読み込む」ことを意味する．

■2.2.3.3　コンパイルと実行

Code 26: sin.c のコンパイル

```
$ gcc -std=c11 -Wall -lm -o sin sin.c
\begin{lstlisting}[language=bash,caption={ 正弦波の生成と再生例 },label=lst:run_sin]
$ ./sin 10000 440 88200 > sin.raw
$ play -t raw -b 16 -c 1 -e s -r 44100 sin.raw
# または
```

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

これを実行することで, 440Hz の単音が 2 秒間流れ, 条件に合った正弦波が得られたと確認できた.

2.2.4 本課題 2.15: 16bit 整数単位でのデータ表示と正弦波の可視化

本課題では，課題 2.8 で作成したバイト列表示プログラム `read_data.c` を改良し，ファイルの内容を 16bit 単位（2 バイト）で読み取り，各サンプルを `int16_t` 型（符号付き短整数）として表示するプログラム `read_data2.c` を作成した．これにより，`sin.c` で生成した 16bit 正弦波データを正しく解釈し，可視化できるようになった.

■2.2.4.1 主な変更点

- `read_data.c` では **1 バイトずつ（uint8_t）** 読み込んで表示していた.
- 本課題の `read_data2.c` では，**2 バイトずつ読み込んで**，`int16_t` 型として解釈して出力する．これにより，16bit signed PCM 音声ファイル（`sin.raw` など）を正しく数値として扱うことができる.
- 読み込みバッファを `int16_t buffer[32];` とし，64 バイト = 32 サンプル単位で読み取って処理する.

■2.2.4.2 コード抜粋 (変更点)

Code 27: read_data2.c（抜粋）

```
int16_t buffer[32]; // 32 個の 16 bit 符号付き整数をまとめて読み込む
ssize_t n = read(fd, buffer, sizeof(buffer)); // 最大 64 バイト読み込み
ssize_t samples = n / 2; // 1 サンプル 2 バイト
for (ssize_t i = 0; i < samples; i++) {
    printf("%d %d\n", sample_index, buffer[i]);
    sample_index++;
}
```

■2.2.4.3 実行例

```
$ gcc -std=c11 -Wall -o read_data2 read_data2.c
$ ./read_data2 sin.raw > waveform2.15.txt
```

■2.2.4.4 gnuplot による波形の可視化

出力されたファイル `waveform2.15.txt` を `gnuplot` で読み込み, 以下のようなスクリプトで波形を可視化した.

Code 28: waveform2.15.gp の内容

```
set terminal pngcairo size 800,400
set output 'waveform2.15.png'
set title '16bit Sine Wave from sin.raw'
set xlabel 'Sample Index'
set ylabel 'Amplitude (Signed 16bit)'
set xrange [0:500]          # 拡大表示用
set yrange [-11000:11000]   # 振幅範囲に合わせる
plot 'waveform2.15.txt' using 1:2 with linespoints title 'sample'
```

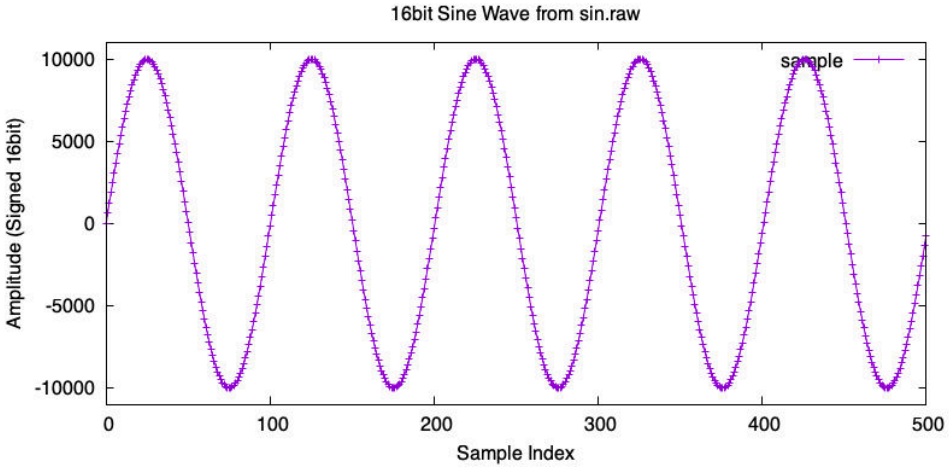

図 2: 16bit 正弦波の波形（最初の 500 サンプルの拡大表示）

■2.2.4.5　結果

図 2 に示すように，440Hz の正弦波が時間軸上で滑らかに振動しており，サンプリング周波数 44100Hz により 1 周期あたり約 100 点のサンプルが得られていることが確認できた．また，振幅は指定した値（± 10000）付近で変動しており，16bit の signed 整数として正しく表現されていることが波形からも視覚的に読み取れた．サンプル点が明示されていることで，各時刻における数値の離散性も確認できた．なお，表示範囲を限定し，描画方法を `linespoints` に変更したことで，波形の構造が視認しやすくなり，各サンプル点が正弦波に従って規則的に分布している様子が確認できた．

2.2.5　選択課題 2.16: ドレミファソラシドの正弦波生成と可視化

本課題では，振幅 A と出力したい音の数 n を指定して，ドレミファソラシドの音階を順に生成する C プログラム `doremi.c` を作成した．各音は長さ約 0.3 秒（13220 標本）とし，16bit signed PCM 形式（モノラル，44100Hz）で出力されるように設計している．

■2.2.5.1　プログラム本文

Code 29: doremi.c 全文

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

#define SR 44100        // サンプリングレート (Hz)
#define LEN 13220       // 一音あたりのサンプル数 約秒 (0.3)
#define SCALE_COUNT 8   // ドレミファソラシドの 8 音

int main(int argc, char *argv[]) {
    if (argc < 3) {
        fprintf(stderr, "Usage: %s <amplitude> <n_notes>\n", argv[0]);
        return EXIT_FAILURE;
    }

    double A = atof(argv[1]); // 振幅
    int n = atoi(argv[2]);    // 出力する音の数

    // ～（ド～ド）の周波数 C4C5 (Hz)
    double freqs[SCALE_COUNT] = {
        261.63, // ド (C4)
        293.66, // レ (D4)
        329.63, // ミ (E4)
        349.23, // ファ (F4)
        392.00, // ソ (G4)
        440.00, // ラ (A4)
        493.88, // シ (B4)
        523.25  // ド (C5)
    };

    for (int i = 0; i < n; i++) {
        double f = freqs[i % SCALE_COUNT]; // 音階を循環
        for (int j = 0; j < LEN; j++) {
            double t = (double)j / SR;
            double v = A * sin(2.0 * M_PI * f * t);
            int16_t sample = (int16_t)v;
            fwrite(&sample, sizeof(sample), 1, stdout);
        }
    }

    return EXIT_SUCCESS;
}
```

■2.2.5.2　コンパイルと実行例

```
$ gcc -std=c11 -Wall -lm -o doremi doremi.c
$ ./doremi 10000 16 > doremi.raw
$ play -t raw -b 16 -c 1 -e s -r 44100 doremi.raw
```

■2.2.5.3 **波形の可視化** read_data2.c を使って数値データに変換し，gnuplot により波形を表示した．今回，音が高くなるにつれて波長が短くなっていく様子がより視覚的に分かりやすくなるように，ドミソの三つを飛び飛びにプロットした．

```
$ ./read_data2 doremi.raw > waveform2.16.txt
set terminal pngcairo size 800,900       # 出力フォーマットと画像サイズ
set output 'doremi_DoMiSo.png'           # 保存先ファイル名

unset key
set multiplot layout 3,1 rowsfirst       # 3 行 1 列（縦に並べる）
set xlabel 'Sample Index'
set ylabel 'Amplitude (16bit)'
set yrange [-11000:11000]                # 振幅スケールを固定

# ---- Do (C4) ----
set title 'Do (C4 261.6 Hz)'
set xrange [0:13220]                     # 0.3 秒 (=13220 サンプル)
plot 'waveform2.16.txt' using 1:2 with lines

# ---- Mi (E4) ----
set title 'Mi (E4 329.6 Hz)'
set xrange [2*13220:3*13220]             # Do から 2 音先
plot 'waveform2.16.txt' using 1:2 with lines

# ---- So (G4) ----
set title 'So (G4 392.0 Hz)'
set xrange [4*13220:5*13220]             # Do から 4 音先
plot 'waveform2.16.txt' using 1:2 with lines

unset multiplot
set output                               # ファイルを閉じる
set terminal qt                          # 必要なら画面表示用に戻す
```

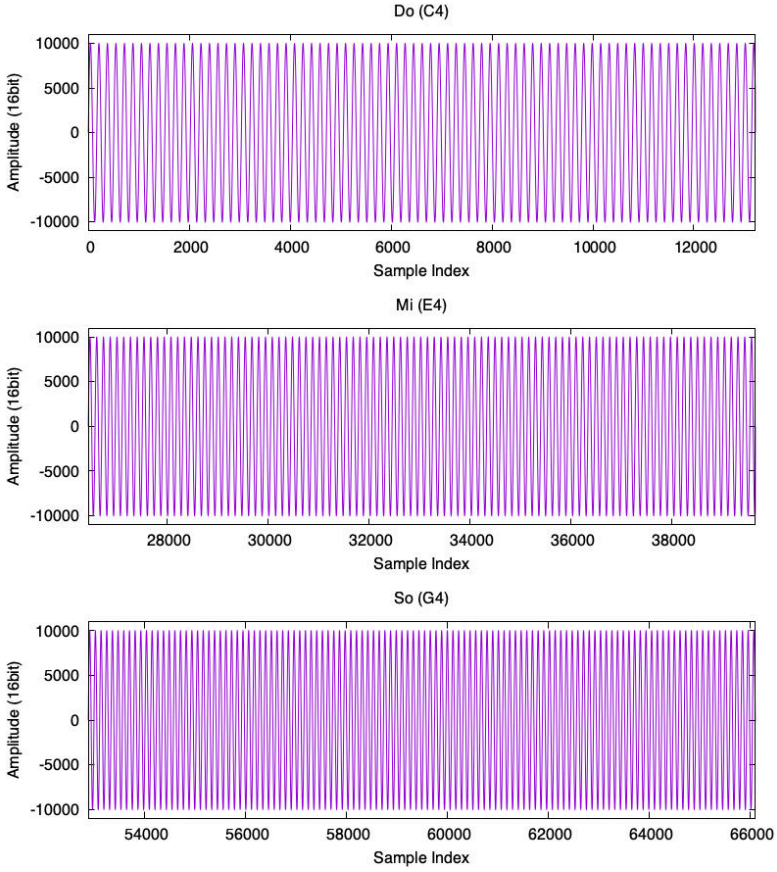

図 3: ド（C4）・ミ（E4）・ソ（G4）3 音の波形（各 0.3 秒）

■2.2.5.4　**結果**　図 3 では，ド→ミ→ソへ進むにつれて波長が短くなり, 周波数が高くなっていることが明確に確認できた. いずれの波形も振幅はおよそ ±10 000 に収まり，16bit signed PCM として正しく生成されている. このように 1 音ずつ切り出して比較することで，音階による周期の変化を視覚的に把握しやすくなった.

2.3　3 日目・4 日目

2.3.1　本課題 4.1: 標本化周波数と音質の関係

本課題では, 標本化周波数 (単位時間あたりいくつの点をとるか) がどのくらいまでであれば, 元の音を完全に復元することができるのか, という問題を考えた. 具体的には, 16 bit のモノラルの raw データを標準入力から受け取って, それを与えられた割合で間引いて (downsampling) 標準出力に出す C プログラム `downsample.c` を作成した.

downsample.c のコード

Code 30: downsample.c

```c
/* 16 bit のモノラルの raw データを標準入力から受け取り、それを与えられた割合で間引いて標準出力に出すプログラム
*/

/* Usage: ./downsample <rario> < orig.raw > new.raw */

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <ratio>\n", argv[0]);
        return EXIT_FAILURE;
    }

    int ratio = atoi(argv[1]); // ratio の文字列を整数変換
    if (ratio <= 0) {
        fprintf(stderr, "周波数は正の値でなければいけません.\n");
        return EXIT_FAILURE;
    }
    /* 16 bit サンプルを一つずつ読み込み、ratio ごとにひとつだけ出力するループ */
    int16_t sample;
    int count = 0;
    while (fread(&sample, sizeof(sample), 1, stdin) == 1) {
    // sizeof(sample) = 2 バイト分を 1 個読み込む。
    //    成功すると 1 が返ってくるので、1 ならループ継続。EOF で -1 が返ってくるまでずっとループを繰り返す。

        if (count % ratio == 0) {
        fwrite(&sample, sizeof(sample), 1, stdout);
        // もし「 count を ratio で割ったあまりが 0 」(=ちょうど ratio ごとの位置)
        //    なら、このサンプルを標準出力に書き出す。
        }

    count++;
    // 読み込んだサンプル数をインクリメント。次のループで判定に使う。
    }

    return EXIT_SUCCESS;


}
```

次にこの downsample.c を用いて, 実験ホームページ上にあるサンプル音源である masami-nagasawa.wav (女性がインタビューを受けている様子を録音した音声) に対し, その音源をさまざまな割合で間引いていった.

1. masami-nagasawa.wav を raw 形式（16bit, モノラル, signed integer, 44100 Hz）に変換する.

Code 31: raw 変換コマンド

```
$ sox masami-nagasawa.wav -t raw -b 16 -c 1 -e s -r 44100 orig.raw
```

2. 間引き比率 r を 2, 3, 5, 10, 15, 20, 25, 30 として，各々について downsample を実行し，間引き後のファイルを生成する.

Code 32: downsample 実行コマンド

```
$ ./downsample r < orig.raw > ds_r.raw
```

3. 生成した raw ファイルを，再生周波数を $44,100/r$ Hz に設定して再生する.

Code 33: 再生コマンド

```
$ play -t raw -b 16 -c 1 -e s -r $((44100/r)) ds_r.raw
```

■2.3.1.1 考察

間引き比率 r を徐々に大きく（＝標本化周波数を低く）すると，音声は高域から順に失われ，雑音まじりの音色へ変化した. 主観的に 「違和感なく会話を理解できる」 限界は $r = 3$（標本化周波数 ≈ 14.7 kHz）であり，$r \geq 20$（≤ 1.76 kHz）以降ではもはや語句を聞き取るのも困難であった.

実験音源は女性インタビュー音声で，スペクトルの主要成分は 0 – 7 kHz に分布する. Nyquist–Shannon の復元条件 $f_s \geq 2f_{\max}$ を満たすのが $f_s \approx 16$ kHz（本実験の $r = 3$ 付近）であり，体感上の“明瞭さ”の境界とも一致した. 実際，ITU-T G.722 などワイドバンド電話規格も 16 kHz サンプリングを採用している. つまり「通話品質」を確保するには $f_s \approx 16$ kHz 程度が実用的な下限であり，それより低いレートではフォルマント折り返しやエイリアシングにより急激な劣化が起こるという理論的結論を, 本実験によって定性的に裏付けられたと言える.

2.3.2 本課題 4.3: 3528 Hz 正弦波と 1/10 間引き標本の可視化

周波数 $f = 3528$ Hz（$= 441$ Hz $\times 8$）の正弦波を標本化周波数 $f_s = 44100$ Hz で生成し，gnuplot を用いて

- 元の標本列（44.1 kHz）
- 10 点ごとに間引いた標本列（4.41 kHz 相当）

を重ね描きして可視化した. 実行手順と結果を以下に示す.

■2.3.2.1 (1) 16 bit RAW 波形の生成

Code 34: 3528 Hz 正弦波の生成

```
$ gcc -o sin sin.c -lm
$ ./sin 10000 3528 44100 > sin_4-3.raw   # 1 秒分（44100 点）
```

二行目は「振幅が 10000 で $f = 3528\,\mathrm{Hz}$（$= 441\,\mathrm{Hz} \times 8$）の正弦波を標本化周波数 $f_s = 44100\,\mathrm{Hz}$ で生成し, それを sin_4-3.raw に保存する」という意味である.

■2.3.2.2 (2) gnuplot 用テキストデータへの変換

Code 35: RAW → テキスト変換

```
$ gcc -o read_data2 read_data2.c
$ ./read_data2 sin_4-3.raw > sin_4-3.dat
```

read_data2 は<サンプル番号> <振幅値>の 2 列テキストを出力するため, 変換後の sin_4-3.dat は gnuplot にそのまま渡せる.

■2.3.2.3 (3) gnuplot による重ね描きと PNG 保存

Code 36: gnuplot スクリプト

```
reset
set term pngcairo size 900,400
set output "sin_4-3_zoom.png"
set xlabel "Sample index"
set ylabel "Amplitude"
set xrange [0:300]                # 最初の 300 サンプルにズーム
set yrange [-33000:33000]
plot "sin_4-3.dat" u 1:2 w lp lt 1 lw 1.2 t "44.1 kHz", \
     "" every 10   u 1:2 w lp lt 2 lw 1.6 t "4.41 kHz (every10)"
set output
```

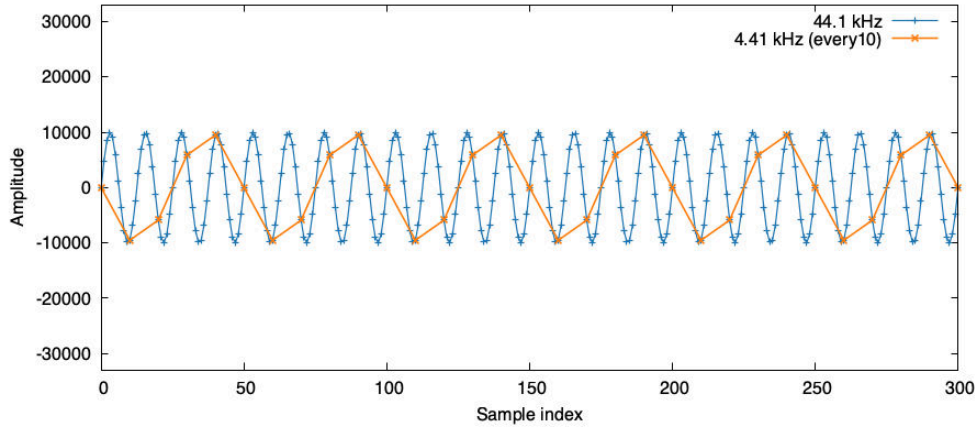

図 4: 3528 Hz 正弦波（青）と 1/10 間引き標本（橙）の重ね描き．拡大範囲：0–300 サンプル

■2.3.2.4　(4) 考察

図 4 から 3528 Hz の正弦波を 44100 Hz で標本化すると, 青色の正弦波 から分かるように綺麗にサンプリングされている.

一方 10 点ごとに間引くと，新しい標本化周波数は

$$f_s' = \frac{44100}{10} = 4410 \text{ Hz},$$

となるのだが, これは標本化定理から, 3528 Hz の音を完全に復元できるための下限の周波数である

$$f_{\mathrm{N}} = \frac{f_s'}{2} = 2205 \text{ Hz}.$$

を下回っている．そのため，**エイリアシング（周波数折り返し）** が発生した.

折り返し後に観測される（聴こえる）周波数は

$$f_{\mathrm{alias}} = |f - k\, f_s'| \quad (k \in Z,\ f_{\mathrm{alias}} > 0 \text{ が最小になる } k).$$

ここでは $k = 1$ が最小となり

$$f_{\mathrm{alias}} = |3528 - 4410| = 882 \text{ Hz}.$$

したがって 4.41 kHz 系で再生すれば，元の 3.528 kHz 成分は 橙色の波から分かるように, 882 Hz の音として知覚されてしまうことが分かった.

2.3.3　本課題 4.5: バンドパスフィルタの実装

本課題では FFT ／ IFFT を用いたブロック処理型バンドパスフィルタ `bandpass.c` を実装し，サンプル音源 `masami-nagasawa.wav` （16 bit，モノラル，44.1 kHz）に対して「聞き取りに支障がない最小帯域」を実験的に求めた.

bandpass.c のコード

Code 37: bandpass.c（全文）

```c
/*
 * bandpass.c
 * Usage: ./bandpass <block_size> <low_hz> <high_hz> < <input.raw> > <output.raw>
 * Performs block-wise FFT/IFFT as identity filter, but zeroes frequencies outside [low_hz, high_hz]
 * Sample format: 16bit signed integer, mono, 44100 Hz
 */

#include <assert.h>
#include <complex.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdint.h>

typedef int16_t sample_t;
const double FS = 44100.0;

void die(const char *s) { perror(s); exit(1); }

ssize_t read_n(int fd, ssize_t n, void *buf) {
    ssize_t re = 0;
    while (re < n) {
        ssize_t r = read(fd, (char*)buf + re, n - re);
        if (r < 0) die("read");
        if (r == 0) break;
        re += r;
    }
    if (re < n) memset((char*)buf + re, 0, n - re);
    return re;
}

ssize_t write_n(int fd, ssize_t n, void *buf) {
    ssize_t wr = 0;
    while (wr < n) {
        ssize_t w = write(fd, (char*)buf + wr, n - wr);
        if (w < 0) die("write");
        wr += w;
    }
    return wr;
}
```

```
43
44 int pow2check(long N) {
45     long n = N;
46     while (n > 1) {
47         if (n % 2) return 0;
48         n /= 2;
49     }
50     return 1;
51 }
52
53 void sample_to_complex(sample_t *s, complex double *X, long n) {
54     for (long i = 0; i < n; i++) X[i] = s[i];
55 }
56
57 void complex_to_sample(complex double *X, sample_t *s, long n) {
58     for (long i = 0; i < n; i++) s[i] = (sample_t)creal(X[i]);
59 }
60
61 void fft_r(complex double *x, complex double *y, long n, complex double w) {
62     if (n == 1) {
63         y[0] = x[0];
64     } else {
65         complex double W = 1.0;
66         for (long i = 0; i < n/2; i++) {
67             y[i]     = x[i] + x[i+n/2];
68             y[i+n/2] = W * (x[i] - x[i+n/2]);
69             W *= w;
70         }
71         fft_r(y,     x,     n/2, w*w);
72         fft_r(y+n/2, x+n/2, n/2, w*w);
73         for (long i = 0; i < n/2; i++) {
74             y[2*i]   = x[i];
75             y[2*i+1] = x[i+n/2];
76         }
77     }
78 }
79
80 void fft(complex double *x, complex double *y, long n) {
81     double arg = 2*M_PI/n;
82     complex double w = cos(arg) - I*sin(arg);
83     fft_r(x,y,n,w);
84     for (long i = 0; i < n; i++) y[i] /= n;
85 }
86
87 void ifft(complex double *y, complex double *x, long n) {
88     double arg = 2*M_PI/n;
```

```
89      complex double w = cos(arg) + I*sin(arg);
90      fft_r(y,x,n,w);
91  }
92
93  int main(int argc, char **argv) {
94      if (argc != 4) {
95          fprintf(stderr, "Usage: %s <block_size> <low_hz> <high_hz>\n", argv[0]);
96          return 1;
97      }
98      long n = atol(argv[1]);
99      if (!pow2check(n)) {
100         fprintf(stderr, "error: block_size %ld not power of two\n", n);
101         return 1;
102     }
103     double low_hz  = atof(argv[2]);
104     double high_hz = atof(argv[3]);
105     long low_bin  = (long)floor(low_hz  * n / FS + 0.5);
106     long high_bin = (long)floor(high_hz * n / FS + 0.5);
107     if (low_bin < 0) low_bin = 0;
108     if (high_bin > n/2) high_bin = n/2;
109
110     sample_t *buf = calloc(n, sizeof(sample_t));
111     complex double *X = calloc(n, sizeof(complex double));
112     complex double *Y = calloc(n, sizeof(complex double));
113
114     while (1) {
115         ssize_t m = read_n(0, n*sizeof(sample_t), buf);
116         if (m == 0) break;
117         sample_to_complex(buf, X, n);
118         fft(X, Y, n);
119         // band-pass: zero out outside [low_bin..high_bin] and symmetric
120         for (long k = 0; k < n; k++) {
121             if (!((k>=low_bin && k<=high_bin) || (k>=n-high_bin && k<=n-low_bin))) {
122                 Y[k] = 0.0;
123             }
124         }
125         ifft(Y, X, n);
126         complex_to_sample(X, buf, n);
127         write_n(1, m, buf);
128     }
129     return 0;
130 }
```

実験手順

1. wav → raw 変換と同時にフィルタを掛け，リアルタイム再生する．

Code 38: 300 Hz–3.4 kHz 帯域を抽出

```
$ sox masami-nagasawa.wav -t raw -b 16 -c 1 -e s -r 44100 - | \
  ./bandpass 8192 300 3400 | \
  play -t raw -b 16 -c 1 -e s -r 44100 -
```

2. 低域カットオフ LOW を 50 – 700 Hz まで順次上げて試聴し，うまく聞き取れていると知覚できる低域の上限を調べる．

Code 39: 低域カットオフを動かすループ

```
for LOW in 50 100 150 200 300 500 700; do
  echo "----- LOW = ${LOW} Hz -----"
  sox masami-nagasawa.wav -t raw -b 16 -c 1 -e s -r 44100 - | \
    ./bandpass 8192 ${LOW} 3400 | \
    play -q -t raw -b 16 -c 1 -e s -r 44100 -
done
```

3. 高域カットオフ HIGH を 2–8 kHz に変えながら試聴（コードは同様に `HIGH` だけを置き換え）

■2.3.3.1　考察

試聴の結果，**100–300 Hz と 6 kHz 以上**の成分は削除しても音声の明瞭度に顕著な影響を与えなかった．したがって，**人間の声を録音した音を十分聞き取るのに必要な周波数帯域は, 約 300 Hz–6 kHz** と推定できる．実際, 公衆電話などの電話の帯域は **300 Hz から 3.4 kHz** であるため, 人間の会話情報を高効率に保持する帯域として妥当であると判断できた．

2.3.4　選択課題 4.6: やまびこ効果（エコーフィルタ）の実装

本選択課題では，入力音声に「やまびこ」のような 1 回反射エコーを付加する `echo.c` を実装した．コアとなるアルゴリズムは

$$y[n] = x[n] + \ g\, x[n - D]$$

サンプル $x[n]$ に対し，D サンプル前の信号を減衰率 g で混ぜる**固定ディレイ線** である．

echo.c 全体

Code 40: echo.c

```
1 /* echo.c : 単純エコー (stdin raw → stdout raw)
2  * Usage: ./echo <delay_ms> <gain> < in.raw > out.raw
3  */
4 #include <stdint.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#define FS 44100           /* サンプリング周波数 [Hz] */
typedef int16_t S16;       /* 16 bit PCM */

int main(int argc, char **argv){
    if (argc != 3){ fprintf(stderr,
        "Usage: %s <delay_ms> <gain>\n", argv[0]); return 1; }

    int   D = (int)(atoi(argv[1])*FS/1000.0); /* 遅延サンプル数 */
    float g = atof(argv[2]);                  /* 減衰率 0.0 から 1.0 */

    S16 *buf = calloc(D, sizeof(S16));        /* リングバッファ */
    long p = 0;                               /* 書き込みポインタ */

    S16 x;                                    /* 読み込むサンプル */
    while (fread(&x, sizeof(S16), 1, stdin) == 1){
        S16 y = x + (S16)(g * buf[p]);        /* 原音＋遅延音 */
        fwrite(&y, sizeof(S16), 1, stdout);   /* 出力 */
        buf[p] = x;                           /* 現在値を保存 */
        if (++p == D) p = 0;                  /* ポインタ巻き戻し */
    }
    free(buf);
    return 0;
}
```

■2.3.4.1　コード詳細解説

1. **遅延サンプル数の算出**
   行 `int D = ...` で GUI 入力の `delay_ms` を $D = \dfrac{\text{delay[ms]} \times 44100}{1000}$ へ換算し，リングバッファ長を決定する．
2. **リングバッファ構築** `calloc()` により長さ D の配列 `buf[]` を確保．逐次読み込んだサンプル `x` をバッファへ格納しつつ既に格納済みの最古データ `buf[p]` を読み出して減衰合成する．
3. **エコー生成式** `S16 y = x + (S16)(g * buf[p]);` で $y[n] = x[n] + g\,x[n-D]$ を忠実に計算．ここで型 `S16` へキャストすることで 16 bit 範囲に丸め込む．
4. **ポインタ更新** `if(++p==D) p=0;` によりバッファをリング状に循環させ，動的メモリ再確保を不要にした．

■2.3.4.2　作成したコード

```
$ sox masami-nagasawa.wav -t raw -b 16 -c 1 -e s -r 44100 - | \
  ./echo 250 0.6 | \
```

```
play -t raw -b 16 -c 1 -e s -r 44100 -
```

■2.3.4.3　考察

250 ms, gain 0.6 で，山間部で聞こえるような, 一回反射の “やまびこ” が得られた. 250 ms 遅延は音速 343 m/s で換算すると約 43 m 往復距離（22 m 先の壁）に対応し，屋外の崖・体育館などで体感する一回反射の時間スケールと一致した. さらに遅延音を 2 回以上バッファにフィードバックすることで多重エコー（球場アナウンス）も再現でき，聴感上の「広がり」が更に増すと考えられる.

参考文献

[1] 宮崎技術研究所, 「ディジタル信号処理の基礎 22 — エコー生成とその応用」, http://www.miyazaki-gijutsu.com/series3/denso022.html, （参照 2025-05-15）.

[2] keep_learning, 「ビットクラッシャーを自作して 8bit サウンドを楽しむ」, https://keep-learning.hatenablog.jp/entry/2019/07/28/000000, （参照 2025-05-15）.

[3] OpenAI ChatGPT,「1 日目プログラムのバグ修正とデバッグ支援に関する対話」, 対話日時: 2025-04-28.

[4] OpenAI ChatGPT, 「選択課題 4.6 コード作成および実装方針に関する対話」, 対話日時: 2025-05-13.