電子制御工学実験報告書

実験題目　：信号処理プログラミング

報告者　　：4 年 32 番　平田 蓮

提出日　　：2020 年 6 月 18 日

実験日　　：2020 年 5 月 21 日, 5 月 28 日, 6 月 4 日, 6 月 11 日

実験班　　：

共同実験者：

※指導教員記入欄

評価項目	配点	一次チェック ・　・	二次チェック ・　・
記載量	20		
図・表・グラフ	20		
見出し，ページ番号，その他体裁	10		
その他の減点	–		
合計	50		

コメント：

1　目的

アナログ信号をディジタルデータに変換し, ディジタル機器で処理するために必要な基礎事項について学習し, C 言語で基本的な信号処理プログラムを作成する. また音声フォーマットの一つである WAVE ファイルの構造を理解し, 音声データの入出力プログラムを作成する.

2　周期関数の生成と可視化

正弦波のように一定周期ごとに同じ波形が繰り返される関数を**周期関数**と呼ぶ. よく知られている周期関数として, のこぎり波などがある. 本節では周期関数に関する演習を行う.

■**演習 1-1**　任意の孤度 r を区間 $[0 : 2\pi]$ に変換する関数 `rad` を作成せよ.

作成した関数をソースコード 1 に示す.

ソースコード 1　rad.c

```
1 double rad(double r) {
2     if (r >= 0) {
3         return fmod(r, 2 * PI);
4     } else {
5         return 2 * PI - fmod(-r, 2 * PI);
6     }
7 }
8
```

r が正のときは 2π との剰余を取る. r が負のときは, r を正数に変換し, 2π と剰余をとったものを 2π から引くことで変換している.

図 1 に横軸に r, 縦軸に `rad(r)` を取ったグラフを示す. この図から, 変換がうまく行われていることがわかる.

■**演習 1-2**　任意の孤度 r に対して, のこぎり波の振幅値を求める関数 `saw` を作成せよ.

作成した関数をソースコード 2 に示す.

ソースコード 2　saw.c

```
1 double saw(double r) {
2     return 1 - rad(r) / PI;
3 }
4
```

まず与えられた孤度 r を演習 1-1 で実装した `rad` を使い $[0 : 2\pi]$ の範囲に変換する.

r が 2π の倍数の場合に 1 を取り, そこから傾き $-\frac{1}{\pi}$ の形を繰り返すので, 上記のように実装することができた.

図 2 にグラフを示す. うまくのこぎり波が現れていることがみてとれる.

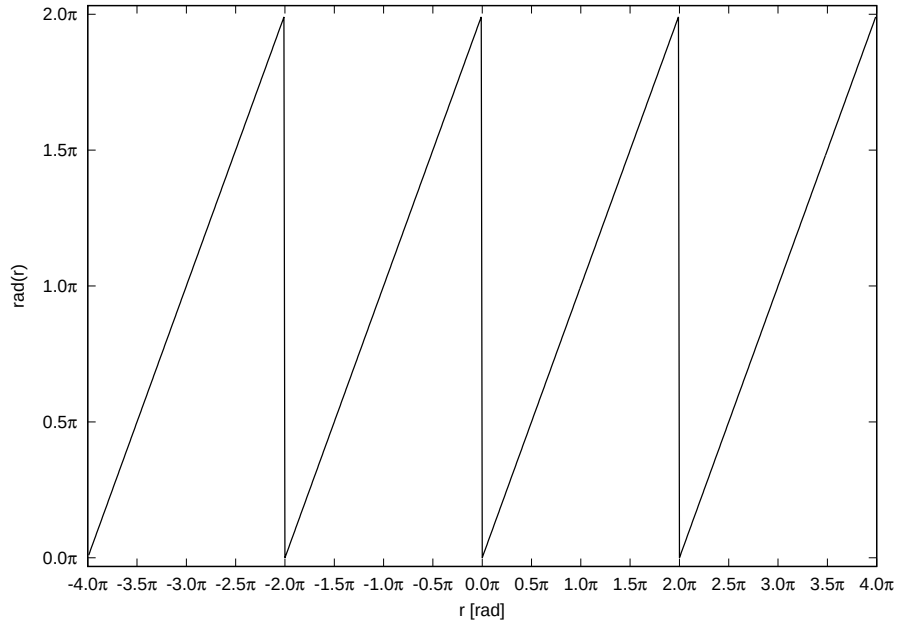

図 1　`rad` による変換

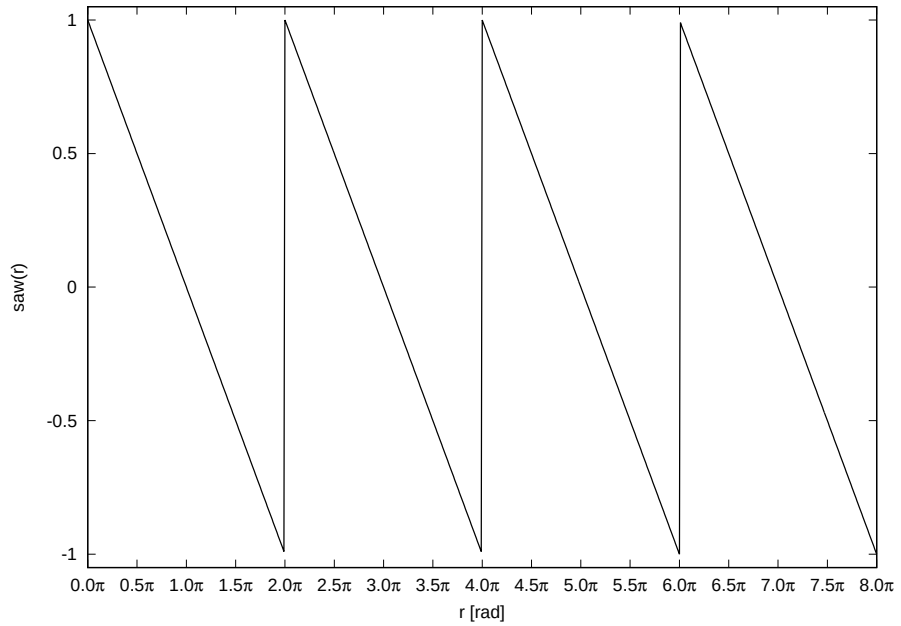

図 2　`saw` による変換

3　サンプリング

アナログ信号波形をディジタルデータに変換する手法に**サンプリング**がある．本節では演習問題を通して実際にサンプリングの様子を観察する．

■**演習 2-1**　リスト 2 のコードを解析し, 完成させよ. 振幅 100, 周波数 4Hz のデータを出力リダイレクションを使って sin100f4.csv に出力せよ. この sin100f4.csv を gnuplot でグラフ化し, サンプリングの効果を確認せよ.

まず, 完成させたリスト 2 のコード, sin10af1.c を出力部分を抜粋して掲載する.

ソースコード 3　sin10af1.c

```
1  int t;
2  double amp, frq, rad, vin;
3  unsigned char vout;
4
5  for (t = 0; t <= TEND; t += DT) {
6      rad = 2 * PI * frq * t / 1000.0;
7      vin = amp * sin(r) + BIAS;
8      if (vin > 255) {
9          vout = 255;
10     } else if (vin < 0) {
11         vout = 0;
12     } else {
13         vout = vin;
14     }
15     printf("%4f, %4d\n", t / 1000.0, vout);
16 }
17
```

8 行目から 14 行目に渡って, クリッピングという処理を施してある. クリッピングをすることで, 出力用の変数 (今回は 8bit の unsigned char 型) に収まらない値を切り捨て, 波形を維持することができる.

図 3 にサンプリングしたデータを示す. 実際に正弦波をサンプリングできていることがわかる.

■**演習 2-2**　振幅 150, 周波数 4Hz のデータ sin150f4.csv を生成しその波形を確認せよ. 波形に不具合があればその原因を考えて不具合を軽減するような修正を行え.

今回は振幅が 150 なので, char 型に収まらない数値をサンプリングすることになる. そこで, 演習 2-1 で実装したクリッピングが働く.

図 4 にクリッピングを施す前と後の波形を示す. この図からクリッピングの効果がみて取れる. 点線のクリッピングをしていない波形では, オーバーフローした値が波形の逆側に飛んでしまい, 波形が崩れているが, 実線のクリッピングを施した方ではオーバーフローした値が切り捨てられ, 波形が維持できている.

■**演習 2-5**　サンプリング定理を満たさないような高い周波数の正弦波を PCM によりディジタルデータ化すると, 周波数や位相が全く異なる波形に見えることがある. この現象を実際に観察せよ.

ソースコード 3 に示した sin10af1.c はサンプリング周波数が 100Hz であるので, 50Hz を超える周波数の波はうまくサンプリングをすることができない.

試しに 98Hz の波形をサンプリングしてみると実際には 98Hz であるはずが, 2Hz の波形のように見える. これは, 実際には一周期以上波形が進んでいるが, サンプリングするときは少しの変化として認識しまっているためである.

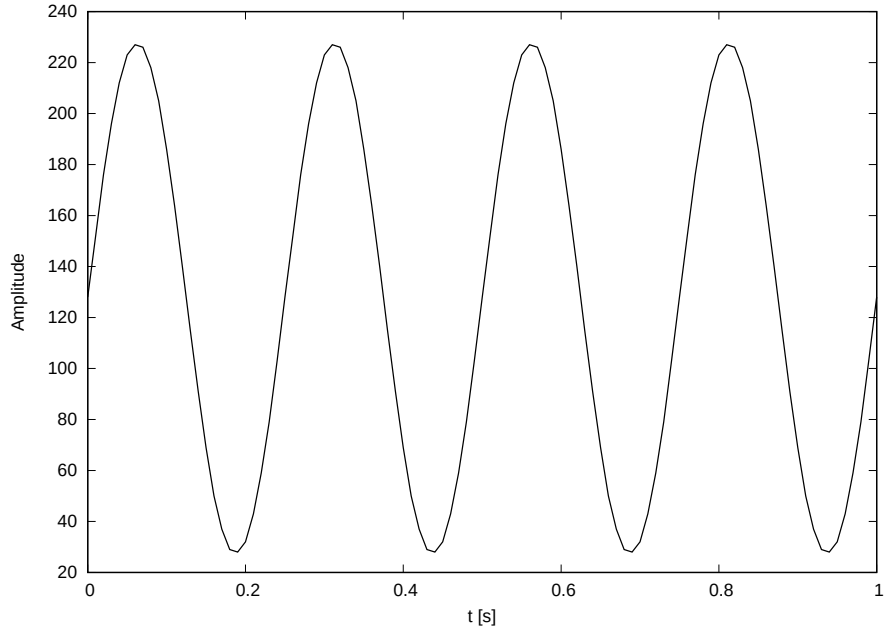

図 3　振幅 100, 周波数 4Hz の正弦波

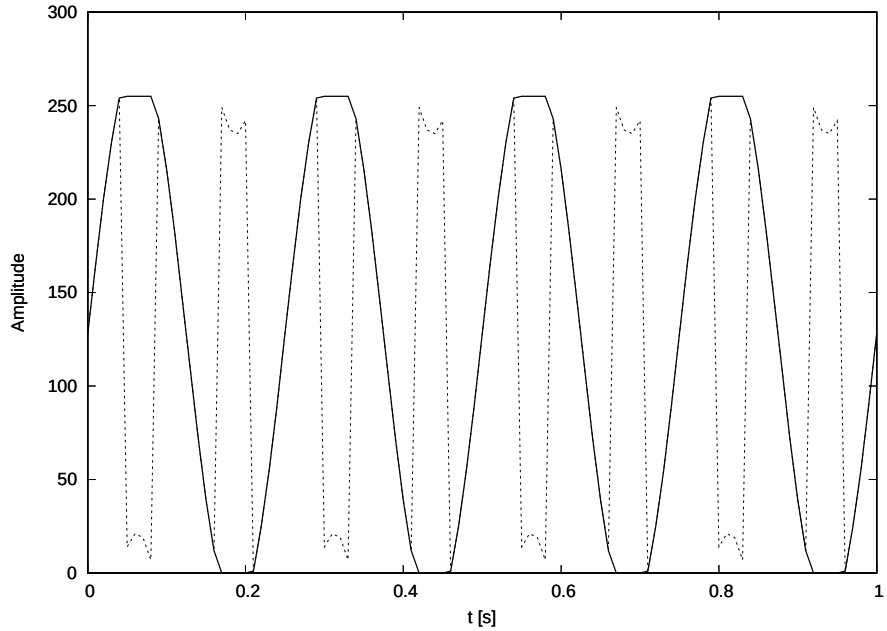

図 4　振幅 150, 周波数 4Hz の正弦波

参考に, 上の波形にサンプリング周波数を上げて計測した本来の 98Hz の波形を重ねたものを図 5 に示す.

4　信号処理プログラムの分類

信号処理プログラムは, オンライン型をオフライン型の二種類に分類することができる. データを一つ取り込むたびに**逐次処理**を行うオンライン型に対して, オフライン型はデータを一定数取り込んだ後に**一括処理**を行う.

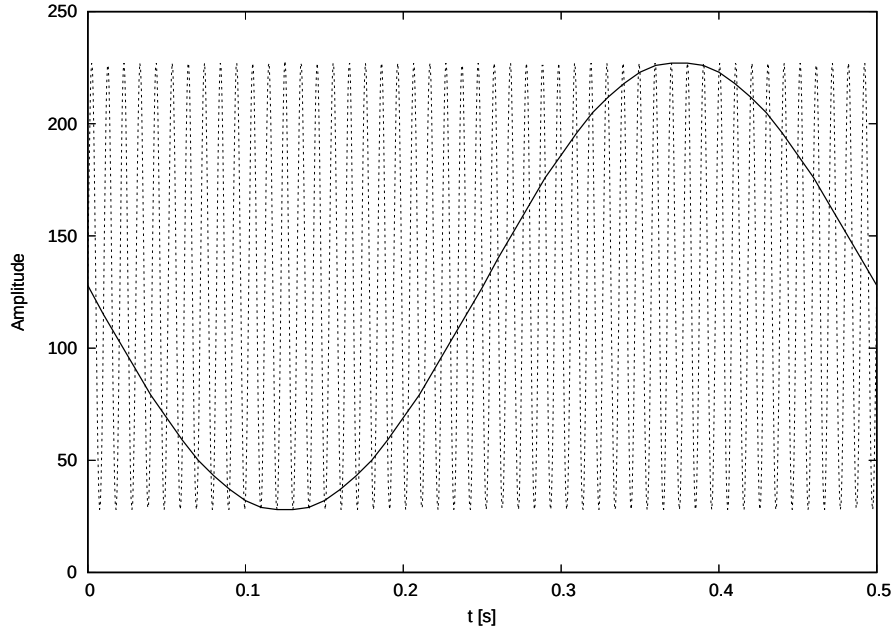

図 5　周波数 98Hz の正弦波

5　雑音除去

信号に重畳された雑音を取り除く処理は, 極めて基本的な信号処理である. 演習 3 では雑音の処理に関する演習を行う.

■演習 3-1　リスト 3 を参考に, コマンドライン引数で元信号の CSV ファイルと, 白色雑音の最大振幅を与えたとき, 白色雑音を加えるオフライン型のプログラム add-wn2.c を作成せよ.

作成したプログラムの信号処理部分を抜粋してソースコード 4 に示す.

ソースコード 4　add-wn2.c

```
1  double tm[DATANUM];
2  int amp[DATANUM], nmax;
3  double err;
4
5  for (int n = 0; n <= DATANUM; n++) {
6      err = nmax * (2.0 * rand() / RAND_MAX - 1.0);
7      amp[n] += ROUND(err);
8
9      printf("%4d, %4d\n", tm[n], amp[n]);
10 }
11
```

通常の sin 波に最大振幅 10 の白色雑音を加えたものを図 6 に示す. 点線が元の正弦波で, 雑音が加えられたものが実線で描いてある. 図から, 適当な雑音が加えられていることがわかる.

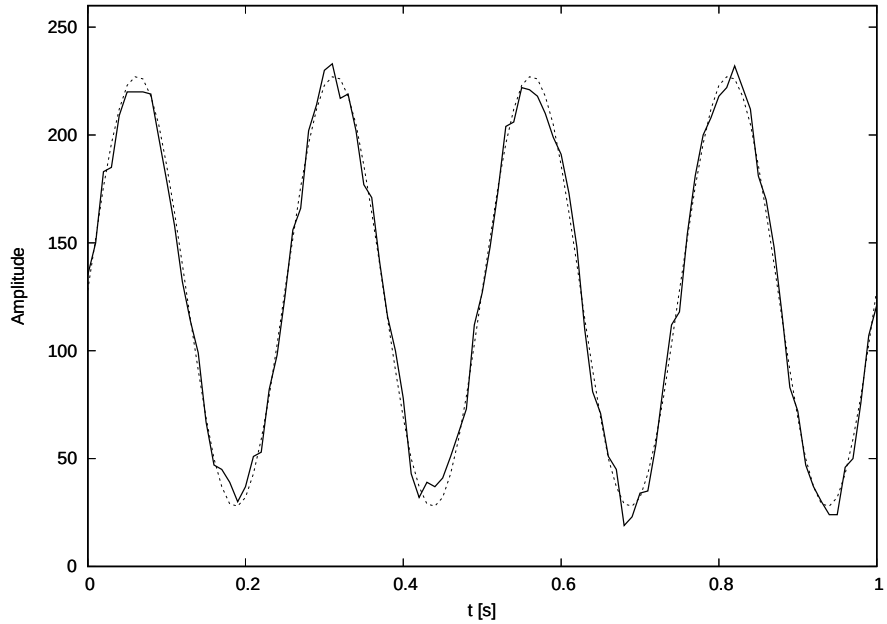

図 6　最大振幅 10 の正弦波を加えた sin 波

■演習 3-2　三点単純移動平均プログラム mvave3-?.c を作成せよ (?はオンライン型は 1, オフライン型は 2).

ソースコード 5, 6 にそれぞれオンライン型とオフライン型のソースコードを抜粋して示す.

ソースコード 5　mvave3-1.c

```
1  int tm, __amp = 0, _amp = 0, amp, _tm;
2
3  while (fgets(buf, sizeof(buf), fp) != NULL) {
4      if (buf[0] == '#') continue;
5
6      tm  = atoi(strtok(buf, ","));
7      amp = atoi(strtok(NULL, "\r\n\0"));
8
9      if (_amp == 0) {
10         _amp = amp;
11         continue;
12     }
13     if (__amp == 0) {
14         __amp = _amp;
15         _tm = tm;
16         continue;
17     }
18
19     vout = ROUND((__amp + _amp + amp) / 3.0);
20     printf("%4d, %4d\n", _tm, (int)vout);
21
22     __amp = _amp;
```

```
23      _amp  = amp;
24      _tm   = tm;
25  }
26
```

ソースコード 6　mvave3-2.c

```
1  double tm[DATANUM], vout;
2  int amp[DATANUM];
3
4  for (int n = 1; n < DATANUM; n++) {
5      vout = ROUND((amp[n - 1] + amp[n] + amp[n + 1]) / 3.0);
6
7      printf("%4d, %4d\n", tm[n], (int)vout);
8  }
9
```

オンライン型のソースコードでは, 三点平均を計算するために二個前までのデータ, また, 出力用に一個前の t を毎回更新している.

これらのプログラムで図 6 から雑音を取り除いた波形を図 7 に示す.

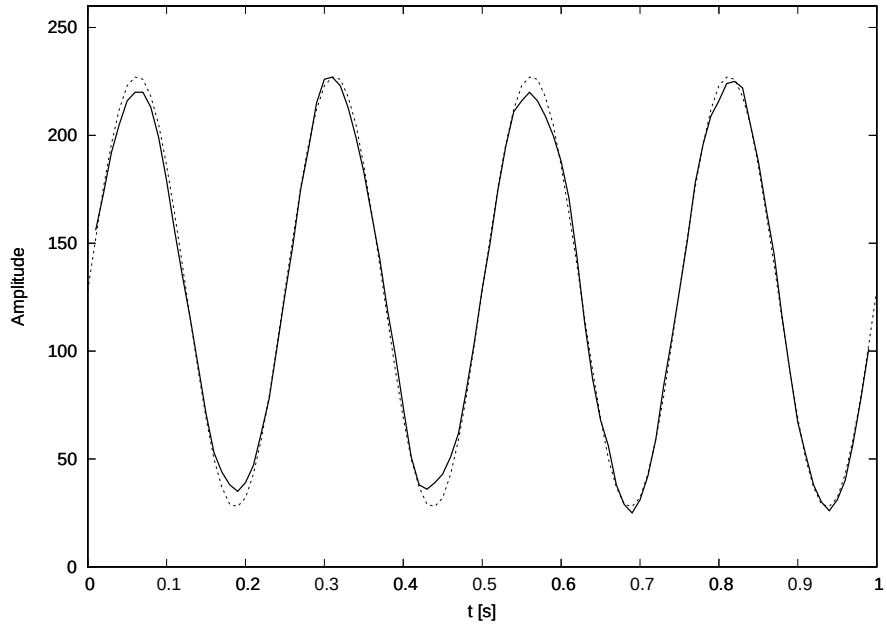

図 7　雑音を取り除いた波形

三点平均を取る過程で始点と終点のデータが一つずつ欠けてしまっているが, 波形は概ね元のものに近づいている.

6　時系列データの解析

周期的なアナログ信号 $x(t)$ に由来する時系列データ $x_i; i = 1, 2, \ldots, N$ が与えられたとき, 元の信号の基本周波数, 振幅, 位相などを求めることを信号解析を呼ぶ. 本実験では信号の**最小値**, **最大値**, **平均値**, **標準偏差**, **最大振幅**, **実効値**を求める.

■平均値と標準偏差　時系列データ $x_i; i = 1, 2, \ldots, N$ の平均値 $\bar{x}$ は次式で定義される:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{1}$$

標準偏差 σ は分散 σ^2 の正の平方根として定義される:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \tag{2}$$

この式では平均値が既知である必要があり, オンライン処理に適用することができない. オンライン処理で標準偏差や分散を計算したい場合は, 変形した次式を用いる:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \bar{x}^2 \tag{3}$$

■演習 4-1　式 (2) を式 (3) に変形する過程を示せ.

$$\begin{aligned}
\sigma^2 &= \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 \\
&= \frac{1}{N}\sum_{i=1}^{N}(x_i^2 - 2x_i\bar{x} + \bar{x}^2) \\
&= \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \frac{2}{N}\bar{x}\sum_{i=1}^{N} x_i + \frac{\bar{x}^2}{N}\sum_{i=1}^{N} 1
\end{aligned}$$

式 (1) より, $\frac{2}{N}\bar{x}\sum_{i=1}^{N} x_i = 2\bar{x}^2$ なので,

$$\begin{aligned}
\sigma^2 &= \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \frac{2}{N}\bar{x}\sum_{i=1}^{N} x_i + \frac{\bar{x}^2}{N}\sum_{i=1}^{N} 1 \\
&= \frac{1}{N}\sum_{i=1}^{N} x_i^2 - 2\bar{x}^2 + \bar{x}^2 \\
&= \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \bar{x}^2
\end{aligned}$$

■**演習 4-2**　コマンドライン引数で与えられた CSV ファイルについて, 最小値, 最大値, 平均値, 標準偏差, 最大振幅, 実効値を求めるプログラム stat?.c を作成せよ (?はオンライン型は 1, オフライン型は 2).

作成したプログラムをソースコード 7, 8 に示す.

ソースコード 7　stat1.c

```
1  int tm, amp;
2  char buf[BUFSIZE];
3  double
4      min_val = INF, max_val = -INF,
5      sum = 0, mean, sum_squ = 0,
6      std_dev, data_num = 0, max_amp, rms = 0;
7  FILE *fp;
8
9  while (fgets(buf, sizeof(buf), fp) != NULL) {
10     if (buf[0] == '#') continue;
11
12     data_num += 1;
13
14     tm  = atoi(strtok(buf, ","));
15     amp = atoi(strtok(NULL, "\r\n\0"));
16
17     if (min_val > amp) min_val = amp;
18     if (max_val < amp) max_val = amp;
19
20     sum += amp;
21
22     sum_squ += amp * amp;
23
24     rms += amp - BIAS;
25 }
26 fclose(fp);
27
28 mean = sum / data_num;
29 std_dev = sqrt(sum_squ / data_num -  mean * mean);
30
31 if (BIAS - min_val > max_val - BIAS) {
32     max_amp = BIAS - min_val;
33 } else {
34     max_amp = max_val - BIAS;
35 }
36
37 rms = sqrt(sum_squ / data_num - 2 * mean * max_amp + max_amp * max_amp);
38
39 printf(
40     "Min: %f, Max: %f, Mean: %f, Std Deviation: %f, Max Amplitude: %f, RMS: %f\n",
41      min_val, max_val, mean,     std_dev,            max_amp,            rms
```

```
);

```

ソースコード 8　stat2.c

```
int tm[DATANUM], amp[DATANUM];
char buf[BUFSIZE];
double
    min_val = INF, max_val = -INF,
    sum = 0, mean, sum_squ = 0, std_dev,
    data_num = 0, max_amp, rms;
FILE *fp;

for (int n = 0; n <= DATANUM; n++) {
    if (min_val > amp[n]) min_val = amp[n];
    if (max_val < amp[n]) max_val = amp[n];

    sum += amp[n];

    sum_squ += amp[n] * amp[n];

    rms += amp - BIAS;
}

mean = sum / data_num;
std_dev = sqrt(sum_squ / data_num -  mean * mean);

if (BIAS - min_val > max_val - BIAS) {
    max_amp = BIAS - min_val;
} else {
    max_amp = max_val - BIAS;
}

rms = sqrt(sum_squ / data_num - 2 * mean * max_amp + max_amp * max_amp);

printf(
    "Min: %f, Max: %f, Mean: %f, Std Deviation: %f, Max Amplitude: %f, RMS: %f\n",
     min_val, max_val, mean,     std_dev,            max_amp,            rms
);

```

試しに, 図 3 のデータを入力した時のデータを表 1 にまとめる.

若干の誤差が見受けられるものの, ある程度の精度の値が出ている. 誤差の原因として考えられるのは, サンプリング周波数の関係で最大値が測定できていないことなどがある.

表 1　sin 波の解析

	測定値	理論値
最小値	28	28
最大値	227	228
平均値	127.5	128
標準偏差	70.42	70.71
最大振幅	100	100
実効値	75.61	70.71

7　Windows WAVE ファイルの解析 · 加工

この節では, Windows 標準のサウンドフォーマットである WAVE ファイルを, 自作の C プログラムで読み書きをする.

■**演習 5-1**　リスト 6 を完成させ, もよりの WAVE ファイルのいくつかについてヘッダ情報を調べ, 表に整理せよ.

今回はサポートページ [1] で配布されている音源を使用した. ヘッダ情報と音源の再生時間を表 2 にまとめる.

表 2　WAVE ファイルのヘッダ情報

	ringout.wav	ringin.wav	chimes.wav	timetone.wav
RIFF 情報サイズ [byte]	5204	10018	55768	77609
fmt チャンクサイズ	16	16	16	16
Format ID	1	1	1	1
チャンネル数	1	1	2	1
サンプリングレート [Hz]	11025	11025	22050	32000
データ速度 [byte/s]	11025	11025	88200	32000
ブロックサイズ [byte]	1	1	4	1
量子化ビット数 [bit]	8	8	16	8
データチャンクサイズ [byte]	5167	9981	55684	77573
時間 [s]	0.469	0.905	0.631	2.424

表から, 量子化ビットやサンプリングレートが増えるほどデータサイズが増えることがわかる.

■**演習 5-2**　モノラル音声 · 量子化ビット数 8 の WAVE ファイルの波形データを CSV ファイルに吐き出すダンププログラム wav2txt-m8.c を作成せよ.

作成した関数 `read_data` と一部抜粋した wav2txt-m8.c をソースコード 9, 10 に示す. また, リスト 6 にて作成し

た関数 read_head の戻り値をデータのサンプリングレートに変更してある.

ソースコード 9　read_data.c

```
1  void read_data(FILE *fp, uLong smprate, int start, int end) {
2      int amp;
3      double now = 0;
4      while ((amp = fgetc(fp)) != EOF) {
5          if (now > end) break;
6          if (now >= start) printf("%f, %d\n", now, amp);
7          now += 1000.0 / smprate;
8      }
9  }
10
```

ソースコード 10　wav2txt-m8.c

```
1  uShort ch, qbit;
2  FILE *fp;
3
4  read_data(fp, read_head(fp, &ch, &qbit), atoi(argv[2]), atoi(argv[3]));
5
```

引数の fp にはヘッダを読み取った後のファイルポインタ, start と end にはミリ秒単位の開始秒数と終了秒数を指定する.

実際にダンプした 3 つのデータを以下に示す.

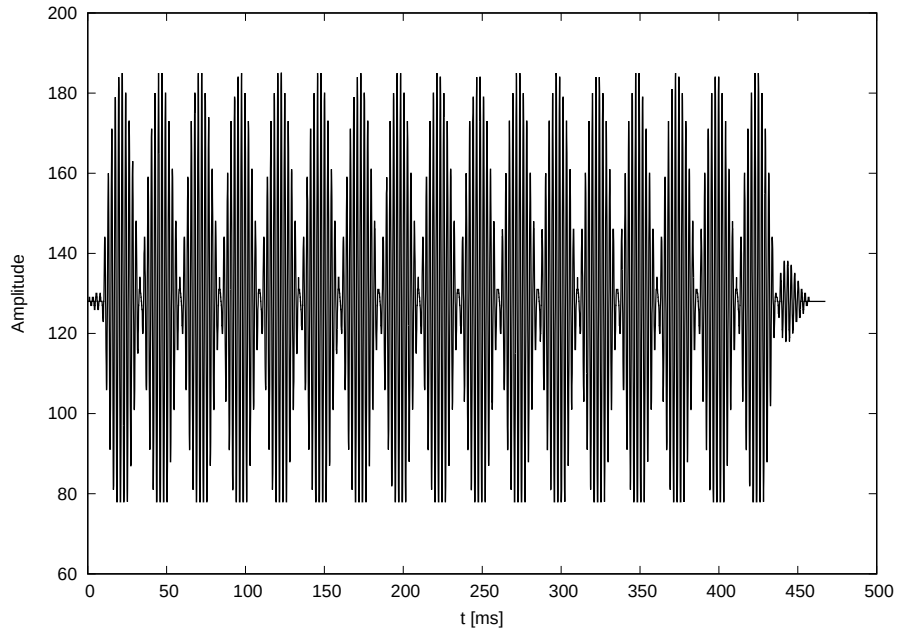

図 8　ringout.wav

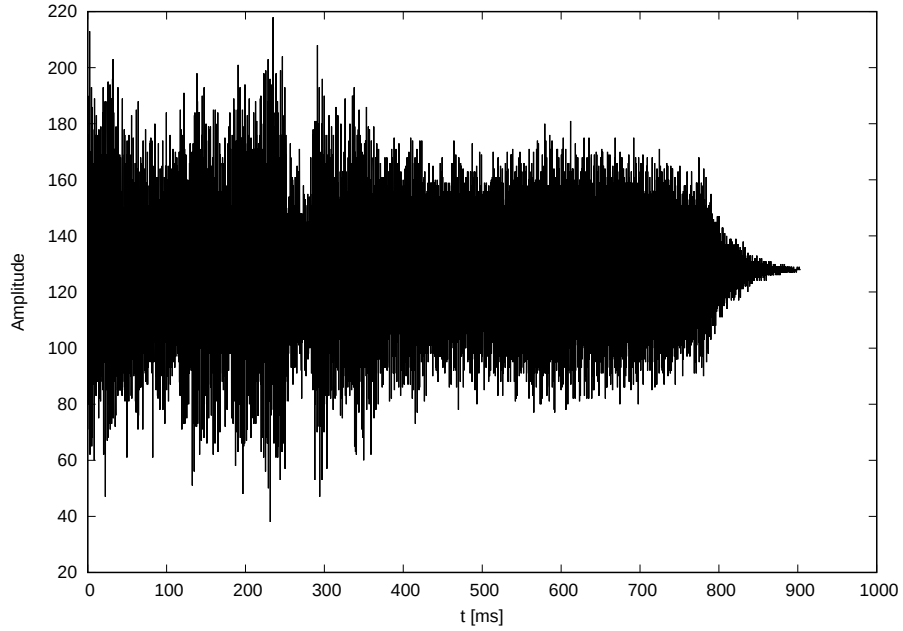

図 9　ringin.wav

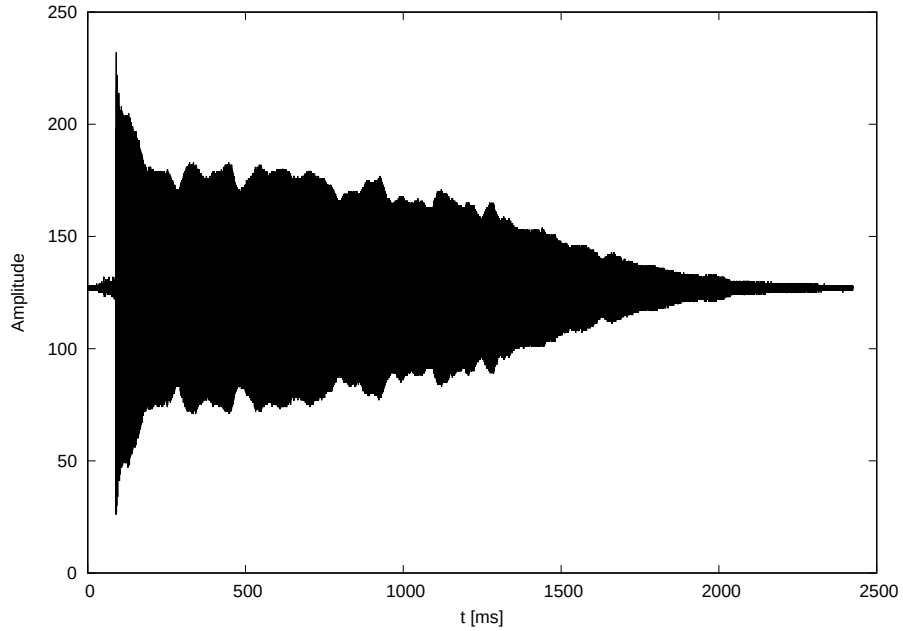

図 10　timetone.wav

■演習 5-4　ユーザが指定した周波数と振幅の正弦波を, モノラル音声で量子化ビット数 8, サンプリング周波数 11025Hz で 1 秒の長さだけ記録するプログラム rec-m8.c を作成し, 動作を確かめよ.

作成した rec-m8.c をソースコード 11 に示す.

ソースコード 11　rec-m8.c

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

```
2  #include <stdlib.h>
3  #include <math.h>
4  #include "pi.h"
5
6  #define BIAS 0x80
7  #define RIFF_SIZE 11061
8  #define FMT_SIZE 16
9  #define FORMAT_ID 1
10 #define SAMPLING_RATE 11025
11 #define CHANNEL_N 1
12 #define BLOCK_SIZE 1
13 #define Q_BIT 8
14
15 void write_header(FILE *fp) {
16     int
17         riff_size     = RIFF_SIZE,
18         fmt_size      = FMT_SIZE,
19         format_id     = FORMAT_ID,
20         channel_n     = CHANNEL_N,
21         sampling_rate = SAMPLING_RATE,
22         block_size    = BLOCK_SIZE,
23         q_bit         = Q_BIT;
24
25     fwrite("RIFF",          sizeof(char),  4, fp);
26     fwrite(&riff_size,     sizeof(int),   1, fp);
27     fwrite("WAVE",          sizeof(char),  4, fp);
28     fwrite("fmt ",          sizeof(char),  4, fp);
29     fwrite(&fmt_size,      sizeof(int),   1, fp);
30     fwrite(&format_id,     sizeof(short), 1, fp);
31     fwrite(&channel_n,     sizeof(short), 1, fp);
32     fwrite(&sampling_rate, sizeof(int),   1, fp);
33     fwrite(&sampling_rate, sizeof(int),   1, fp);
34     fwrite(&block_size,    sizeof(short), 1, fp);
35     fwrite(&q_bit,         sizeof(short), 1, fp);
36     fwrite("data",          sizeof(char),  4, fp);
37     fwrite(&sampling_rate, sizeof(int),   1, fp);
38 }
39
40 int main(int argc, char **argv) {
41     int t;
42     double rad, frq, amp;
43     unsigned char vout;
44     FILE *fp;
45
46     if (argc != 4) {
47         return EXIT_FAILURE;
48     }
```

```
49      amp = atof(argv[1]);
50      frq = atof(argv[2]);
51
52      if ((fp = fopen(argv[3], "a+b")) == NULL) {
53          return EXIT_FAILURE;
54      }
55
56      write_header(fp);
57
58      for (t = 0; t <= SAMPLING_RATE; t++) {
59          rad = 2 * PI * frq * t / (double)SAMPLING_RATE;
60          vout = amp * sin(rad) + BIAS;
61          fwrite(&vout, 1, 1, fp);
62      }
63  }
64
```

コマンドライン引数には順番に, 振幅, 周波数, WAVE ファイル名を与える. `write_header` で WAVE ファイルにヘッダを書き込む. その後, 1 秒間分正弦波のデータをサンプリングしつつファイルに書き込む処理を行っている.

試しに書き込まれた 440Hz のデータをサンプリングし直してグラフに表示してみる.

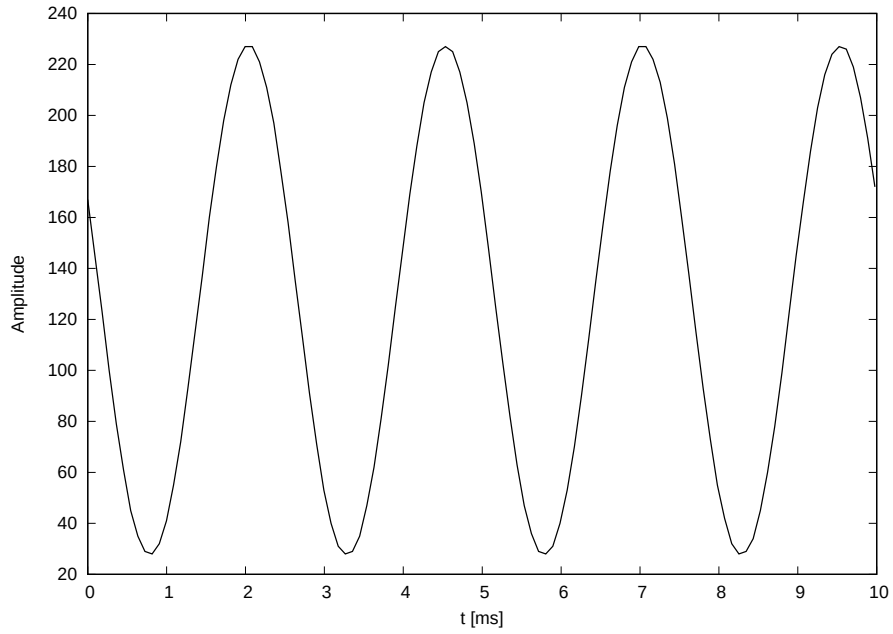

図 11　440Hz の正弦波

440Hz では 1 周期が約 2.27ms なので, 正しく WAVE ファイルに書き込めていることがわかる.

参考文献

[1] Ec4 電子制御工学実験 https://www2.st.nagaoka-ct.ac.jp/