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 8 月 7 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 1-3**　任意の孤度 r に対して, 矩形波の振幅値を求める関数 `squ` を作成せよ.

作成した関数をリスト 1 に示す.

リスト 1　squ.c

```
1 double squ(double r) {
2     return rad(r) < PI ? 1 : -1;
3 }
```

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. その孤度が π 未満の場合に 1 を返し, それ以外の場合は-1 を返すことで矩形波を出力できた.

図 1 に横軸に r, 縦軸に `rad(r)` を取ったグラフを示す. 図より, 矩形波が正しく出力できていることがわかる.

■**演習 1-4**　任意の孤度 r に対して, 三角波の振幅値を求める関数 `tri` を作成せよ.

作成した関数をリスト 2 に示す.

リスト 2　tri.c

```
1 double tri(double r) {
2     r += PI / 2.0;
3     if (rad(r) < PI) {
4         return 2.0 * rad(r) / PI - 1.0;
5     } else {
6         return 2.0 - 2.0 * (rad(r) - PI / 2.0) / PI;
7     }
8 }
```

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. その後, r の位相を変化させた後, 区間 $[0 : \pi), [\pi : 2\pi)$ に分けて処理を行っている.

図 2 に横軸に r, 縦軸に `tri(r)` を取ったグラフを示す. 図より, 三角波が正しく出力できていることがわかる.

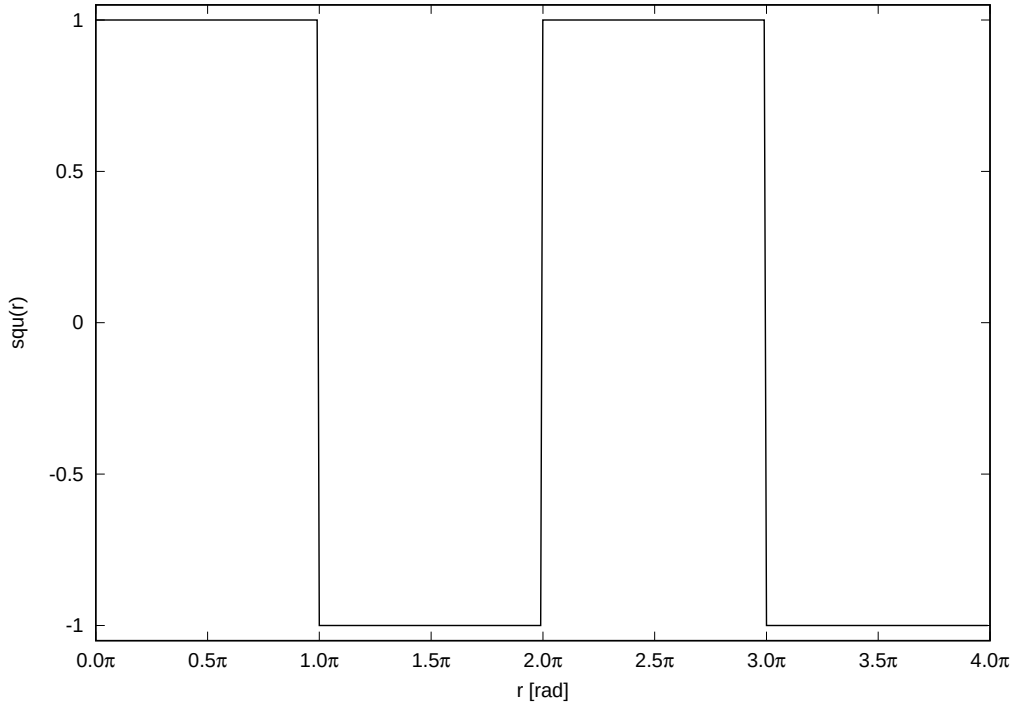

図 1　squ による矩形波

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 2-3**　量子化誤差を評価するため, sin10af1.c に量子化における 2 乗平均平方根誤差を計算する機能を追加せよ．sin10af1.c に追加した部分を一部抜粋してリスト 3 に示す．

リスト 3　sin10af1.c

```
1  #define DT 10
2  #define TEND 1000
3
4  int main(int argc, char **argv) {
5      int t;
6      double vin, esum = 0, rms;
7      unsigned char vout;
8
9      for (t = 0; t <= TEND; t += DT) {
10         esum += (vin - vout) * (vin - vout);
11     }
12
13     rms = sqrt(esum / (TEND / DT + 1));
14     printf("#E_RMS:␣%f\n", rms);
```

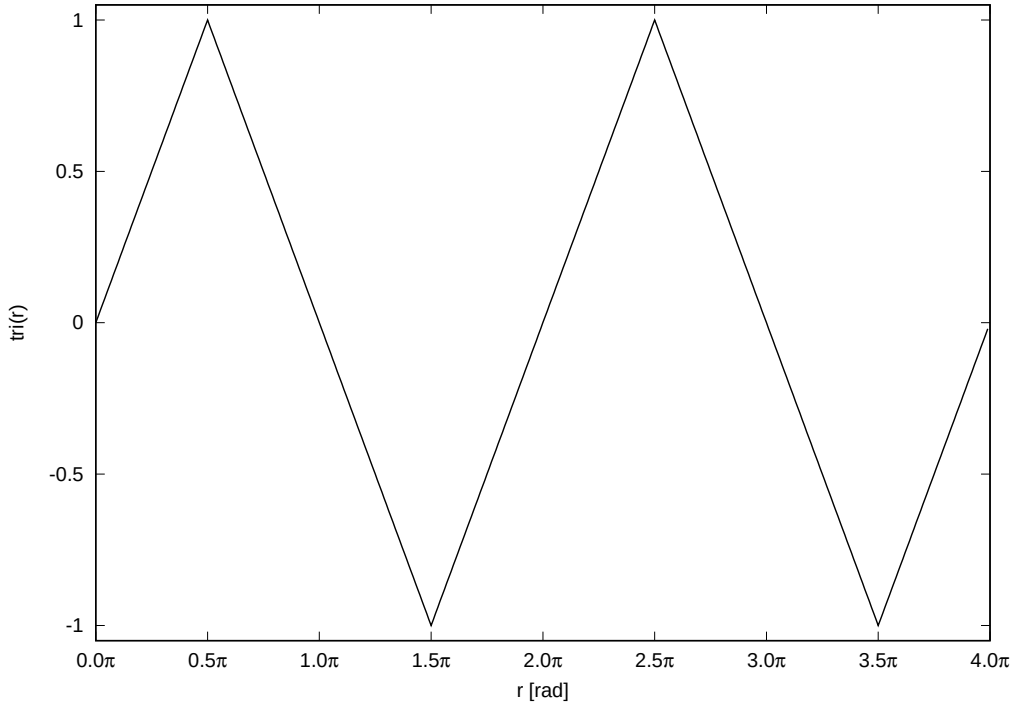

図 2　tri による三角波

```
15 }
```

上のプログラムを使用して振幅 100, 周波数 4 Hz の正弦波をサンプリングしたときの RMS 誤差を求めると, 約 0.559 であった.

■演習 2-4　各時刻の振幅値の小数点以下第 1 位を四捨五入して量子化するプログラム sin10af2.c を作成し, 量子化による RMS 誤差の軽減効果を定量的に調べよ.

作成したプログラムを一部抜粋してリスト 4 に示す.

リスト 4　sin10af2.c

```
1  #define DT 10
2  #define TEND 1000
3
4  int main(int argc, char **argv) {
5      int t;
6      double vin, esum = 0, rms;
7      unsigned char vout;
8
9      for (t = 0; t <= TEND; t += DT) {
10         if (vin > 255) {
11             vout = 255;
12         } else if (vin < 0) {
13             vout = 0;
14         } else {
```

```
15              vout = vin + 0.5;
16          }
17
18          esum += (vin - vout) * (vin - vout);
19      }
20
21      rms = sqrt(esum / (TEND / DT + 1));
22      printf("#E_RMS:␣%f\n", rms);
23  }
```

15 行目で, sin10af1.c では vout = vin としていたところを, 0.5 を足すことで整数値に直されるときに四捨五入されることになる.

このプログラムを使用して演習 2-3 を同じように振幅 100, 周波数 4 Hz の正弦波をサンプリングした時の RMS 誤差を求めると, 約 0.293 であった.

四捨五入をしなかった場合と比べると, 約 0.266 の差があることがわかり, 四捨五入をすることに RMS 誤差の軽減効果が大きくあることがわかる.

■**演習 2-6**　正弦波の振幅, 周波数, 位相, 標本化間隔をコマンドライン引数で指定した時, サンプリング結果を出力するプログラム sinafpt.c を作成せよ.

作成したプログラムをリスト 5 に一部抜粋して示す. ここで, 正弦波の位相は rad 単位の実数で指定することとした.

リスト 5　sinafpt.c

```
1  #define BIAS 0x80
2  #define TEND 1000
3
4  int main(int argc, char **argv) {
5      int t;
6      double amp, frq, smp, phf, r, vin;
7      unsigned char vout;
8
9      for (t = 0; t <= TEND; t += smp) {
10         r = t / 1000.0 * 2.0 * PI * frq + phf;
11         vin = amp * sin(r) + BIAS;
12         if (vin > 255) {
13             vout = 255;
14         } else if (vin < 0) {
15             vout = 0;
16         } else {
17             vout = vin + 0.5;
18         }
19
20         printf("%d,␣%4d\n", t, vout);
21     }
```

```
22 }
```

このプログラムを使用して, 振幅 100, 周波数 5 Hz, 位相 1.57 rad ($\approx \frac{\pi}{2}$ [rad]), 標本化間隔 5 ms の正弦波をサンプリングした結果を図 3 に示す. 図より, 正しくサンプリングされていることがわかる.

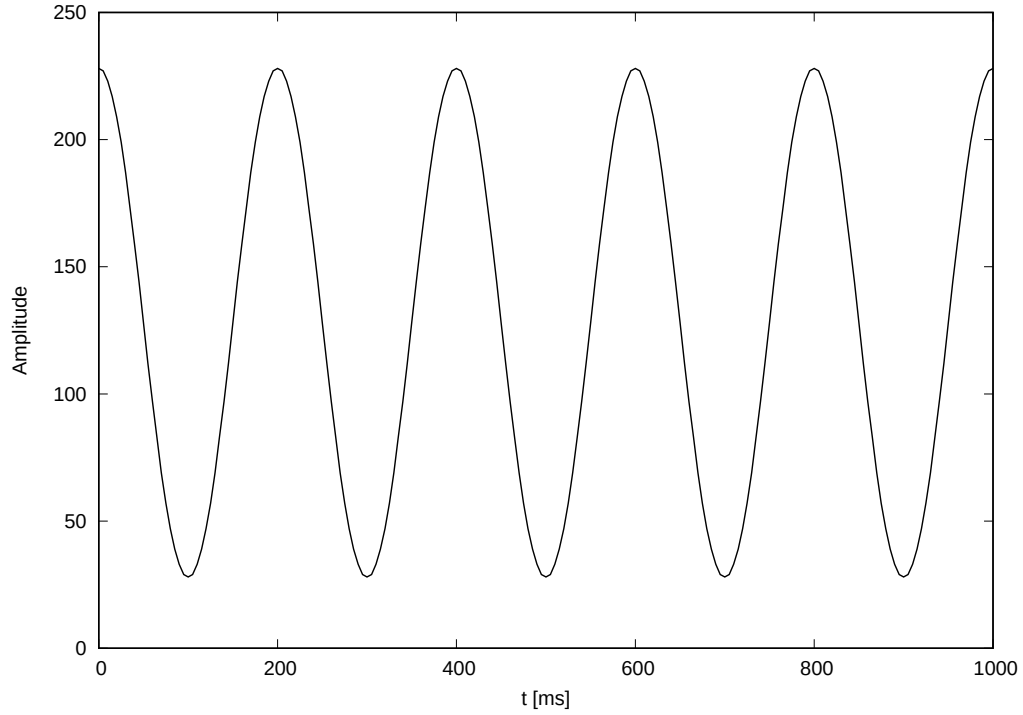

図 3　振幅 100, 周波数 5 Hz, 位相 1.57 rad の正弦波

■**演習 2-7**　振幅, 周波数, 標本化間隔をコマンドライン引数で指定したとき, のこぎり波, 矩形波, 三角波を出力するプログラム XXXaft.c を作成せよ. (XXX はのこぎり波は saw, 矩形波は squ, 三角波は tri)

作成したプログラムを一部抜粋してリスト 6, 7, 8 に示す.

リスト 6　sawaft.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
9  double saw(double r) {
10     return 1 - rad(r) / PI;
11 }
12
13 int main(int argc, char **argv) {
```

```
    int t;
    double r, amp, frq, smp, vin, vout;

    amp = atof(argv[1]);
    frq = atof(argv[2]);
    smp = atof(argv[3]);

    for (t = 0; t <= 1000; t += smp) {
        r = t / 1000.0 * 2.0 * PI * frq;
        vin = amp * saw(r);
        vout = vin;
        printf("%d, %4f\n", t, vout);
    }
}
```

リスト 7　squaft.c

```
double rad(double r) {
    if (r >= 0) {
        return fmod(r, 2 * PI);
    } else {
        return 2 * PI - fmod(-r, 2 * PI);
    }
}

double squ(double r) {
    return rad(r) < PI ? 1 : -1;
}

int main(int argc, char **argv) {
    int t;
    double r, amp, frq, smp, vin, vout;

    amp = atof(argv[1]);
    frq = atof(argv[2]);
    smp = atof(argv[3]);

    for (t = 0; t <= 1000; t += smp) {
        r = t / 1000.0 * 2.0 * PI * frq;
        vin = amp * squ(r);
        vout = vin;
        printf("%d, %4f\n", t, vout);
    }
}
```

リスト 8　triaft.c

```
double rad(double r) {
```

```
    if (r >= 0) {
        return fmod(r, 2 * PI);
    } else {
        return 2 * PI - fmod(-r, 2 * PI);
    }
}

double tri(double r) {
    r += PI / 2.0;
    if (rad(r) < PI) {
        return 2.0 * rad(r) / PI - 1.0;
    } else {
        return 2.0 - 2.0 * (rad(r) - PI / 2.0) / PI;
    }
}

int main(int argc, char **argv) {
    int t;
    double r, amp, frq, smp, vin, vout;

    amp = atof(argv[1]);
    frq = atof(argv[2]);
    smp = atof(argv[3]);

    for (t = 0; t <= 1000; t += smp) {
        r = t / 1000.0 * 2.0 * PI * frq;
        vin = amp * tri(r);
        vout = vin;
        printf("%d,␣%4f\n", t, vout);
    }
}
```

これらのプログラムを使って振幅 2.5, 周波数 3 Hz, 標本化間隔 5 ms ののこぎり波, 矩形波, 三角波をサンプリングしたものを図 4, 5, 6 に示す.

図から, 正しくサンプリングできていることがわかる.

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.

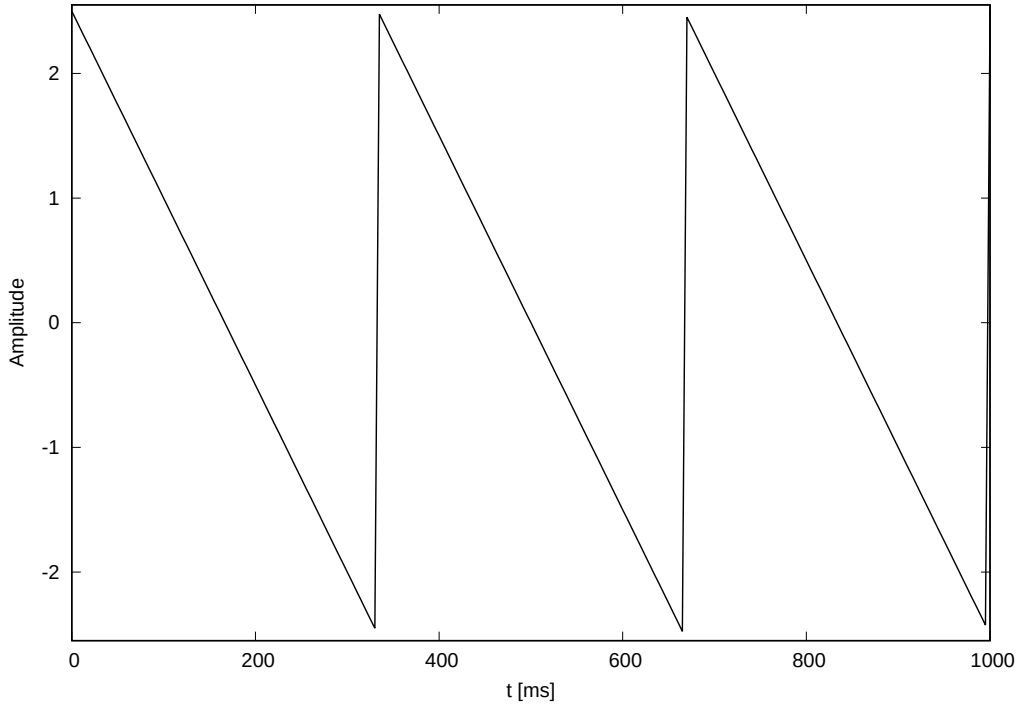

図 4　振幅 2.5, 周波数 3 Hz, 標本化間隔 5 ms ののこぎり波

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 3-3**　5 点単純移動平均プログラム mvave5-?.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リスト 9, 10 に一部抜粋して示す.

リスト 9　mvave5-1.c

```
1  #define BUFSIZE 80
2  #define ROUND(x) ((x > 0) ? (x + 0.5) : (x - 0.5))
3
4  #define POINTS 5
5
6  int main(int argc, char **argv) {
7      int tm[(POINTS + 1) / 2] = {}, amp[POINTS] = {}, sum = 0, pt1 = 0, pt2 = 0;
8      char buf[BUFSIZE];
9      unsigned char vout;
10     FILE *fp;
11
12     while (fgets(buf, sizeof(buf), fp) != NULL) {
13         if (buf[0] == '#') continue;
14
```

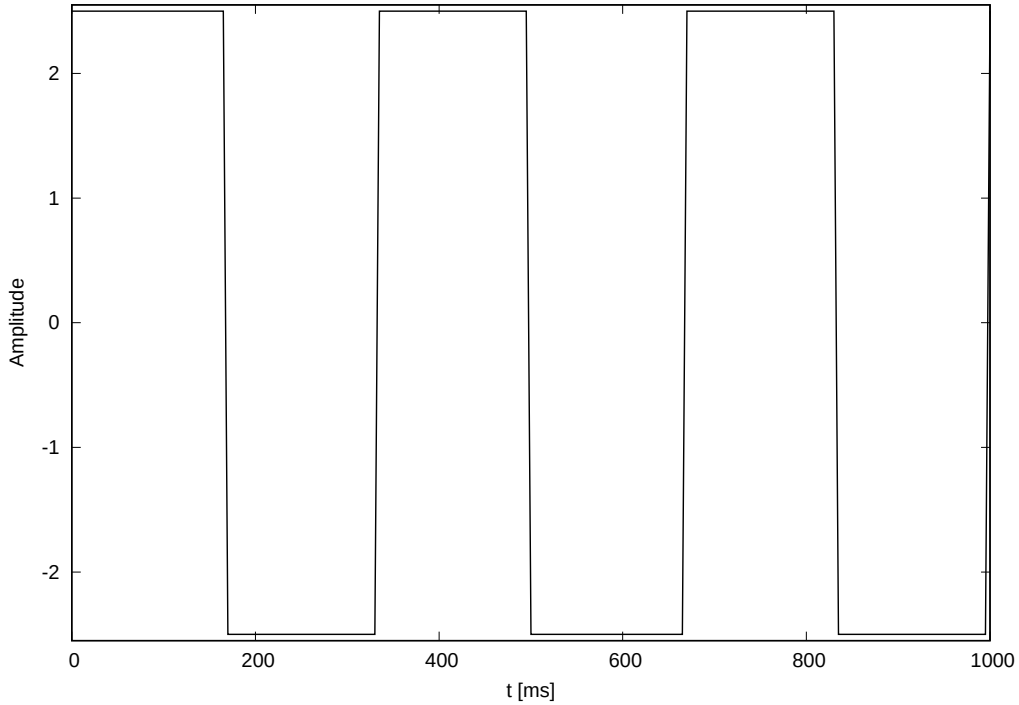

図 5　振幅 2.5, 周波数 3 Hz, 標本化間隔 5 ms の矩形波

```
15          sum -= amp[pt1];
16
17          tm[pt2]  = atoi(strtok(buf, ","));
18          amp[pt1] = atoi(strtok(NULL, "\r\n\0"));
19
20          sum += amp[pt1];
21
22          pt1 = (pt1 + 1) % POINTS;
23          pt2 = (pt2 + 1) % ((POINTS + 1) / 2);
24
25          if (amp[pt1] == 0) continue;
26
27          vout = ROUND(sum / (double)POINTS);
28          printf("%d,␣%4d\n", tm[pt2], vout);
29      }
30  }
```

リスト 10　mvave5-2.c

```
1  #define BUFSIZE 80
2  #define DATANUM 1000
3  #define ROUND(x) ((x > 0) ? (x + 0.5) : (x - 0.5))
4
5  #define POINTS 5
6
```

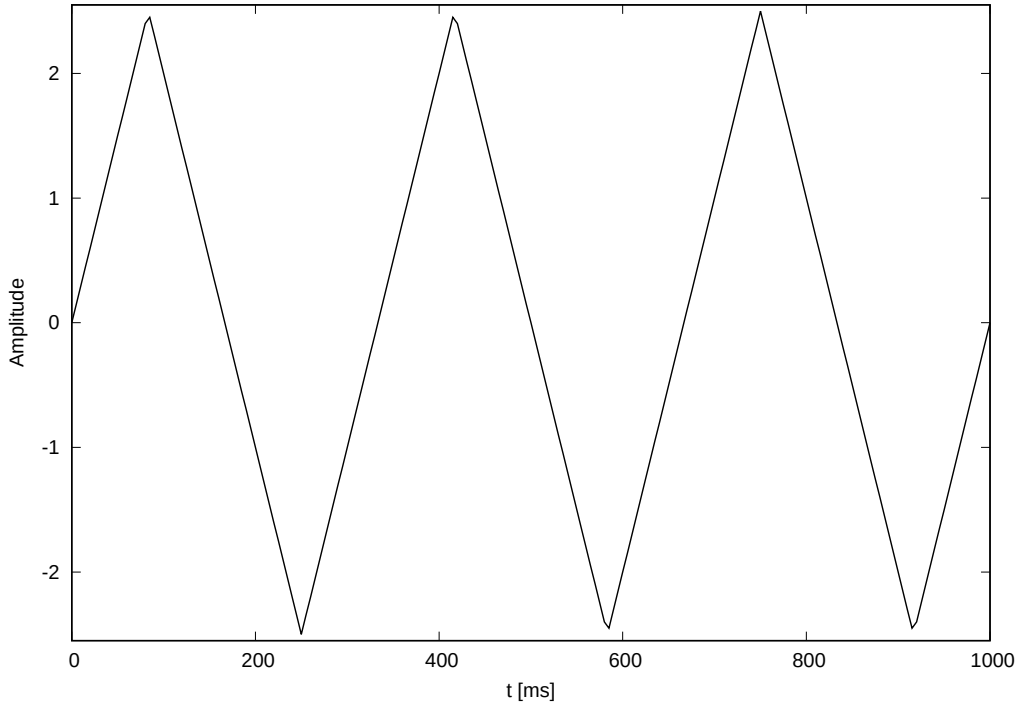

図 6　振幅 2.5, 周波数 3 Hz, 標本化間隔 5 ms の三角波

```
7  int main(int argc, char **argv) {
8      int tm[DATANUM], amp[DATANUM], sum = 0, n, data_num;
9      char buf[BUFSIZE];
10     unsigned char vout;
11     FILE *fp;
12
13     for (n = 0; n < DATANUM; n++) {
14         if (fgets(buf, sizeof(buf), fp) == NULL) break;
15         if (buf[0] == '#') continue;
16         tm[n]  = atoi(strtok(buf, ","));
17         amp[n] = atoi(strtok(NULL, "\r\n\0"));
18     }
19
20     data_num = n;
21
22     for (int n = 0; n < data_num; n++) {
23         sum += amp[n];
24
25         if (n < POINTS - 1) continue;
26         if (n >= POINTS) sum -= amp[n - POINTS];
27
28         vout = ROUND(sum / (double)POINTS);
29
30         printf("%d, %4d\n", tm[n - POINTS / 2], vout);
31     }
```

```
32 }
```

それぞれ, POINTS の値を N にすることで N 点単純移動平均プログラムとすることができる. 直近 5 つのデータと 2 つの時刻を利用することで平均を求める際の計算量を軽減することができた.

これらのプログラムで, 振幅 100, 周波数 4 Hz の正弦波に演習 3-1 で実装した add-wn2.c を使用して最大振幅 10 の白色雑音を加えた波形に雑音除去を行った波形を図 7, 8 に示す.

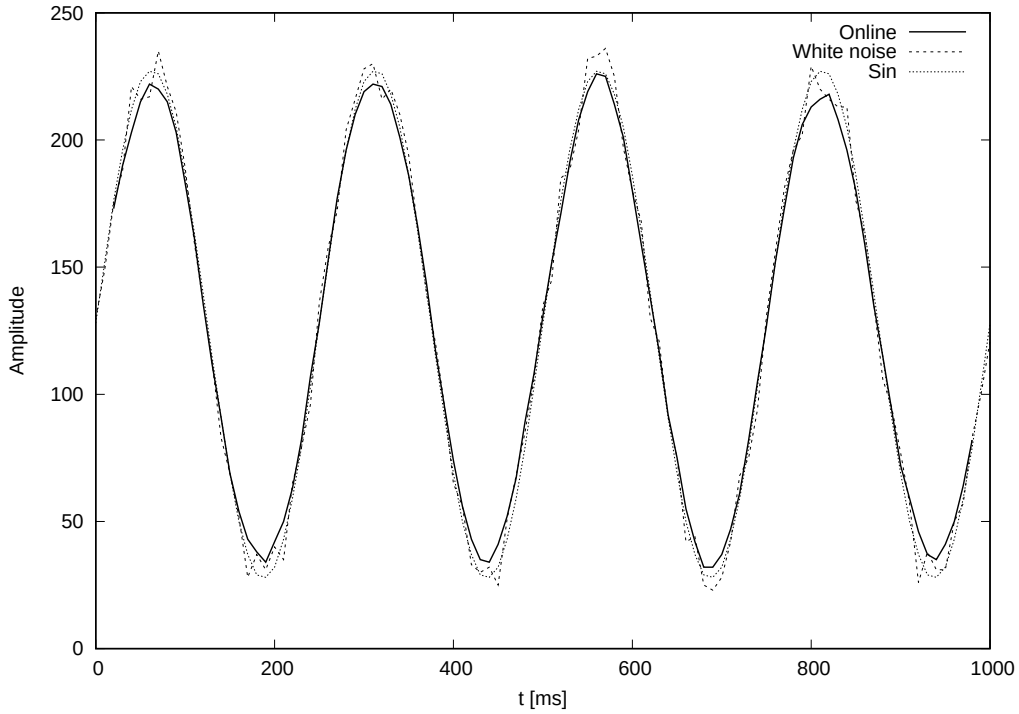

図 7　オンライン型プログラムを利用して雑音を取り除いた波形

図から, どちらの波形も元の波形に近づいていることがわかる. また, 5 点平均を取っているため, データの始点と終点がそれぞれ 2 つずつ欠けている.

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.

■演習 4-3　参考文献 [2] のリスト 5 を完成させよ. 次に, コマンドライン引数として指定された 2 つの CSV ファイルについてファイルについて, 時系列データの差を取るオフライン型プログラム diff2.c を作成せよ.

作成したプログラムを一部抜粋してリスト 11 に示す.

リスト 11　diff2.c

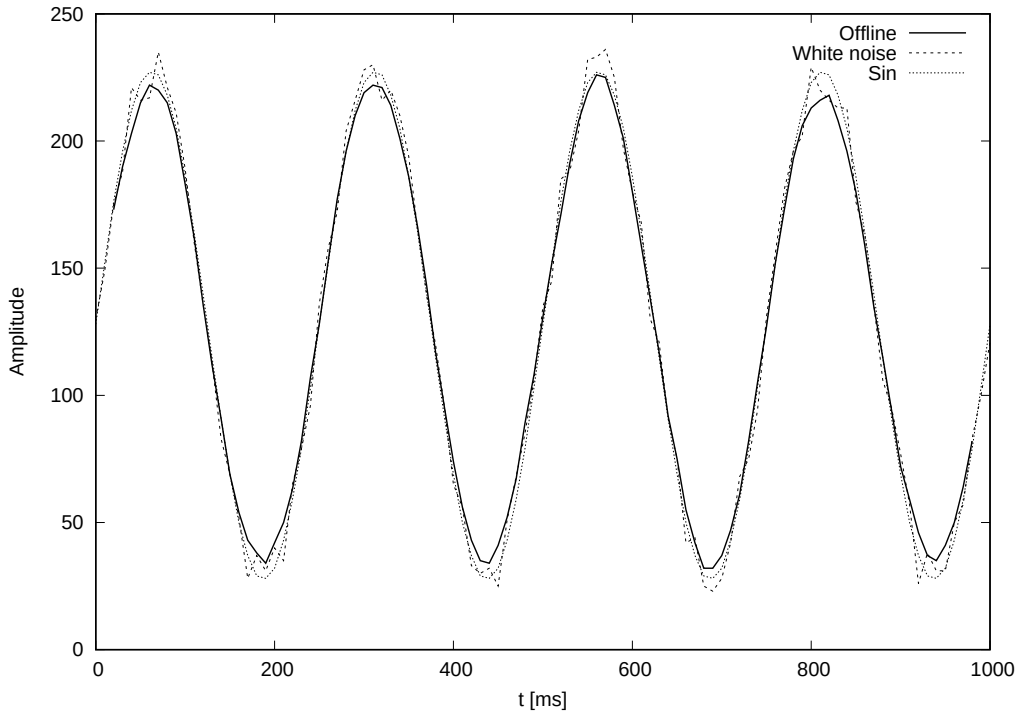

図 8　オフライン型プログラムを利用して雑音を取り除いた波形

```
1  #define BUFSIZE 80
2  #define DATANUM 1000
3
4  int main(int argc, char **argv) {
5      int tm1[DATANUM] = {}, tm2[DATANUM] = {}, amp1[DATANUM], amp2[DATANUM], dif, n, m;
6      char buf[BUFSIZE];
7
8      char *p;
9      FILE *fp1, *fp2;
10
11     for (n = 0; n <= DATANUM; n++) {
12         if (fgets(buf, sizeof(buf), fp1) == NULL) break;
13         if (buf[0] == '#') continue;
14         tm1[n]  = atoi(strtok(buf, ","));
15         amp1[n] = atof(strtok(NULL, "\r\n\0"));
16     }
17
18     for (n = 0; n <= DATANUM; n++) {
19         if (fgets(buf, sizeof(buf), fp2) == NULL) break;
20         if (buf[0] == '#') continue;
21         tm2[n]  = atoi(strtok(buf, ","));
22         amp2[n] = atoi(strtok(NULL, "\r\n\0"));
23     }
24
25     for (n = m = 0; n < DATANUM; n++) {
```

```
26          while (tm2[m] < tm1[n]) m++;
27          while (tm1[n] < tm2[m]) n++;
28
29          if (tm1[n] != tm2[m]) continue;
30
31          dif = amp1[n] - amp2[m];
32          printf("%d,␣%4d\n", tm1[n], dif);
33      }
34  }
```

はじめに 2 つのデータを読み込み, その後に同時刻のデータの差をとるオフライン処理をしている.

このプログラムを使用して振幅 100, 周波数 4 Hz の正弦波とそれに最大振幅 10 の白色雑音を加えた波の差を取ったデータを図 9 に示す.

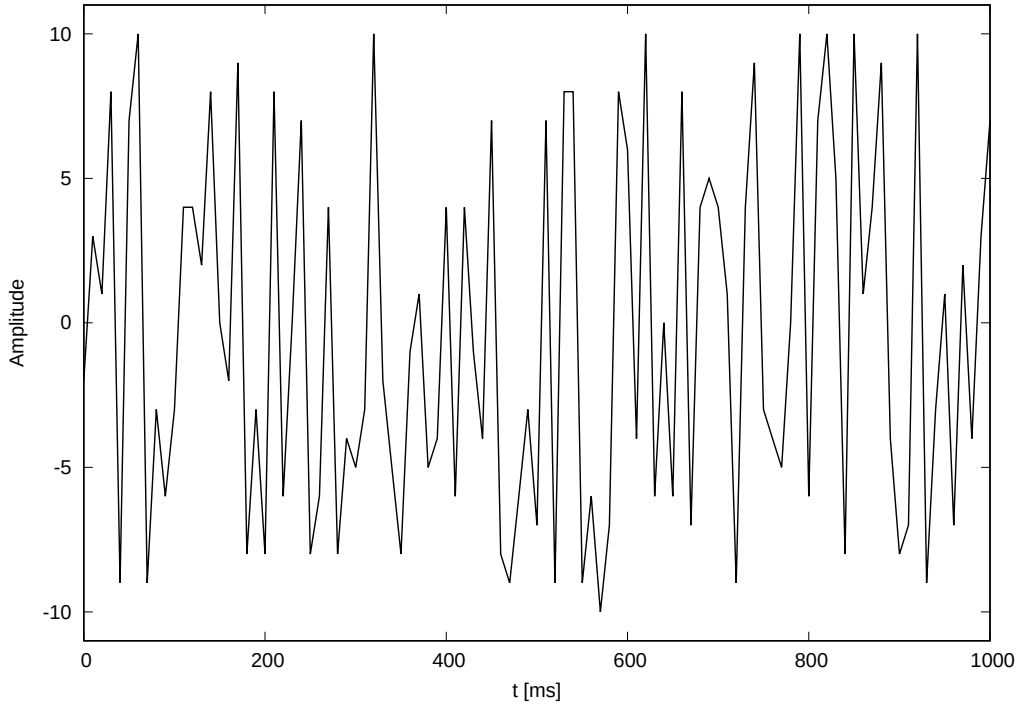

図 9　最大振幅 10 の白色雑音

図より, 正しく最大振幅 10 の白色雑音が取り出せていることがわかる.

■演習 4-4　元信号の CSV ファイルを第 1 引数, 雑音が重畳された CSV ファイルを第 2 引数として与えられたときに, SNR[dB] を求めるプログラム snr.c を作成せよ. このプログラムを使って, 元信号が正弦波の場合について, 3 点移動平均と 5 点移動平均の雑音除去性能を定量的に比較せよ.

作成したプログラムを一部抜粋してリスト 12 に示す.

リスト 12　snr.c

```
1  #define BIAS 0x80
```

```
#define BUFSIZE 80
#define DATANUM 1000

int main(int argc, char **argv) {
    int tm1[DATANUM] = {}, tm2, amp1[DATANUM], amp2, n, sum1 = 0, sum2 = 0;
    char buf[BUFSIZE];

    FILE *fp1, *fp2;

    for (n = 0; n <= DATANUM; n++) {
        if (fgets(buf, sizeof(buf), fp1) == NULL) break;
        if (buf[0] == '#') continue;
        tm1[n]  = atoi(strtok(buf, ","));
        amp1[n] = atof(strtok(NULL, "\r\n\0"));

        sum1 += (amp1[n] - BIAS) * (amp1[n] - BIAS);
    }

    for (n = 0; n <= DATANUM; n++) {
        if (fgets(buf, sizeof(buf), fp2) == NULL) break;
        if (buf[0] == '#') continue;
        tm2  = atoi(strtok(buf, ","));
        amp2 = atoi(strtok(NULL, "\r\n\0"));

        sum2 += (amp2 - amp1[n]) * (amp2 - amp1[n]);
    }

    printf("SNR[dB]:␣%f\n", 10 * log10(sum1 / (double)sum2));
}
```

このプログラムを使って各種正弦波とそれに白色雑音を加えた波の SNR を求めた結果を表 1 に示す.

表 1　正弦波と雑音が加わった正弦波の SNR [dB]

元の正弦波の振幅		25		50		100	
元の正弦波の周波数 [Hz]		4	8	4	8	4	8
白色雑音の最大振幅 [Hz]	5	14.40	14.46	19.97	20.00	26.82	26.92
	10	9.16	9.02	14.92	14.47	20.51	20.38
	20	3.59	3.17	9.24	9.77	14.71	15.05

表から, SNR は元の正弦波の振幅の増加に伴って増加し, 白色雑音の最大振幅の増加に伴って減少することがわかる. また, 周波数の変化は SNR にほとんど影響がないこともわかる.

次に, 白色雑音を加えた正弦波から演習 3 で作成したプログラムを使用して雑音除去を施したデータと, 元の正弦波の SNR を求める. 結果を表 2 に示す.

表 2　正弦波と雑音を除去した正弦波の SNR [dB]

雑音除去の手法		3 点移動平均				5 点移動平均			
元の正弦波の振幅		50		100		50		100	
元の正弦波の周波数 [Hz]		4	8	4	8	4	8	4	8
白色雑音の最大振幅 [Hz]	5	11.53	6.50	12.01	6.42	6.29	1.41	6.43	1.36
	10	10.64	6.27	11.68	6.26	5.89	1.39	6.32	1.29
	20	9.17	6.19	11.24	5.96	5.51	1.41	6.24	1.19

表から, SNR は表 1 と同様に元の正弦波の振幅の増加に伴って増加し, 白色雑音の最大振幅の増加に伴って減少することがわかる. また, 全体的に 5 点移動平均の方が SNR が小さく, 雑音除去性能が高いといえる. さらに, 周波数の増加に伴って SNR は低下しているため, 単純移動平均は高周波の信号に対してより有効であることがわかる.

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 5-3　ステレオ音声・量子化ビット数 16 の WAVE ファイルの波形データを CSV ファイルにダンプするプログラム wav2txt-s16.c を作成せよ.

作成したプログラムを一部抜粋してリスト 13 に示す. なお, 参考文献 [2] のリスト 6 内にある `read_head` の戻り値をサンプリングレートに変更して使用している.

リスト 13　wav2txt-s16.c

```
1  typedef unsigned short uShort;
2  typedef unsigned long uLong;
3
4  void read_data(FILE *fp, uLong smprate, int start, int end) {
5      short amp_l, amp_r;
6      double now = 0;
7      while (fread(&amp_l, 2, 1, fp) && fread(&amp_r, 2, 1, fp)) {
8          if (now >= start && now <= end) printf("%f,␣%d,␣%d\n", now, amp_l, amp_r);
9          now += 1000.0 / smprate;
10     }
11 }
12
13 int main(int argc, char **argv) {
14     uShort ch, qbit;
15     FILE *fp;
16
17     read_data(fp, read_head(fp, &ch, &qbit), atoi(argv[2]), atoi(argv[3]));
```

```
18 }
```

このプログラムを使用して, 参考文献 [1] にある chimes.wav を区間 [200 : 210] ms についてダンプしたものを図 10 に示す.

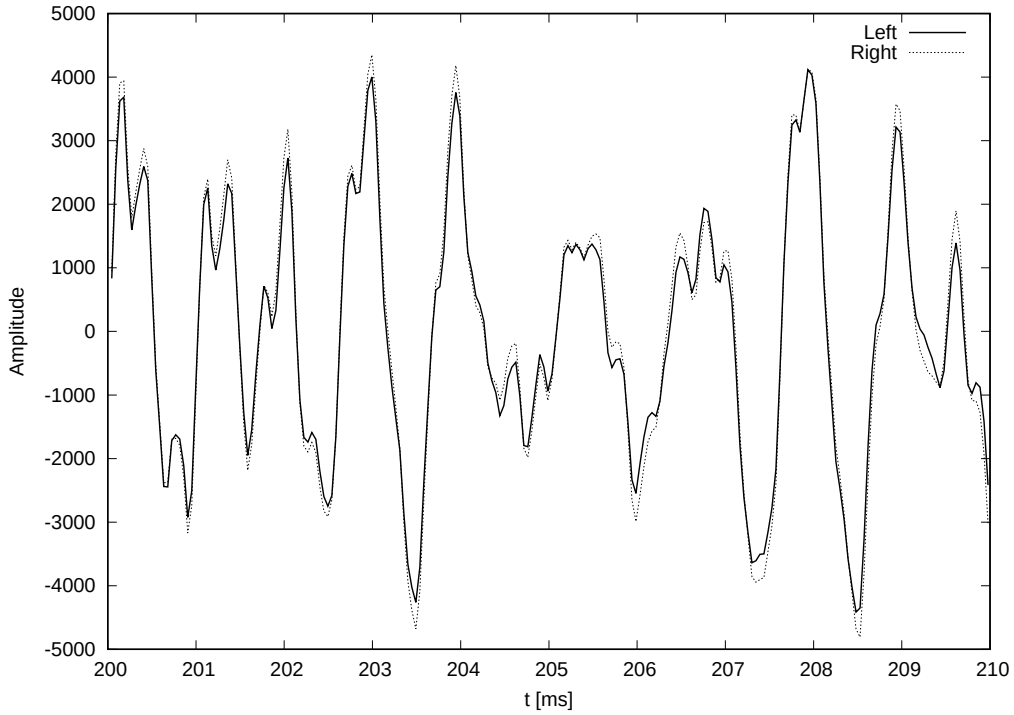

図 10　chimes.wav

図から, ほぼ同波形だが, 左右のデータが別々にサンプリングされていることがわかる.

8　課題

$[0 : 2\pi)$ の範囲を取る変数 θ について, 3 種類の周期関数 $f_i(\theta)$ を考える.

- $f_1(\theta) = \dfrac{2}{\pi} \displaystyle\sum_{k=1}^{M} \frac{\sin k\theta}{k}$
- $f_2(\theta) = \dfrac{4}{\pi} \displaystyle\sum_{k=1}^{M} \frac{\sin\{(2k-1)\theta\}}{2k-1}$
- $f_3(\theta) = \dfrac{8}{\pi^2} \displaystyle\sum_{k=1}^{M} \left(\sin\frac{k\pi}{2}\right) \frac{\sin k\theta}{k^2}$

それぞれの波形を振幅 A, 周波数 f [Hz] の時系列データ $Af_i(t) + A_0$ として生成するプログラム f1.c, f2.c, f3.c を一部抜粋してリスト 14, 15, 16 に示す.

リスト 14　f1.c

```
1 #define BIAS 0x80
2
```

```
int main(int argc, char **argv) {
    int m, n, t;
    double amp, r, vin;
    unsigned char vout;

    for (t = 0; t <= 1000; t += 2) {
        for (vin = BIAS, n = 1; n <= m; n++) {
            r = (t * 2.0 / 1000.0 * 2.0 * PI) * n;
            vin += 2.0 * amp * sin(r) / (double)n / PI;
        }

        printf("%d,␣%4d\n", t, vout);
    }
}
```

リスト 15　f2.c

```
#define BIAS 0x80

int main(int argc, char **argv) {
    int m, n, t;
    double amp, r, vin;
    unsigned char vout;

    for (t = 0; t <= 1000; t += 2) {
        for (vin = BIAS, n = 1; n <= m; n++) {
            r = (t * 2.0 / 1000.0 * 2.0 * PI) * (2 * n - 1);
            vin += 4.0 * amp * sin(r) / (double)(2 * n - 1) / PI;
        }

        printf("%d,␣%4d\n", t, vout);
    }
}
```

リスト 16　f3.c

```
#define BIAS 0x80

int main(int argc, char **argv) {
    int m, n, t;
    double amp, r, vin;
    unsigned char vout;

    for (t = 0; t <= 1000; t += 2) {
        for (vin = BIAS, n = 1; n <= m; n++) {
            r = (t * 2.0 / 1000.0 * 2.0 * PI) * n;
            vin += 8.0 * amp * sin(r) / (double)n / (double)n / PI / PI * sin(n * PI /
    2.0);
```

```
12          }
13
14          printf("%d,␣%4d\n", t, vout);
15      }
16  }
```

これらのプログラムを使って, $A = 100, A_0 = 128, f = 2$ Hz とし, $M = 1, 10, 100$ について波形を出力したものを図 11, 12, 13 に示す.

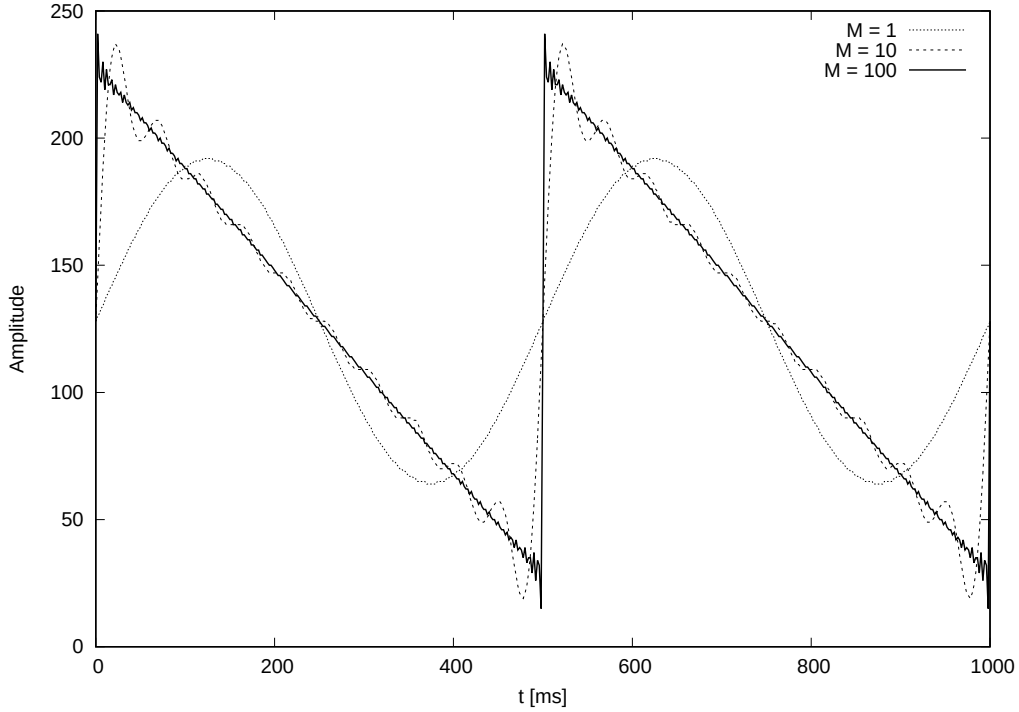

図 11　f_1 の波形

図より, それぞれ M が増加するほどのこぎり波, 矩形波, 三角波に近づいていることがわかる.

次に, $M = 100$ の場合のそれぞれの波形に, 演習 3-1 で実装した add-wn2.c を使用して最大振幅 10 の白色雑音を加える. この波形を演習 3-2, 3-3 で実装した mvave3-1.c, mvave5-1.c を使用して雑音除去したものを図 14, 15, 16 に示す.

また, これらの波形と雑音を加える前のそれぞれの波形の SNR を演習 4-4 で実装した snr.c を使用して求め, 表 3 にまとめる.

表 3　雑音を加え, 除去した波形と雑音を加える前の波形の SNR [dB]

波形	f_1	f_2	f_3
3 点移動平均	17.35	19.34	23.97
5 点移動平均	14.46	15.69	23.22

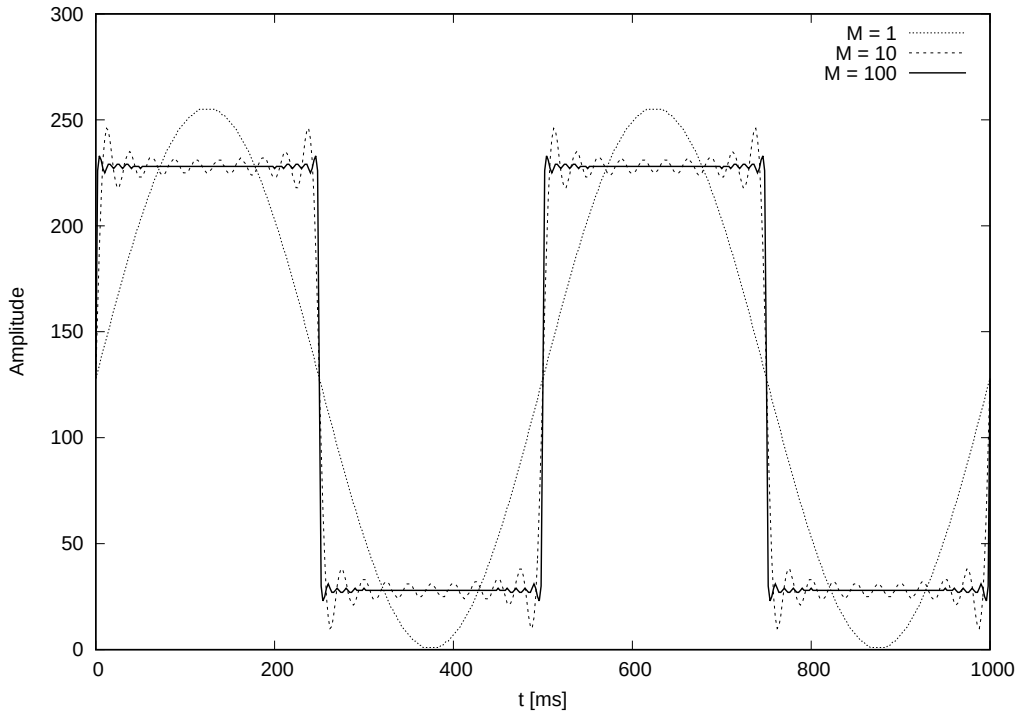

図 12　f_2 の波形

これらの結果から, 演習 4-4 の結果と合わせて, 5 点移動平均の方が雑音除去性能が優れているといえる.

9　発展課題

今回は, 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入力した時に, それをステレオ音声に変換し, さらに一定周期で音が周りを回っているように感じるように信号処理を行う.

元データのファイル, 出力先のファイル, 回転の周波数をコマンドライン引数で入力するようにし, 時間によって角度を増加させ, 左右の出力を調節する.

作成したプログラム revolute.c を一部抜粋してリスト 17 に示す.

リスト 17　revolute.c

```
1  #define BIAS 0x80
2
3  int copy_header(FILE *fp1, FILE *fp2) {
4      char str[H_LEN];
5      int
6          riff_size, fmt_size, format_id, channel_n, sampling_rate, byte_per_sec, block_size,
       q_bit, data_size;
7
8      fread(str, 1, H_LEN, fp1);
9      fread(&riff_size, 4, 1, fp1);
10     fwrite("RIFF",          sizeof(char),  4, fp2);
```

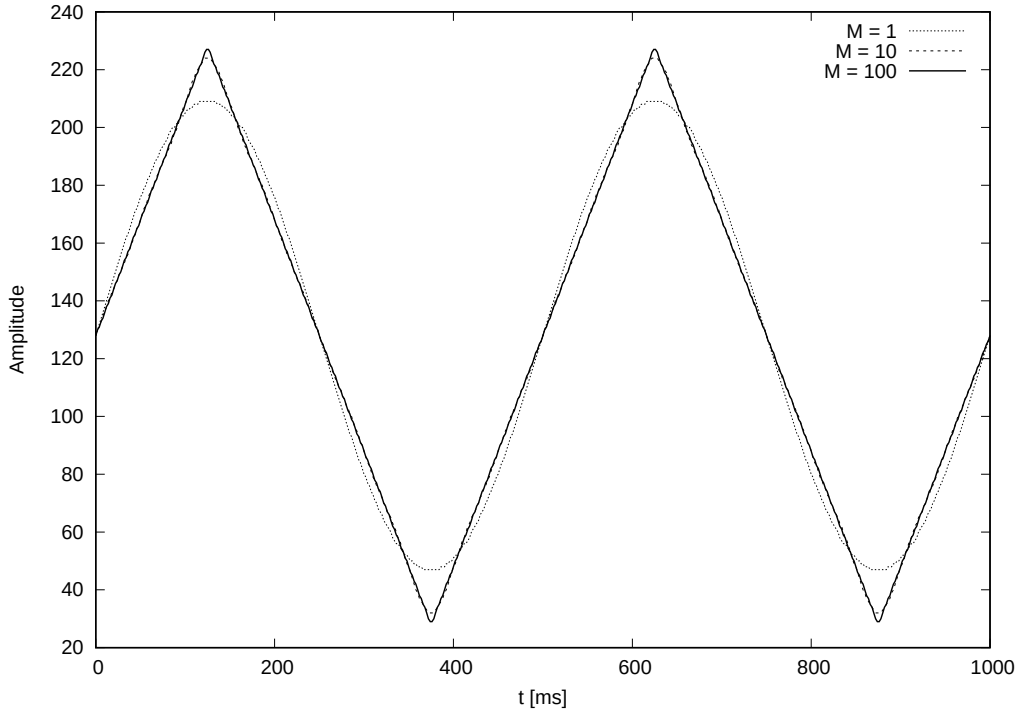

図 13　f_3 の波形

```
11      fwrite(&riff_size,     sizeof(int),   1, fp2);
12
13      fread(str, 1, H_LEN, fp1);
14      fread(str, 1, H_LEN, fp1);
15      fread(&fmt_size, 4, 1, fp1);
16      fwrite("WAVE",          sizeof(char),  4, fp2);
17      fwrite("fmt␣",          sizeof(char),  4, fp2);
18      fwrite(&fmt_size,      sizeof(int),   1, fp2);
19
20
21      fread(&format_id, 2, 1, fp1);
22      fwrite(&format_id,     sizeof(short), 1, fp2);
23
24      fread(&channel_n, 2, 1, fp1);
25      channel_n = 2;
26      fwrite(&channel_n,     sizeof(short), 1, fp2);
27
28      fread(&sampling_rate, 4, 1, fp1);
29      fwrite(&sampling_rate, sizeof(int),   1, fp2);
30
31      fread(&byte_per_sec, 4, 1, fp1);
32      fwrite(&byte_per_sec,  sizeof(int),   1, fp2);
33
34      fread(&block_size, 2, 1, fp1);
35      block_size = 2;
```

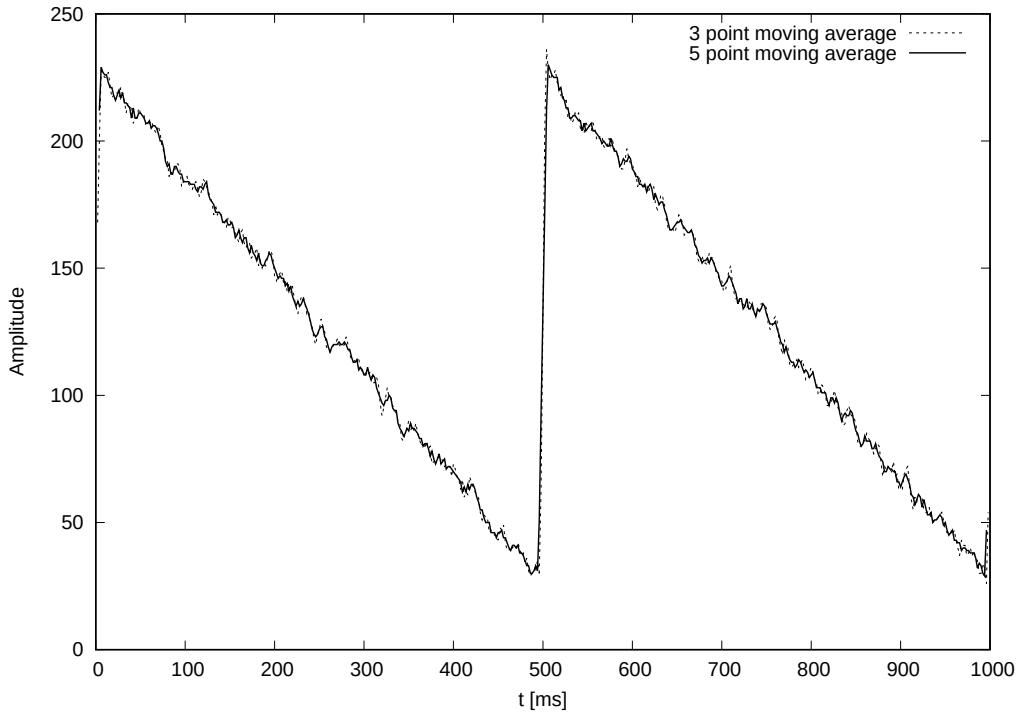

図 14　雑音を加え, 除去した f_1 の波形

```
36      fwrite(&block_size,    sizeof(short), 1, fp2);
37
38      fread(&q_bit, 2, 1, fp1);
39      fwrite(&q_bit,          sizeof(short), 1, fp2);
40
41      fwrite("data",          sizeof(char),  4, fp2);
42
43      fread(&data_size, 4, 1, fp1);
44      data_size *= 2;
45      fwrite(&data_size, sizeof(int),   1, fp2);
46
47      return sampling_rate;
48  }
49
50  void revolute(FILE *fp1, FILE *fp2, int sampling_rate, double frq) {
51      int amp, ampl, ampr;
52      double theta = 0;
53      while ((amp = fgetc(fp1)) != EOF) {
54          ampl = (amp - BIAS) * (cos(theta / 180.0 * PI) / 2.0 + 0.5) + BIAS;
55          ampr = (amp - BIAS) * (0.5 - cos(theta / 180.0 * PI) / 2.0) + BIAS;
56
57          fwrite(&ampl, 1, 1, fp2);
58          fwrite(&ampr, 1, 1, fp2);
59          theta += 360.0 * frq / sampling_rate;
60      }
```

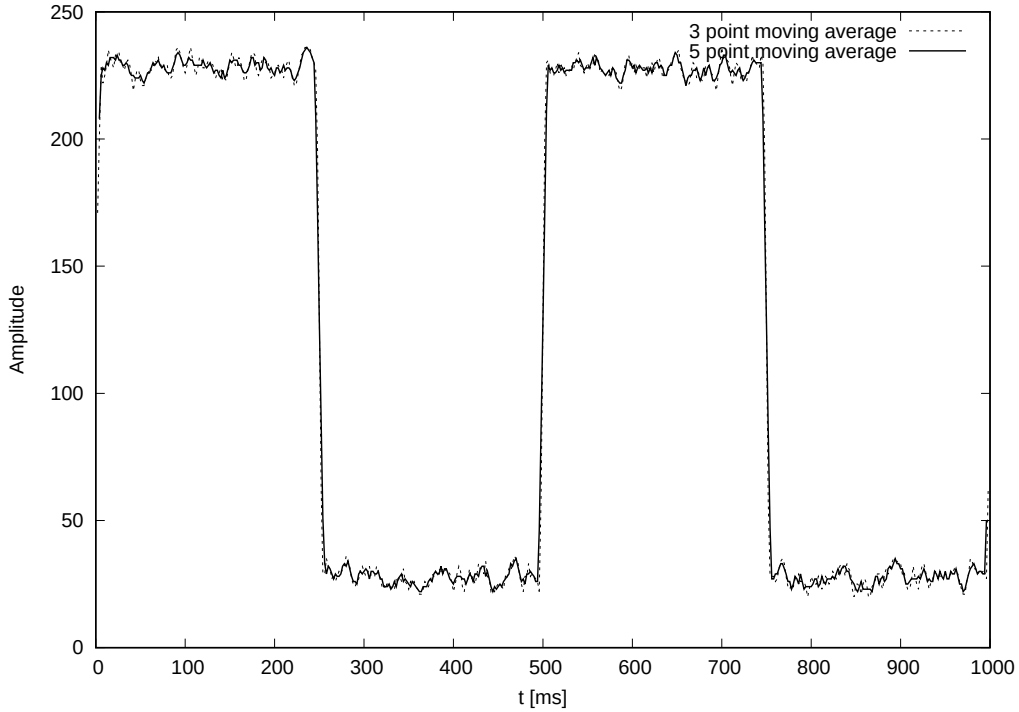

図 15　雑音を加え, 除去した f_2 の波形

```
61 }
62
63 int main(int argc, char **argv) {
64     double frq;
65     FILE *fp1, *fp2;
66
67     revolute(fp1, fp2, copy_header(fp1, fp2), frq);
68 }
```

このプログラムを使用して, 参考文献 [1] にある timetone.wav に信号処理を施し, 左右に分けて波形を出力したものを図 17, 18 に示す. また, 参考までに timetone.wav の波形を図 19 に示す.

図より, 左右から交互に音が出力できていることがわかる. 実際に音を聞いてみると, 音が回って聞こえる.

元ファイルのヘッダを詳しく解析することで, モノラル音声・量子化ビット数 8 以外のデータにも対応させられることが予想される. また機会があれば挑戦したい.

10　感想および改善案

6 月末に, 「まだ最終レポートまでは時間がたくさんあるな」などと思っていたが, 気づけば 8 月になっていた. 幸い演習は終わらせてあったため, あまり慌てることなく実験内容を復習しながらレポート作成に取り組めた. 今回の実験を通して, gnuplot やコマンドライン引数など, 今後に役立てられるようなツールを使いこなせるようになり, とても良い経験ができたと思う.

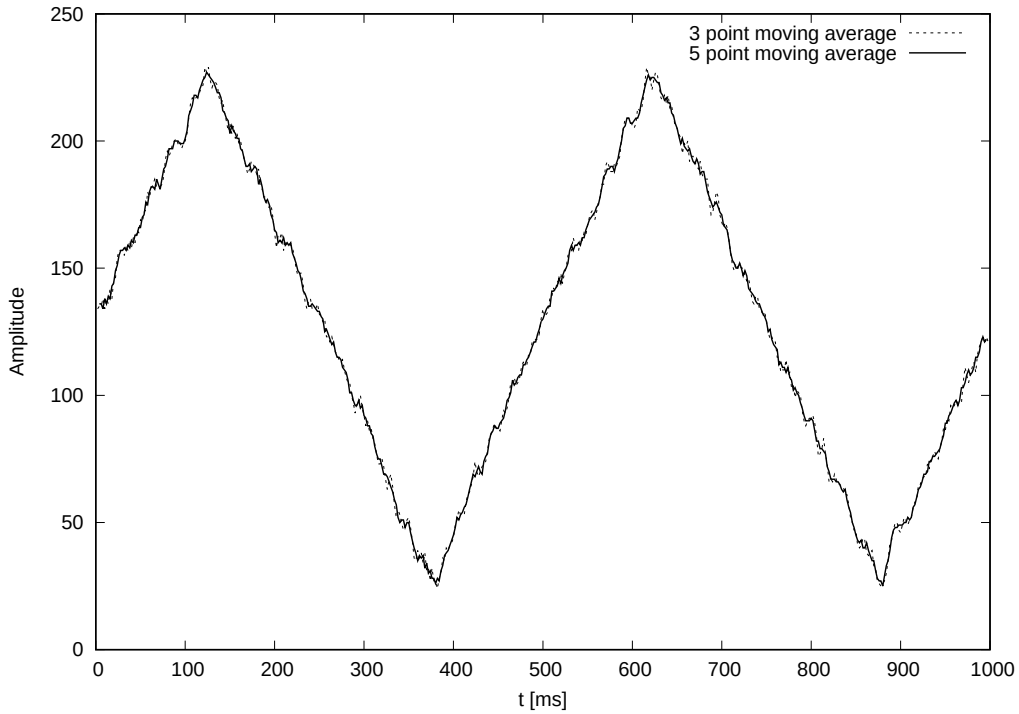

図 16　雑音を加え, 除去した f_3 の波形

課題について, のこぎり波などのフーリエ級数を導出からする課題であればより楽しめたと思った.

参考文献

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

[2] 高橋章, 信号処理プログラミング, 令和 2 年 · 前期

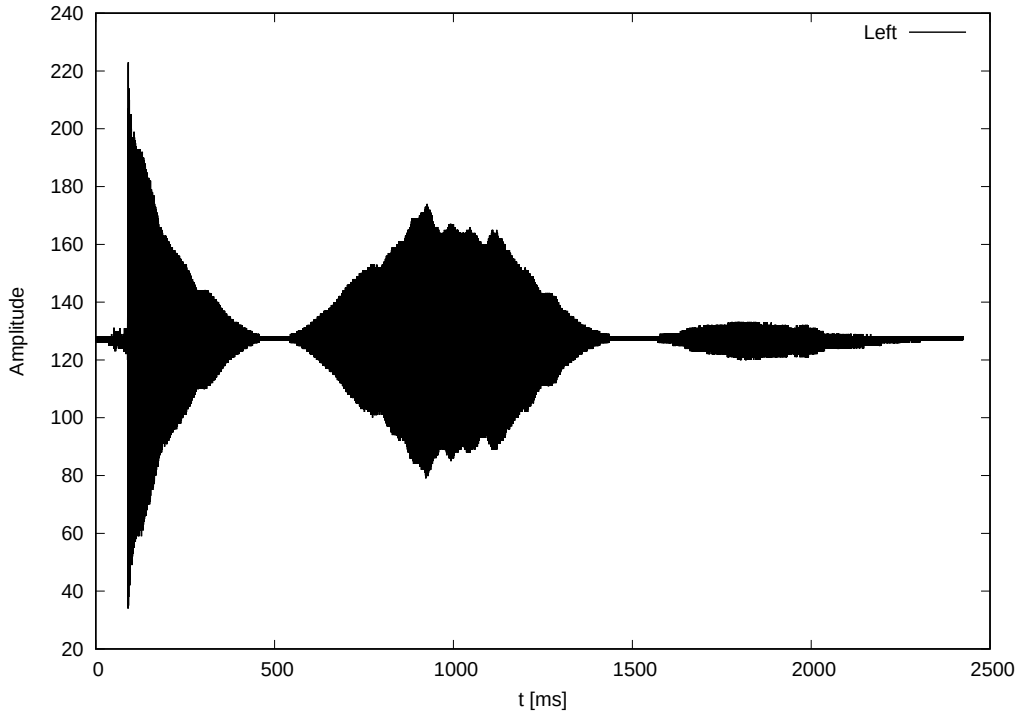

図 17　処理を施したデータの左の波形

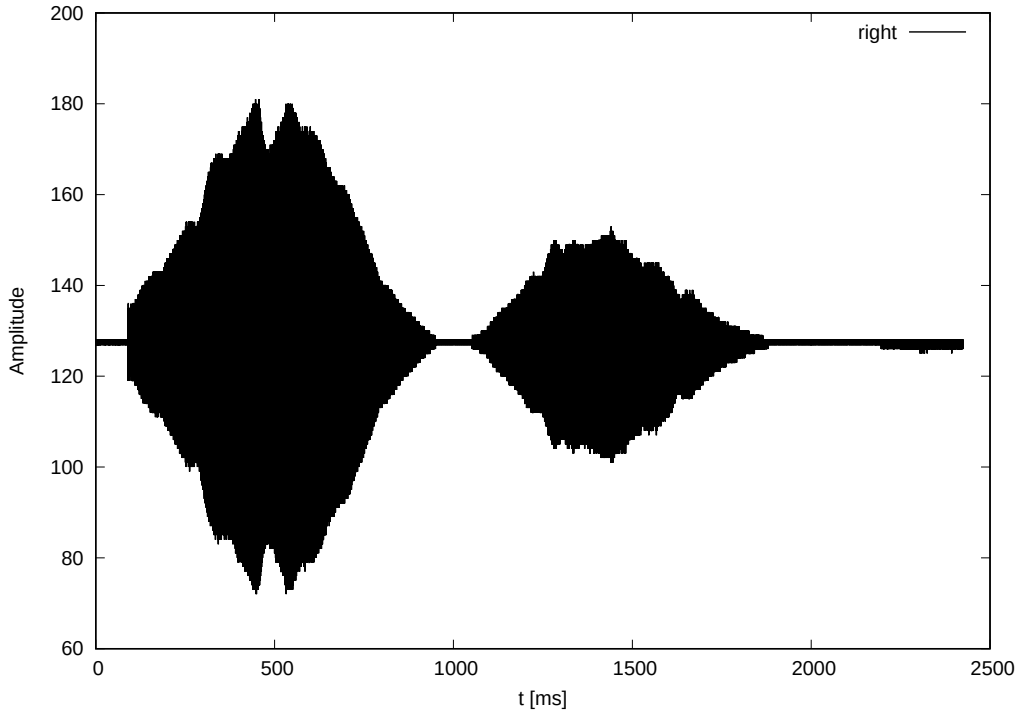

図 18　処理を施したデータの右の波形

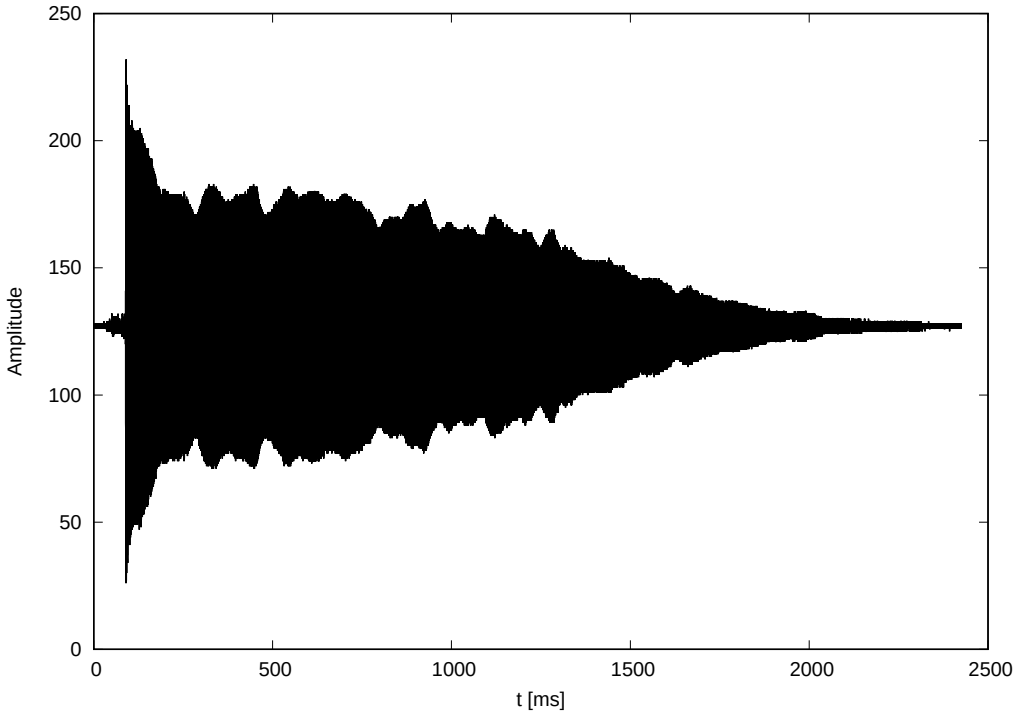

図 19　timetone.wav