

受 付	閱

電子情報工学実験報告書

No. 4 実験課題 音声データの信号処理

実験場所 情報処理 実験室(実習室)

実験年月日 2026 年 4 月 20 日 天候 _____ 気温 _____ °C

2026 年 4 月 27 日 湿度 _____ % 気圧 _____ hPa

電子情報工学科 5 学年

G 班 学籍番号 22330 報告者氏名 野口 煌陽

共同実験者氏名(班長) _____

担当教職員 川本

提出期限 2026 年 5 月 11 日

提出年月日 2026 年 4 月 29 日

実験レポート自己点検・不備事項指摘票

実験の目的

- 意味も理解せず指導書の丸写しとなっていないか

原理

- 意味も理解せず指導書の丸写しとなっていないか
- 必要な事項を省き過ぎていないか
- 図を省略していないか

実験方法

- 意味も理解せず指導書の丸写しとなっていないか
- 必要な事項を省き過ぎていないか
- 図を省略していないか

使用器具

- フォーマットに沿って表形式としているか
- 規格には、その器具にとって必要なことが書かれているか
- 使用した器具を特定できる「その他」の項が書かれているか

実験結果

- 本文中で引用して（「図○に××特性を示す」など）、表・図にまとめたか
- 実験結果について、わかったこと・気づいたことを述べたか

考察

- 考察は1/4ページ以上記載したか
- 実験したことについて、自分の言葉で論じたか
- 手引書にある考察事項については、文献等を調べ、実験したことと関連付けて書けたか

表の書き方

- 表の上に通し番号（表1など）をつけ、内容の分かる具体的な題名を付けたか
- 表の最上部に量名・記号・単位を付けたか
- 複数ページに渡る表の場合、具体的な題名のあとに（つづき）や（1/3）→（2/3）→（3/3）などを付けたか
- 複数ページに渡る表の場合、ページ毎に表の最上部に量名・記号・単位を付けたか（最初の表だけでは不備）
- 定規を用いて書かれたか
- 用紙の位置を考慮して書かれたか

図の描き方

- 図の下に通し番号（表とは別、図1など）をつけ、内容の分かる具体的な題名を付けたか
- 回路図などは定規を用いて書かれたか
- 用紙の位置を考慮して書かれたか
- グラフでは縦軸・横軸のタイトルと単位を適切な位置に書いたか
- グラフの縦軸のタイトルは縦軸に沿った向きに書いたか
- グラフでデータ点は大きく描き、自在定規などを用いて滑らかな線で結んだか
- グラフで複数曲線がある場合は、パラメータを付けたか
- 横長のグラフの綴じる位置を間違えていないか

プログラム・リストの掲載方法

- リストの下に通し番号（リスト1, リスト2, …）をつけ、リストの内容が分かる具体的な題名（例：バブルソート・昇順版）を付けたか
- リストを印刷する場合、行番号付きで印刷したか
- 複数ページに渡るリストの場合、ページ毎に題名を付け、題名のあとに（つづき）や（1/3）→（2/3）→（3/3）などを付けたか

レポート全体

- 表紙の記入を行ったか（プレレポート時、レポート時）
- レポートにはページ番号を付けたか
- （4年生以上）概要を書いたか
- 図等をスキャンして印刷していないか（認められた場合を除く）
- 「レポートの書き方」が配布されている場合、それにしたがって書いたか

概要

本実験は音声データの分析を対象に、1次元の時系列信号に対する処理の実装を通じて、信号処理の基礎を理解することを目的としている。離散フーリエ変換 (DFT) とは、標本化されたデジタル信号に対する周波数解析を行うための方法である。本実験では、DFT および DFT を高速化した高速フーリエ変換 (FFT) を C 言語で実装し、それらの実行効率や計算結果の比較を行った。さらに、窓関数を適用するプログラムを実装し、振幅スペクトルにどのような変化が現れるかについての確認と考察を行った。分析する音声データには、標本化周波数 16 kHz、量子化ビット数 16 bit、チャンネル数 1ch のものを使用した。各実験課題の実験結果を gnuplot を用いてグラフとして描画し、それをもとに実験結果を考察した。振幅スペクトルの計算と可視化の実験では、どちらのフーリエ変換で周波数解析を行っても結果は変わらないが、DFTの方がFFTに比べ約7千倍もの実行時間の差があることが確認された。窓関数の導入の実験では、窓関数により振幅スペクトルが全体的に小さくなることが確認された。各母音の音声の周波数解析の実験では、発声内容によって振幅スペクトルのピーク値が異なることが確認された。これらの実験を通じて、離散フーリエ変換による音声データの周波数解析の動作原理と実装方法を理解することができた。また、考察事項として窓関数適用による振幅値の減少、ダウンサンプリングの副作用、窓関数と振幅スペクトルの関係についての考察を行った。以下にその要点を述べる：考察 1) 窓関数の値域が $0 \leq w[n] \leq 1$ であるため、振幅スペクトルは小さくなる。本実験では、信号の平均パワーが窓関数適用により約 0.27 倍になることが確認された。考察 2) 前処理をせず、単純に観測信号を間引くだけでは高音域の情報の損失や折り返し雑音が発生してしまう。折り返し雑音対策のために、前処理としてアンチエイリアシングフィルタを適用する。考察 3) 窓関数を特徴づける要素として、メインローブの幅とサイドローブのレベルがある。これらの違いにより、様々な窓関数が存在するため、状況に応じて選択する必要がある。

1 目的

音声データの分析を対象に、1次元の時系列信号に対する処理の実装を通じて、信号処理の基礎を理解する。

2 原理

2.1 アナログ信号とデジタル信号

音声は、空気の振動などにより伝播する。音声を信号として捉えると、振幅が時間的に変化することで情報が伝わる。音声の情報は、マイクロフォンにより電気信号として観測できる。このように観測される信号は、時間変化する1次元の情報であり、音波形として時間成分と振幅成分として表現できる。電気信号の時点では、時間成分及び振幅成分ともに連続値であり、「アナログ信号」という。

音声信号をコンピュータに取り込み、処理するためには、時間成分及び振幅成分ともに離散値に変換した「デジタル信号」にする必要がある。アナログ信号をデジタル信号に変換することをA/D変換という。逆に、デジタル信号をアナログ信号に変換することをD/A変換という。コンピュータ上の音声信号をスピーカーやヘッドフォンで再生する際にはD/A変換が必要となる。

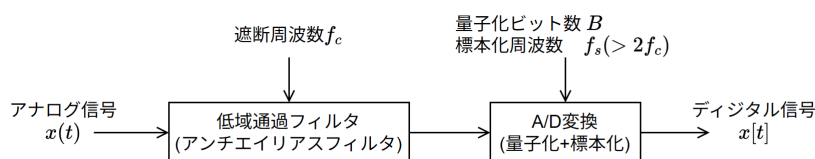


図 1: A/D 変換の流れ

入力のアナログ信号をコンピュータに取り込むデジタル信号に変換するための基本的な処理の流れを図 1 に示す。A/D 変換は 2 種の離散化により実現される。時間方向に離散化することを標本化（サンプリング）といい、振幅方向に離散化することを量子化という。

標本化は、アナログ信号から一定の時間間隔毎に信号値を得る操作である。信号値を得る時間間隔 T を標本化間隔あるいは標本化周期、1 秒あたりの標本化回数 ($= \frac{1}{T}$) を標本化周波数という。標本化定理*1により、標本化対象の信号は、標本化周波数の半分未満に帯域制限されている必要がある。このため、標本化前にはアンチエイリアスフィルタと呼ばれる低域通過フィルタを通し、不要な高域成分を除去する。もし、信号の最大周波数の 2 倍未満である低い周波数で標本化を行なうと、見かけ上、低い周波数が存在しているかのように見える現象が生じる。このような現象をエイリアシング（折り返し歪）という。

量子化は、ある範囲の振幅値を代表値に置き換えることで離散的な値に変換する。このため、量子化では代表値に置き換える際に誤差を生じることとなる。この誤差を量子化誤差という。また、量子化した振幅値を何ビットの整数値で表現するかにより、振幅情報をなめらかさが変化する。この情報を量子化ビット数という。量子化ビット数が大きいほど、多くの代表値で表現することでなめらかな振幅が表現でき、量子化誤差が小さくなる。本実験では、波形符号化方式として線形パルス符号化 (linear pulse code modulation: linear PCM)

*1 信号の最大周波数が f_{max} [Hz] のとき、信号を $2f_{max}$ [Hz] より高い標本化周波数で標本化すると元の信号を復元することができる。

を採用する。これは、アナログ信号を一様なステップで量子化するもので、通常 A/D 変換として行われているものと同じである。

2.2 離散フーリエ変換

標本化定理を満たすように標本化周期 T で標本化されたデジタル信号 $x[n]$ について考える。デジタル信号に対する周波数解析を行うには、離散化されたフーリエ変換、すなわち離散フーリエ変換 (DFT; discrete Fourier transform) が用いられる。

周期 N 点の周期信号と見なし、 N 点のデジタル信号に対する離散フーリエ変換は式 (1) のように定義される。

$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j \frac{2\pi}{N} kn} = \sum_{n=0}^{N-1} x[n] W_N^{kn} \quad (1)$$

また、逆離散フーリエ変換は式 (2) のように定義される。

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] e^{j \frac{2\pi}{N} kn} = \sum_{k=0}^{N-1} X[k] W_N^{-kn} \quad (2)$$

ただし、 $W_N = e^{-j \frac{2\pi}{N}}$ 、 n は時間に関する整数変数、 k は周波数に関する整数変数である。 n は切り出した N 点のデジタル信号の先頭を 0 としたときの時間 $t = nT$ と対応する。 k は各周波数 $\omega = \frac{2\pi}{T} \frac{k}{N}$ に対応する。なお、 N 点の DFT の時間計算量は $O(N^2)$ となる。

2.3 高速フーリエ変換

DFT では分析点数 N の 2 乗オーダーの演算量を必要とする。このため、分析点数が多い場合や繰り返しフーリエ変換を行う場合には、その演算量が問題となることも多い。そのため、多くの場合で高速フーリエ変換 (FFT; fast Fourier transform) という方法が用いられる。ここでは分析点数 N が 2 のべき乗の形で与えられる条件下で、時間分割法に基づく FFT を紹介する。

時間分割法では、分析対象の信号を「奇数番目のデータ」と「偶数番目のデータ」に分けていくことで、問題を小分けし、計算量を削減する。ここでは分析点数 $N = 8$ を例として考える。

まずは信号 $x[n]$ を、

$$\begin{cases} \text{偶数番目の信号} & x_0[n] = x[2n] & (n = 0, 1, 2, 3) \\ \text{奇数番目の信号} & x_1[n] = x[2n + 1] & (n = 0, 1, 2, 3) \end{cases}$$

に分割する。このとき、信号 $x[n]$ の DFT は、 $W_N = e^{-j\frac{2\pi}{N}}$ とおくと、式 (3) のように表現できる。

$$\begin{aligned}
 X[k] &= \sum_{n=0}^7 x[n] W_8^{kn} \\
 &= \sum_{n:\text{even}} x[n] W_8^{kn} + \sum_{n:\text{odd}} x[n] W_8^{kn} \\
 &= \sum_{n=0}^3 x_0[n] W_8^{k(2n)} + \sum_{n=0}^3 x_1[n] W_8^{k(2n+1)} \\
 &= \underbrace{\sum_{n=0}^3 x_0[n] W_4^{kn}}_{x_0[n] \text{ の DFT}} + W_8^k \underbrace{\sum_{n=0}^3 x_1[n] W_4^{kn}}_{x_1[n] \text{ の DFT}} \tag{3}
 \end{aligned}$$

つまり、 $x_0[n]$ の DFT を $X_0[k]$ 、 $x_1[n]$ の DFT を $X_1[k]$ とおくと、

$$X[k] = X_0[k] + W_8^k X_1[k] \quad (k = 0, 1, 2, 3)$$

となる。ここで k は 0,1,2,3 であり、8 点の DFT の前半 4 点分の計算のみである。同様に後半 4 点分の計算 $X[k+4]$ について考える。式 (3) において、 k に $k+4$ を代入すると、次のようになる。

$$X[k+4] = \sum_{n=0}^3 x_0[n] W_4^{(k+4)n} + W_8^{k+4} \sum_{n=0}^3 x_1[n] W_4^{(k+4)n} \tag{4}$$

ここで、 $W_N = e^{-j\frac{2\pi}{N}}$ を考慮すると、

$$W_4^{(k+4)n} = W_4^{kn} \tag{5}$$

$$W_8^{(k+4)} = -W_8^k \tag{6}$$

となるため、これらを式 (4) に代入すると、

$$\begin{aligned}
 X[k+4] &= \sum_{n=0}^3 x_0[n] W_4^{kn} - W_8^k \sum_{n=0}^3 x_1[n] W_4^{kn} \\
 &= X_0[k] - W_8^k X_1[k] \tag{7}
 \end{aligned}$$

となる。以上より、

$$\begin{cases}
 X[k] &= X_0[k] + W_8^k X_1[k] & (n = 0, 1, 2, 3) \\
 X[k+4] &= X_0[k] - W_8^k X_1[k] & (n = 0, 1, 2, 3)
 \end{cases}$$

を得る。

4 点の DFT についても同様に 2 点の DFT に分割することができる。このように最終的に 2 点の DFT まで分割し、計算結果を統合することで FFT が実現される。なお、 N 点間の時間分割法による FFT の時間計算量は $O(N \log N)$ となる。

2.4 信号の周波数分析

信号 $x[n]$ の離散フーリエ変換で得られるスペクトル $X[k]$ は複素数であり、図 2 のように実数成分 $Re(X[k])$ と虚数成分 $Im(X[k])$ から、振幅成分である振幅スペクトル $|X[k]|$ と、位相成分である位相スペクトル $\angle X[k]$

に分解できる.

$$\begin{cases} \text{振幅スペクトル } |X[k]| &= \sqrt{\{Re(X[k])\}^2 + \{Im(X[k])\}^2} \\ \text{位相スペクトル } \angle X[k] &= \tan^{-1} \left(\frac{Im(X[k])}{Re(X[k])} \right) \end{cases}$$

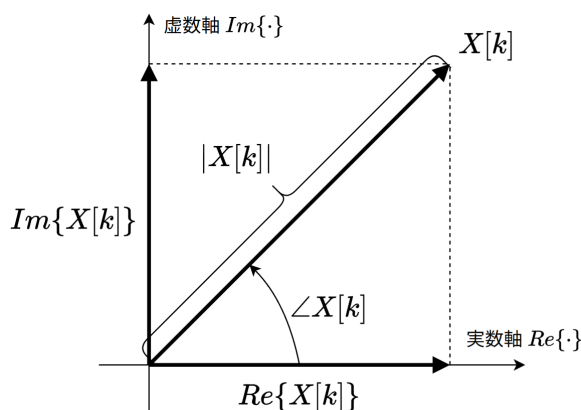


図 2: 振幅スペクトルと位相スペクトル

2.5 窓関数

DFT や FFT では、分析点数 N を周期とする信号と見なして分析される。そのため、切り出し区間の両端点で異なる値を持つ場合、信号における不連続点が生じてしまい、元の信号に含まれない周波数成分が現れることがある。この現象をスペクトル漏れという。このため、切り出し区間の両端点付近を 0 に近くなるような関数を掛け合わせることで、両端点での波形の急激な変化を抑える方法が用いられる。このように、信号を指定区間に切り出すための関数を窓関数という。元の信号を $x[n]$ 、窓関数を $w[n]$ とすると、窓関数により切り出された信号 $y[n]$ は次のように表される。

$$y[n] = x[n] \cdot w[n]$$

窓関数としては、用途に応じて様々なものが存在する。例えば、図 3 に示すハニング窓は、以下の式で表現される。

$$w[n] = 0.5 - 0.5 \cos \left(\frac{2\pi n}{N-1} \right) \quad (8)$$

2.6 音声信号の周波数分析

音声信号は、信号に含まれる周波数特性を時間変化させることにより情報を伝達している。このため、音声の周波数分析を行う場合は、時間変化する信号であっても「切り出した短時間の中では周波数特性が変化しない」と仮定し、切り出した短時間の信号を周期信号と見なして DFT や FFT を適用することで、周波数分析を行う。このような短時間の信号に対する分析を、一定周期で切り出す区間をずらしながら行う。このとき、短時間の信号に切り出す処理をフレーム化処理といい、切り出した信号をフレーム、フレームに含まれる信号

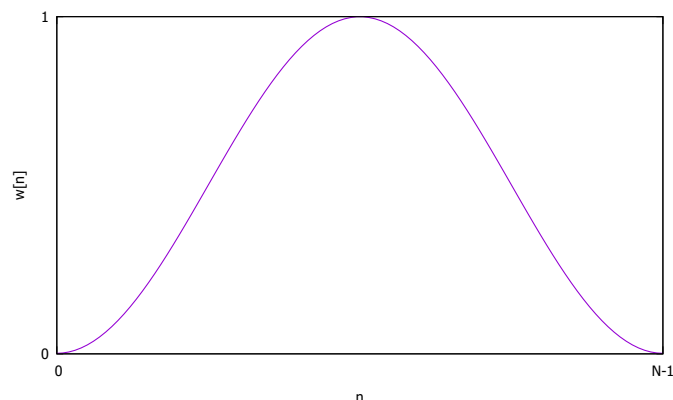


図 3: ハニング窓の例

点数をフレーム長，フレームをずらす時間間隔をフレーム周期という，フレーム化処理のイメージ図を図 4 に示す．フレーム化処理により得られた各フレームに対し，必要に応じて窓関数をかけ，FFT を適用した結果

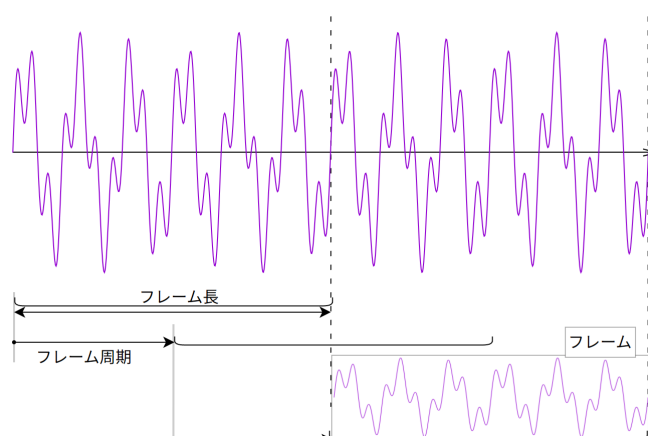


図 4: フレーム化処理のイメージ図

から振幅スペクトルを求めることで，フレームを切り出した時刻における振幅スペクトルを推定することができる．

フレーム毎に得られる振幅スペクトルを時系列と見なし，時間・周波数・信号成分の 3 次元の情報として表現したものをスペクトログラムという．一般的なスペクトログラムの可視化方法としては，時間を横軸，周波数を縦軸，振幅を輝度や色で表現した 2 次元画像を用いることが多い．

振幅スペクトル $|X[k]|$ やパワースペクトル $|X[k]|^2$ を扱うとき，対数で表現することも多い．音声においても信号の比として相対量を可視化することが多い．この時用いられるのがデシベル (dB) 単位での表現であり，振幅スペクトルの場合は，以下のように表される．

$$20 \log_{10} \frac{|X[k]|}{|X_b|} \text{ [dB]}$$

ここで X_b は振幅スペクトルの相対量を表現するための基準値であり，可視化などにおいては「最大振幅値」

を基準値とすることが多い。つまり、最大振幅値を 0[dB] とし、可視化のために設定する下限値*2までの幅でスペクトルあるいはスペクトログラムを可視化する。例えば信号の振幅値の量子化ビット数が 16bit(65536 段階)である場合、信号のダイナミックレンジ*3は、

$$20 \log_{10} 65536 \simeq 96[\text{dB}]$$

となる。なお符号付き 2 バイト整数で振幅値を扱う場合、最大振幅値は 32767 であり、実質的な信号のダイナミックレンジ 90[dB] 程度となる。そのため、振幅スペクトルの最大値を 0[dB] としたとき、0 ~ -90[dB] 程度の範囲で十分な情報を可視化できる。実際には、観測した信号の最大振幅などによって調整する必要がある。

3 実験課題

以下の課題を実現する C 言語のプログラム、および gnuplot スクリプトを作成するとともに、所望の図を得るまでの手順を示せ。なお、信号処理のための各プログラムは C 言語を用いて実装し、C 言語の仕様に付随して提供される標準的な関数のみを使用して作成すること。また、原理のアルゴリズムを対応付けてプログラムの概要を説明すること。

課題 1 バイナリファイルの読み込み

ヘッダのない音声データ sample01.sw は以下の条件でデジタル化された音声波形のデータである：

標準化周波数：16 kHz
 量子化ビット数：16 bit
 チャンネル数：1 ch(モノラル)
 符号化方式：Linear PCM

1 サンプル分のデータは 2 bytes で表現され、C 言語における、signed short 型の整数変数 1 つに対応する。この信号の先頭 30 ミリ秒 (msec) 分のデータを読み込み、波形データとして gnuplot を用いて可視化せよ。なお、縦軸は振幅値 (単位なし)、横軸は時間 [msec] として可視化すること。また、読み込む際、signed short 型変数で期待される最大値において、最大振幅の絶対値が 1 となるように振幅を正規化した実数値として扱うこと (つまり、サンプル値 32767 が振幅値 1 となるように正規化)。

課題 2 DFT および FFT による振幅スペクトルの計算と可視化

音声データ sample01.sw について、先頭 512 点を読み込み、フーリエ変換することで振幅スペクトルを算出し、0 ~ 8kHz の範囲で振幅スペクトルを可視化せよ。なお、縦軸は振幅スペクトル値 (単位なし)、横軸は周波数 [Hz] として可視化すること。また、同じデータについて縦軸振幅スペクトル値 [dB]、横軸は周波数 [Hz] として可視化する場合についても合わせて示すこと。フーリエ変換については離散フーリエ変換 (DFT; Discrete Fourier Transform) を実現する関数 dft および高速フーリエ変換 (FFT; Fast Fourier Transform) を実現する関数 fft を実装し、その動作原理と使い方、計算結果の比較、計算速度の比較をレポートに示すこと。

*2 対数において $\log 0 \rightarrow -\infty$ となることもあり、可視化の目的に沿ってユーザが設定したある一定の幅で可視化する。

*3 ダイナミックレンジは、識別可能な信号の最小値と最大値の比を表す情報のことである。

課題 3 窓関数の導入

音声データ sample01.sw について、先頭 512 点をそのまま読み込んだ後、ハニング窓 (hanning window) を適用する。窓長 N は 512 となる。観測信号 $x[n]$ と窓関数適用後の信号 $y[n]$ をそれぞれフーリエ変換し、振幅スペクトルを可視化することで比較せよ。なお、縦軸は振幅スペクトル (単位なし)、横軸は周波数 [Hz] とし、横軸の範囲 0 ~ 8kHz にて可視化すること。また、同じデータについて縦軸は振幅スペクトル [dB]、横軸は周波数 [Hz] として可視化する場合についても合わせて示すこと。

課題 4 各自の音声の周波数分析

レポート作成者本人の音声を対象として振幅スペクトルを計算・可視化せよ。なお、適切な部分を複数箇所切り出し、それぞれの振幅スペクトルを比較することで、発生内容による振幅スペクトルの違いを分析すること。

4 使用器具

表 1 に使用器具を示す。

表 1: 使用器具

器具名	規格	製造会社	その他
パソコン	GALLERIA XF	Dospara	
OS	Windows 11 25H2 26200.8246	Microsoft	
C コンパイラ	gcc 8.1.0	GNU	
グラフ描画ツール	gnuplot 6.0.4	gnuplot	

5 実験結果

5.1 バイナリファイルの読み込み

まず、先頭 30 ミリ秒が何バイトまでなのかを計算する。sample01.sw は標本化周波数が 16 kHz であるから、1 サンプルあたりの時間は、

$$\frac{1}{16 \text{ kHz}} = 0.0625 \text{ m sec}$$

となる。先頭 30 ミリ秒のデータが欲しいから、そのサンプル数は、

$$\frac{30 \text{ m sec}}{0.0625 \text{ m sec}} = 480$$

となる。したがって、480 サンプル分のデータを使えば良い。

図 5 に gnuplot を用いて指定のデータを可視化したグラフを示す。6 ミリ秒から 24 ミリ秒にかけて規則的な波形が見られる。最大振幅の絶対値が 1 となるように描画されており、正しく出力できたことがわかる。

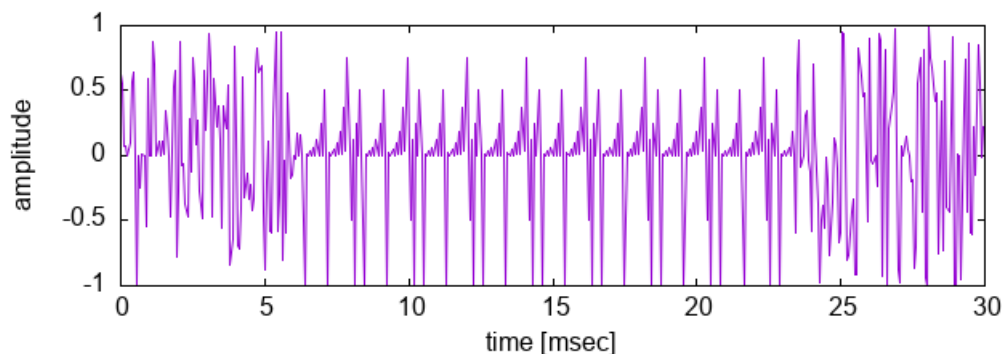


図 5: sample01.sw の先頭 30msec のデータの波形

可視化の際には、リスト 1 のスクリプトを使用した²⁾。1, 2 行目で出力ファイルの拡張子と名前を設定している。4 行目では標準化周波数を変数として宣言している。6, 7 行目では、横軸と縦軸それぞれに軸名を設定している。9, 10 行目では、横軸と縦軸それぞれの描画範囲を設定している。12 行目でグラフの縦横比を設定している。今回は縦軸の長さが横軸の長さの 0.3 倍になるようにした。14~16 行目でデータの出力を行っている。binary format="%int16"により、データを 16bit 符号付き整数として読み込むように指定する。every :::479 は、データを 0 個目から 479 個目までの計 480 サンプルを読み込む設定である。using (\$0 * 1000.0 / fs) では、横軸に関して、\$0 (データ番号) に周波数で割ることで単位を秒にし、1000 を掛けてミリ秒に変換している。(\$1/32767.0) では、縦軸に関して、\$1 (データ) を 16bit 整数の最大値 32767 で割ることで、値を-1.0 から 1.0 の範囲に正規化している。with lines でデータを点ではなく線をつないで描画するように、notitle でラインの名前を表示しないようにしている。

5.2 DFT および FFT による振幅スペクトルの計算と可視化²⁾

DFT と FFT のプログラムを実装したプログラムをリスト 2 に示す。フーリエ変換における複素数を扱うため、実部 r と虚部 i をメンバに持つ構造体 Complex を定義した (10~13 行目)。また、C 言語標準では複素数演算が直接定義されていないため、以下の基本演算を関数として実装した (197~233 行目)。

- addComplex, subComplex, timeComplex: 複素数同士の加減乗算 (199~208 行目)
- divComplexByReal: 複素数の実数による除算 (210~212 行目)
- cexptheta: オイラーの公式 $e^{i\theta} = \cos\theta + i\sin\theta$ (197 行目)
- powComplex: 回転子の累乗計算用 (214~233 行目, DFT で使用)

関数 dft (173~188 行目) では、離散フーリエ変換の定義式 (1) を直接計算している。計算の効率化のため、genTwiddleTable 関数 (160~166 行目) により回転子 W_N の値をあらかじめ計算してテーブルに保持する手法を採用した。しかし、アルゴリズムの性質上、シグマループと信号長による反復 (180~187 行目) という二重ループの計算が必要であり、計算量は $O(N^2)$ となる。

関数 fft (120~159 行目) では、時間間引き型の Cooley-Tukey アルゴリズムを実装した³⁾。本実装はビット反転とバタフライ演算の 2 つのステップで構成される。

FFT をインプレース (同一配列内) で計算するため、入力データの添字をビット反転順に並べ替える必要

がある。関数 `bit_reversal` (94~119 行目) は、信号長 N が 2 のべき乗であることを前提とし、ビットシフト演算を用いて高速に並べ替えを実行する。

並べ替えられたデータに対し、バタフライ演算を行う。外側のループ (130 行目~) でステージ数 ($\log_2 N$ 回) を管理し、内側のループ (136 行目~) で各ブロックの複素積と加減算 (140~146 行目) を行う。回転子 W の更新には、各ステップで基本となる回転係数 w_m を順次乗算していく手法 (148 行目) をとり、三角関数の呼び出し回数を最小限に抑えている。このアルゴリズムにより、計算量は $O(N \log N)$ に削減される。

DFT と FFT の計算効率を定量的に評価するため、Windows 環境の提供する高分解能タイマー (`QueryPerformanceCounter`) を用いて処理時間を計測した⁴⁾。音声データ `sample01.sw` の先頭 512 点をバイナリモードで読み込んで配列の初期値に代入した後、DFT (78~82 行目) および FFT (83~87 行目) の各アルゴリズムの実行直前と直後のカウント差から秒単位の経過時間を算出した。

このプログラムを使用するには、以下のコマンドを実行する。

```
gcc -fexec-charset=CP932 FFT.c -o FFT ; .\FFT.exe sample01.sw
```

計測時間の出力結果を図 6 に示す。FFT に比べ DFT は約 7 千倍遅くなった。

```
PS H:\5jExperiment\4Voice> gcc -fexec-charset=CP932 FFT.c -o FFT ; .\FFT.exe sample01.sw
DFT Elapsed time: 0.522504 seconds
FFT Elapsed time: 0.000072 seconds
```

図 6: 512 点データに対する DFT および FFT の実行速度

計算結果を確認するために、リスト 3 のプログラムを 88 行目に挿入する。 `dft` 関数の動作確認の際は配列 `c` を、 `fft` 関数の動作確認の際は配列 `x` を用いて振幅 `amp` を求める。同時に動作確認をすることはできないため、片方の実行時はもう片方をコメントアウトしなければならない。また、82 行目、87 行目の実行時間を出力する `printf` 関数もコメントアウトしなければならない。そして、以下のコマンドを実行して離散フーリエ変換済みのデータファイル `output.txt` を得る。

```
gcc -fexec-charset=CP932 FFT.c -o FFT ; .\FFT sample01.sw > output.txt
```

このとき、 `output.txt` には離散フーリエ変換されたデータが N 行 1 列に並ぶ。

得られたデータを `gnuplot` を用いて可視化する。可視化したグラフを図 7 に示す。可視化の際には、単位なしのグラフはリスト 4 のスクリプト、デシベルのグラフはリスト 5 のスクリプトを使用した。最大振幅値を 0[dB] にするために、 `stats` コマンドを使いデータの最大値を取得して、正規化してからデシベルにしている。図を見れば、DFT と FFT のどちらで離散フーリエ変換を行っても、結果は同じになっていることがわかる。

5.3 窓関数の導入

離散フーリエ変換を行う前の入力信号配列に窓関数をかければ適用できる。リスト 6 に、配列に窓関数を適用する関数 `applyWindow` を示す。配列の全要素に対し、式 (8) を計算してかける。この関数をリスト 2 の 77 行目で呼び出し実行する。得られたデータを `gnuplot` で可視化した結果を図 8 に示す。可視化の際には、単位なしのグラフはリスト 4 のスクリプト、デシベルのグラフはリスト 5 のスクリプトを使用した。最大振幅値を 0[dB] にするために、データを正規化してからデシベルにしている。

図 8a と図 8c を見ると、窓関数を適用することで、振幅スペクトルが全体的に小さくなり、ピークとされているところが際立っていることがわかる。しかし、図 8b と図 8d を見るとどちらも際立った周波数の振幅ス

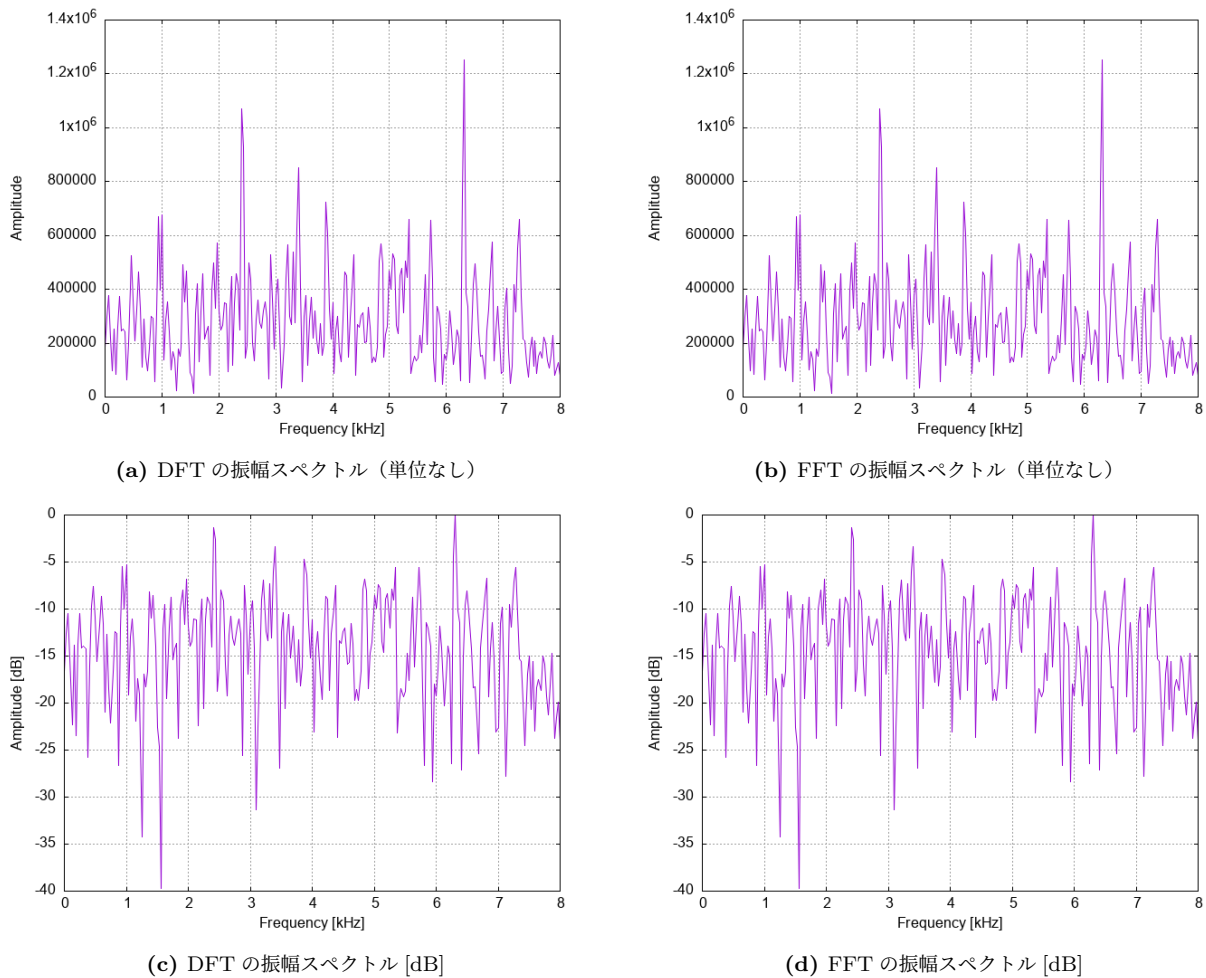


図 7: sample01.sw の振幅スペクトル

ペクトルはなく、ノイズのようなグラフとなっていることがわかる。また、窓関数を適用したグラフは、そのままグラフに比べ急激な変化が少なくなっている。

5.4 各自の音声の周波数分析

レポート作成者本人の音声として、「あ」、「い」、「う」、「え」、「お」の5つの音声データを用意した。それぞれ1つの音声データにまとめられているため、各音声の開始時刻を確認し、それに対応する場所から512点を読み込み、窓関数を適用してから離散フーリエ変換を行う。リスト7に、本節の実験を行うためのmain関数を示す。これをリスト2のmain関数と置き換えて実行する。このプログラムを使用するには、以下コマンドを実行する。

```
gcc -fexec-charset=CP932 FFT.c -o FFT ; .\FFT
```

音声データを使用するために、ファイル名をj22330.swとした音声データファイルを同ディレクトリに配置する必要がある。7行目の変数start_secに処理したい音声の開始時刻[sec]を設定する。このとき、この時刻から512点分のデータを読み込むため、その終了時刻まで音声が続くように開始時刻を選ぶ必要がある。使用

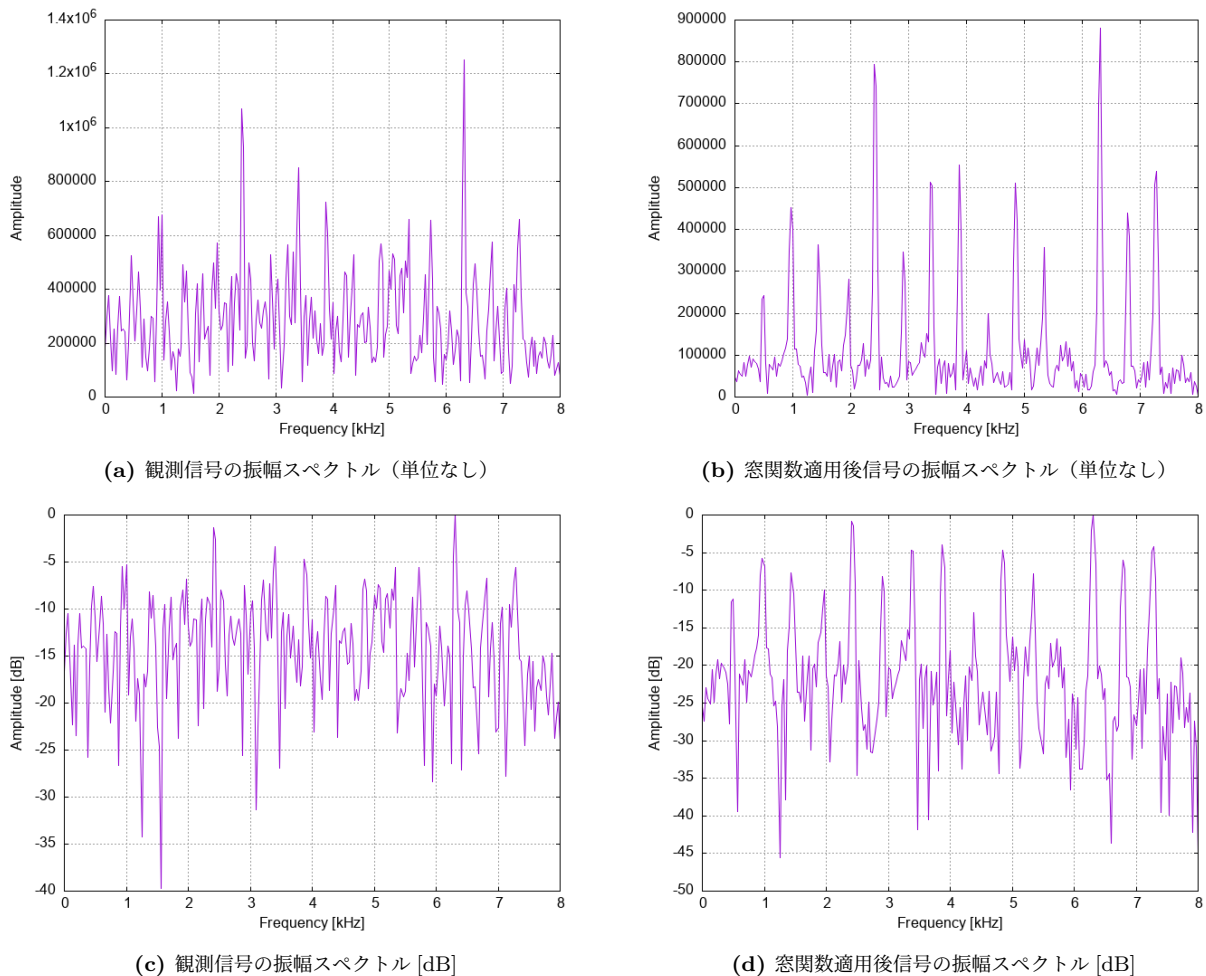


図 8: 窓関数適用で生じる振幅スペクトルの違い

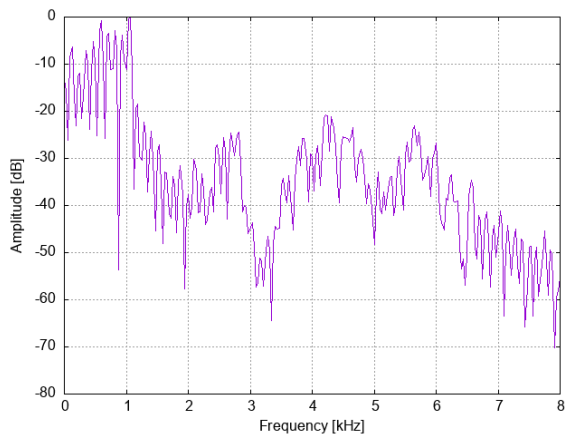
する音声データの標本化周波数は 16 kHz であるから、512 点分の経過時間は、

$$\frac{1}{16 \text{ kHz}} \cdot 512 = 0.032 \text{ sec}$$

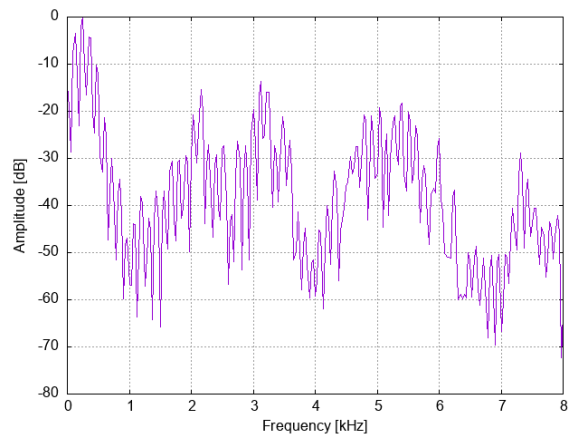
である。10 行目の変数 `bytes_per_sample` は、1 サンプルあたりのバイト数である。使用する音声データの量子化ビット数は 16 bit であるから、2 とした。11 行目で、開始時刻に対応するデータが何バイト目に存在するか求め、変数 `start_byte` に初期化する。20 行目にある通り、使用する音声データファイルのファイル名を `j22330.sw` で指定している。32 行目で、`fseek` 関数を用いて、ファイルポインタを求めた変数 `start_byte` の位置までファイルポインタを移動させる。ここを基準にして音声データを読み込む。

プログラムを実行し、得られたデータを `gnuplot` で可視化した結果を図 9 に示す。可視化の際には、リスト 5 のスクリプトを使用した。最大振幅値を 0 [dB] にするために、データを正規化してからデシベルにしている。各振幅スペクトルを見ると、次のことがわかる。

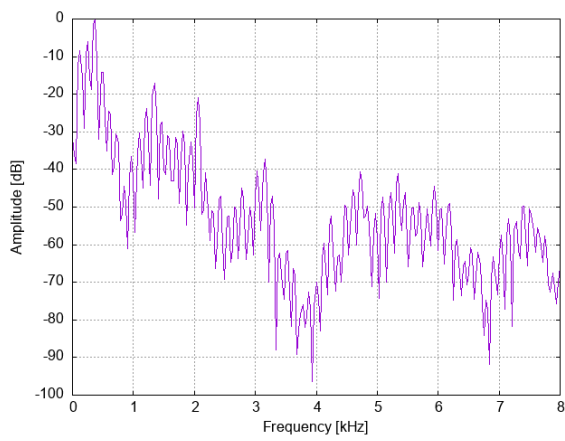
- 「あ」：1 kHz 付近にピークがあり、高域になるにつれて減衰している。
- 「い」：300 Hz 付近と、2,3,5 kHz 付近の高い位置にピークがある。



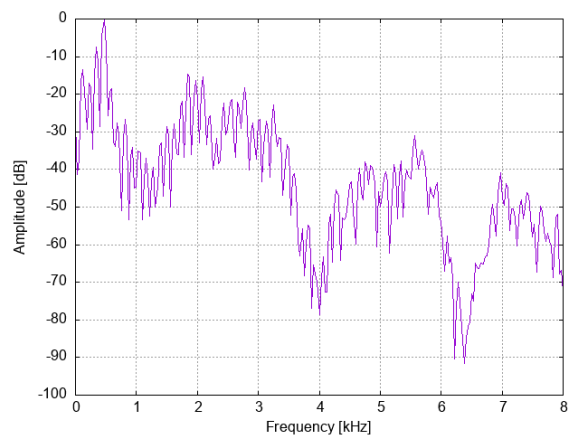
(a) あ



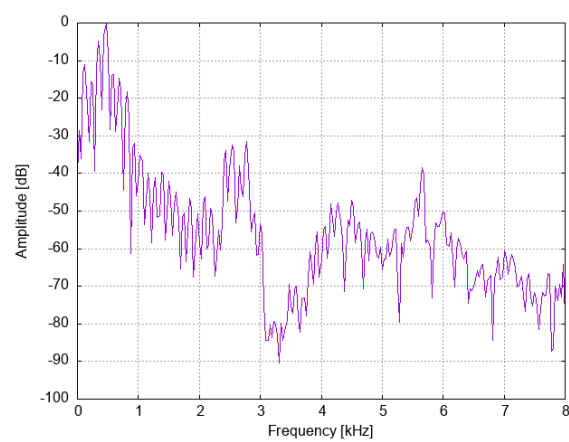
(b) い



(c) う



(d) え



(e) お

図 9: 各音声データの振幅スペクトル [dB]

- 「う」：全体的に低い周波数 (2kHz) 以下に集中している。
- 「え」：500 Hz 付近にピークがあり，2~3kHz に山がある。
- 「お」：500 Hz と 2.5 kHz 付近にピークがあり，それ以降は落ち込んでいる。

6 考察

6.1 窓関数適用による振幅値の減少

5.3 節で確認した通り，窓関数を適用した音声データの振幅スペクトルは，そのままの信号の振幅スペクトルに比べ全体的に振幅値が小さくなっていることがわかった。これは窓関数適用によるスペクトル漏れの改善とトレードオフの関係である。窓関数は式 (8) からわかるように，基本的に 1 以下の値である。この影響で信号の大部分が減衰させられるため，振幅スペクトルが減少する。

信号のパワーは振幅の 2 乗で求められる⁵⁾。音声データの窓関数適用前後で各要素のパワーを足し合わせ，サンプル数で割ることで平均を求め検証する。検証のために，リスト 8 のプログラムを作成した。58~67 行目が平均パワーを求める関数 `calcAmpPower` である。また，関数 `applyWindow` の引数を複素数型配列ではなく `short` 型配列に変更した。このプログラムを以下のコマンドで実行する。

```
gcc -fexec-charset=CP932 power.c -o power ; .\power sample01.sw
```

実行した結果を図 10 に示す。窓関数を適用すると，信号の平均パワーが約 0.27 倍になることがわかった。

```
PS H:\5jExperiment\4Voice> gcc -fexec-charset=CP932 power.c -o power ; .\power sample01.sw
BEFORE : 224097506.908203
AFTER  : 60686398.851563
DIFF   : 163411108.056641
RATIO  : 0.270804
```

図 10: 窓関数適用による信号の平均パワーの変化

長さ N の配列信号 $x[n]$ の信号の平均パワー P_x は以下の式で計算できる。

$$P_x = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2$$

次に，この信号に窓関数 $w[n]$ を適用した信号 $y[n]$ は，

$$y[n] = x[n] \cdot w[n]$$

となる。窓関数適用後の信号パワー P_y は，

$$P_y = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]w[n]|^2 = \frac{1}{N} \sum_{n=0}^{N-1} |x[n]|^2 |w[n]|^2$$

ここで，ハンニング窓関数の定義式 (8) より， $w[n]$ は

$$w[n] = 0.5 - 0.5 \cos\left(\frac{2\pi n}{N-1}\right)$$

である。この関数の値域に関して， $-1 \leq \cos(\theta) \leq 1$ であり，

$$\cos(\theta) = 1 \text{ のとき} : w[n] = 0.5 - 0.5 \cdot 1 = 0$$

$$\cos(\theta) = -1 \text{ のとき} : w[n] = 0.5 - 0.5 \cdot -1 = 1$$

となるから、すべての n に対して

$$0 \leq w[n] \leq 1$$

が成り立つ。これより、 $0 \leq |w[n]|^2 \leq 1$ であるから、以下の関係が成り立つ。

$$|x[n]|^2 |w[n]|^2 \leq |x[n]|^2$$

これを全要素について総和を取ると、

$$\sum_{n=0}^{N-1} |x[n]|^2 |w[n]|^2 \leq \sum_{n=0}^{N-1} |x[n]|^2$$

両辺を N で割れば、

$$P_y \leq P_x$$

となる。

6.2 ダウンサンプリングの副作用

より低い標本化周波数に再標本化することをダウンサンプリングという⁶⁾。16 kHz で標本化された信号を、8 kHz にダウンサンプリングする方法として「何も前処理をせずに、単純に観測信号を 1/2 に間引く (2 回に 1 回サンプルを破棄する)」がある。この方法により、高周波成分がスキップされてしまうことによる高音域情報の損失や、信号に標本化周波数の半分より高い周波数成分が含まれているときに起こる折り返し雑音⁷⁾といった副作用が生じてしまう。折り返し雑音が生じると、高周波成分が低周波成分として混入してしまい、本来の音声とは異なるノイズが発生してしまう。これは標本化定理により説明できる。例えば、信号に 6 kHz の正弦波が含まれているとする。このとき、16 kHz で標本化されていれば、 $6 \text{ kHz} < 8 \text{ kHz} = \frac{16 \text{ kHz}}{2}$ であり、標本化定理を満たすから、正しく音を再現することができる。しかし、ダウンサンプリングを行い、8 kHz の信号として再生すると、2 kHz の音として観測される。これが折り返し雑音である。

信号の標本化を数式で表すと、時間 t は $\frac{n}{f_s}$ となるから、元の信号 $x[n]$ は、

$$x[n] = \cos(2\pi ft) = \cos\left(2\pi \cdot 6000 \cdot \frac{n}{16000}\right) = \cos\left(\frac{3\pi}{4}n\right)$$

と表される。1/2 に間引く操作は n を $2m$ と置くことで表せるから、ダウンサンプリング後の信号 $y[n]$ は、

$$y[m] = x[2m] = \cos\left(\frac{3\pi}{4} \cdot 2m\right) = \cos\left(\frac{3\pi}{2}m\right)$$

となる。ここで、三角関数の性質 $\cos(\theta + 2\pi n) = \cos(\theta)$ と $\cos(-\theta) = \cos(\theta)$ を利用すれば、

$$\cos\left(\frac{3\pi}{2}m\right) = \cos\left(2\pi m - \frac{\pi}{2}m\right) = \cos\left(-\frac{\pi}{2}m\right) = \cos\left(\frac{\pi}{2}m\right)$$

と変形できる。この結果を新しい標本化周波数 $f'_s = 8000 \text{ Hz}$ で表せば、

$$\cos\left(2\pi \cdot f_{new} \cdot \frac{n}{8000}\right) = \cos\left(\frac{\pi}{2}m\right)$$

より, $f_{new} = 2\text{kHz}$ と求められる.

これらのダウンサンプリングによる副作用を低減させるには, 前処理としてローパスフィルタを適用し, 標準化周波数の $1/2$ 以上の成分を除去する必要がある. このとき使われるローパスフィルタを, アンチエイリアシングフィルタという⁸⁾.

6.3 窓関数と振幅スペクトルの関係⁹⁾

窓関数の形状は, その後の周波数分析における振幅スペクトルの概形に影響する. 本実験では, 窓関数としてハンニング窓を扱った. 他にも有名な窓関数を以下に示す.

- 矩形窓: 窓関数を掛けない状態である.
- ハミング窓: 両端が完全に 0 にならない. 高い周波数分解能が求められる場合に使う.

$$w(x) = 0.54 - 0.46 \cos(2\pi x)$$

- ブラックマン窓: ハミング窓よりも裾野の広がり小さい. 振幅の小さい信号を大きい信号の近くで測定したい場合に使う.

$$w(x) = 0.42 - 0.5 \cos(2\pi x) + 0.08 \cos(4\pi x)$$

窓関数は, 時間領域で信号との積により適用する. 時間領域の積は周波数領域の畳み込みに対応するから, 振幅スペクトルは元の信号のスペクトルと窓関数のスペクトルの畳み込みで得られる. 信号のスペクトルにピークがあれば, 窓関数のスペクトルにより畳み込まれ, そのピークが横に広がったようなグラフになる.

窓関数のスペクトルは, スペクトルの中心にある最も大きな山であるメインローブの幅と, その左右に現れる小さな山であるサイドローブのレベルから特徴づけられる. メインローブの幅は周波数分解能に関わる. 幅が広いほど近接する周波数成分がつかなくなってしまい, 区別できなくなる. サイドローブのレベルはスペクトル漏れに関わる. サイドローブが高ければ, 強い信号の近くにある微小な信号が埋もれて観測できなくなる.

矩形窓は, メインローブが鋭く, サイドローブが高いスペクトルである. つまり, 周波数分解能は高いが, スペクトル漏れが多くなると考えられる. ハミング窓やハミング窓, ブラックマン窓は, メインローブが太いが, サイドローブが低いスペクトルである. つまり, 周波数分解能が低い, スペクトル漏れは少ないと考えられる.

7 まとめ

本実験では, 1 時限時系列信号である音声データの分析を対象に, 周波数解析の実装, 窓関数の影響について理解することができた. C 言語を用いて離散フーリエ変換と高速フーリエ変換を実装し, その計算効率を比較した. 得られたデータを gnuplot を用いて波形観測を行うことができた. スペクトル漏れを防ぐために窓関数を適用することを知った. また, その窓関数にも様々な種類が存在して, 行うデータ分析によって使用する窓関数を選択する必要があることもわかった. 母音の音声データの周波数分析を行い, 発声内容によって振幅スペクトルのピーク位置が異なることを確認した. これにより, 音声の情報の差異が周波数成分の分布として表現されていることを実データから学んだ.

8 参考文献

- 1) Thomas Williams & Colin Kelley 「gnuplot 6.0」,
<http://www.gnuplot.info/docs/gnuplot-ja.pdf> (更新:2023/12) (参照:2026/04/29)
- 2) tatsukisaito 「フーリエ変換を用いて多項式の掛け算を行う」, Teckbot!,
<https://blog.applibot.co.jp/2020/12/16/fourier-transform/> (更新:2020/12/16) (参照:2026/04/29)
- 3) Takuya OOURA 「FFT (高速フーリエ・コサイン・サイン変換) の概略と設計法」, 京都大学数理解析研究所,
<https://www.kurims.kyoto-u.ac.jp/~ooura/fftman/> (参照:2026/04/29)
- 4) 参考書庫 「C++ 高速化 処理時間の計測」,
<http://www.sanko-shoko.net/note.php?id=rnfd> (更新:2018/07/13) (参照:2026/04/29)
- 5) MATLAB ヘルプセンター 「信号のパワーの測定」,
<https://jp.mathworks.com/help/signal/ug/measure-the-power-of-a-signal.html> (参照:2026/04/29)
- 6) e-Words 「ダウンサンプリング」,
<https://e-words.jp/w/ダウンサンプリング.html> (参照:2026/04/29)
- 7) 代表高瀬 「ノイズ対策講座-8 折り返し雑音とは?」, デルタテックラボラトリ,
https://deltatech-labo.com/2023/04/16/noise_seminar_08 (更新:2025/03/07) (参照:2026/04/29)
- 8) ANALOG DEVICES 「フィルタの基礎: アンチエイリアシング」,
<https://www.analog.com/jp/resources/technical-articles/guide-to-antialiasing-filter-basics.html> (更新:2002/01/11) (参照:2026/04/29)
- 9) kennzo の備忘録 「窓関数」,
<https://www.kennzo.net/window-function> (更新:2024/12/06) (参照:2026/04/29)

```
1 set term png
2 set output "kadai1.png"
3
4 fs = 16000.0
5
6 set xlabel "time [msec]"
7 set ylabel "amplitude"
8
9 set xrange [0:30]
10 set yrange [-1:1]
11
12 set size ratio 0.3
13
14 plot "sample01.sw" binary format="%int16" every :::479 \
15     using ($0 * 1000.0 / fs):($1/32767.0) \
16     with lines notitle
```

リスト 1: 課題 1 で作成した gnuplot スクリプト

```
1 #define _USE_MATH_DEFINES
2
3 #include <math.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <windows.h>
7
8 #define LENGTH 512 /* 信号長 ただし, 2の整数乗 */
9
10 typedef struct {
11     double r; /* 実部 */
12     double i; /* 虚部 */
13 } Complex; /* 複素数 */
14
15 /* プロトタイプ宣言 */
16 void bit_reversal(Complex *x, int N);
17 void fft(Complex *x, int N, int inverse);
18 void genTwiddleTable(Complex *, int);
19 Complex calcTwiddlePower(Complex *, int, int);
20 void dft(Complex *, Complex *, int);
21
22 Complex getComplex(double r, double i);
23 Complex cexptheta(double theta);
24 Complex addComplex(Complex a, Complex b);
25 Complex subComplex(Complex a, Complex b);
26 Complex timeComplex(Complex a, Complex b);
27 Complex divComplexByReal(Complex z, double div);
28 Complex powComplex(Complex, int);
29
30 int main(int argc, char *argv[]) {
31     int N = LENGTH; /* 信号長 */
32     double amp; /* 振幅 */
33
34     int i;
35
36     /* 時間計測用変数 */
37     LARGE_INTEGER start, end, freq;
38     double elapsedTime;
39     if (!QueryPerformanceFrequency(&freq)) {
40         fprintf(stderr, "High-resolution timer not supported.\n");
41         return 1;
42     }
```

```
43 FILE *fp = NULL;
44 if (argc > 1) {
45     fp = fopen(argv[1], "rb");
46     if (fp == NULL) {
47         fprintf(stderr, "File open error (%s)\n", argv[1]);
48         exit(1);
49     }
50 } else {
51     fprintf(stderr, "Usage: %s input-filename\n", argv[0]);
52     exit(1);
53 }
54
55 /* 入力信号配列 */
56 short input[LENGTH];
57 Complex x[LENGTH];
58 /* N点DFT係数 */
59 Complex c[LENGTH];
60
61 /* DFT係数の初期化 */
62 for (i = 0; i < N; i++) {
63     c[i] = getComplex(0.0, 0.0);
64 }
65
66 /* 音声データを読み込み */
67 if (fread(input, sizeof(short), N, fp) != (size_t)N) {
68     fprintf(stderr, "File read error (%s)\n", argv[1]);
69     fclose(fp);
70     exit(1);
71 }
72
73 /* 音声データを複素数型に変換 */
74 for (int i = 0; i < N; i++) {
75     x[i] = getComplex(input[i], 0);
76 }
77
78 QueryPerformanceCounter(&start);
79 dft(x, c, N);
80 QueryPerformanceCounter(&end);
81 elapsedTime = (double)(end.QuadPart - start.QuadPart) / freq.QuadPart;
82 printf("DFT Elapsed time: %f seconds\n", elapsedTime);
```

リスト 2: 入力データにフーリエ変換を行う C 言語プログラム (2/6)

```
83     QueryPerformanceCounter(&start);
84     fft(x, N, 0);
85     QueryPerformanceCounter(&end);
86     elapsedTime = (double)(end.QuadPart - start.QuadPart) / freq.QuadPart;
87     printf("FFT Elapsed time: %f seconds\n", elapsedTime);
88
89     fclose(fp);
90
91     return 0;
92 }
93
94 /**
95  * @brief 配列の要素をビット逆順に並べ替える
96  *
97  * @param x 信号配列
98  * @param N 信号長 ただし、2の整数乗
99  */
100 void bit_reversal(Complex *x, int N) {
101     int i, j, k;
102     Complex temp;
103     j = 0;
104     for (i = 0; i < N; i++) {
105         if (i < j) {
106             temp = x[i];
107             x[i] = x[j];
108             x[j] = temp;
109         }
110
111         /* 逆方向カウントアップ */
112         k = N >> 1;
113         while (k <= j && k > 0) { /* もしjのk番目のビットが1だったら */
114             j -= k; /* そのビットを0にする */
115             k >>= 1; /* 次のビットに移る */
116         }
117         j += k; /* 0を見つけたら1にする */
118     }
119 }
```

リスト 2: 入力データにフーリエ変換を行う C 言語プログラム (3/6)

```
120 /**
121  * @brief 高速フーリエ変換
122  *
123  * @param x 入力信号配列
124  * @param N 信号長 ただし、2の整数乗
125  * @param inverse 0ならDFT, 1ならIDFT
126  */
127 void fft(Complex *x, int N, int inverse) {
128     bit_reversal(x, N);
129
130     for (int stage = 1; (1 << stage) <= N; stage++) {
131         int block = 1 << stage; /* 現在のブロックのサイズ */
132         int block2 = block >> 1; /* ブロックサイズの半分 */
133         double theta = (inverse ? 2.0 : -2.0) * M_PI / block;
134         Complex wm = cexptheta(theta); /* 基本となる回転子 */
135
136         /* 各ブロックの計算 */
137         for (int k = 0; k < N; k += block) {
138             Complex w = getComplex(1.0, 0);
139             for (int j = 0; j < block2; j++) {
140                 /* バタフライ演算の本体 */
141                 Complex u = x[k + j]; /* X_0[k] */
142                 Complex v =
143                     timeComplex(w, x[k + j + block2]); /* W^k * X_1[k] */
144
145                 x[k + j] = addComplex(u, v);
146                 x[k + j + block2] = subComplex(u, v);
147
148                 w = timeComplex(w, wm);
149             }
150         }
151     }
152
153     /* 逆変換の場合、最後に N で割る */
154     if (inverse) {
155         for (int i = 0; i < N; i++) {
156             x[i] = divComplexByReal(x[i], N);
157         }
158     }
159 }
```

リスト 2: 入力データにフーリエ変換を行う C 言語プログラム (4/6)

```
160 /* 回転子W_Nの0乗からN-1乗を計算 */
161 void genTwiddleTable(Complex *W, int N) {
162     int i;
163     for (i = 0; i < N; i++) {
164         W[i] = getComplex(cos(i * 2.0 * M_PI / N), -sin(i * 2.0 * M_PI / N));
165     }
166 }
167
168 /* 回転子W_Nのn乗のk乗を計算 */
169 Complex calcTwiddlePower(Complex *W, int n, int k) {
170     return powComplex(W[n], k);
171 }
172
173 /* 離散フーリエ変換 */
174 void dft(Complex *x, Complex *c, int N) {
175     int k, n;
176     Complex W[LENGTH];
177
178     genTwiddleTable(W, N);
179
180     /* 信号長で反復 */
181     for (k = 0; k < N; k++) {
182         /* シグマループ */
183         for (n = 0; n < N; n++) {
184             c[k] =
185                 addComplex(c[k], timeComplex(x[n], calcTwiddlePower(W, k, n)));
186         }
187     }
188 }
189
190 Complex getComplex(double r, double i) {
191     Complex c;
192     c.r = r;
193     c.i = i;
194     return c;
195 }
196
197 Complex cexptheta(double theta) { return getComplex(cos(theta), sin(theta)); }
198
199 Complex addComplex(Complex a, Complex b) {
200     return getComplex(a.r + b.r, a.i + b.i);
201 }
```

```
202 Complex subComplex(Complex a, Complex b) {
203     return getComplex(a.r - b.r, a.i - b.i);
204 }
205
206 Complex timeComplex(Complex a, Complex b) {
207     return getComplex(a.r * b.r - a.i * b.i, a.r * b.i + a.i * b.r);
208 }
209
210 Complex divComplexByReal(Complex z, double div) {
211     return getComplex(z.r / div, z.i / div);
212 }
213
214 /* 第一引数の第二引数乗を返す
215  * 第二引数が負の場合は,第一引数の大きさが1のときのみ正しく計算される
216  */
217 Complex powComplex(Complex z, int pow) {
218     if (pow == 0)
219         return getComplex(1.0, 0.0);
220
221     int i;
222     int p = (pow > 0) ? pow : -pow;
223     Complex ret = z;
224     for (i = 0; i < p - 1; i++) {
225         ret = timeComplex(ret, z);
226     }
227
228     /* |z|==1のとき,zの-1乗はzの共役複素数 */
229     if (pow < 0)
230         ret.i = -ret.i;
231
232     return ret;
233 }
```

リスト 2: 入力データにフーリエ変換を行う C 言語プログラム (6/6)

```
1 for(i = 0; i < N; i++) {
2     amp = sqrt(c[i].r * c[i].r + c[i].i * c[i].i);
3     printf("%f\n", amp);
4 }
```

リスト 3: 離散フーリエ変換済みデータを出力する C 言語プログラム

```
1 set term png
2 set output "output.png"
3
4 fs = 16000.0
5 N_fft = 512.0
6
7 set xlabel "Frequency [kHz]"
8 set ylabel "Amplitude"
9
10 set xrange [0:8]
11 set grid
12
13 plot "output.txt" using ($0 * fs / N_fft / 1000):1 \
14     with lines notitle
```

リスト 4: 振幅スペクトル (単位なし) を描画する gnuplot スクリプト

```
1 set term png
2 set output "output.png"
3
4 fs = 16000.0
5 N_fft = 512.0
6
7 set xlabel "Frequency [kHz]"
8 set ylabel "Amplitude [dB]"
9
10 stats "output.txt" using 1 nooutput
11
12 set xrange [0:8]
13 set grid
14
15 plot "output.txt" using ($0 * fs / N_fft / 1000):(20 * log10(($1 > 1e-10 ? $1 : 1e
    -10) / STATS_max)) \
16     with lines notitle
```

リスト 5: 振幅スペクトル (デシベル) を描画する gnuplot スクリプト

```
1 void applyWindow(Complex *x, int N) {
2     for (int i = 0; i < N; i++) {
3         double w = 0.5 - 0.5 * cos(2 * M_PI * i / (N - 1)); // ハニング窓
4         x[i].r *= w;
5         x[i].i *= w;
6     }
7 }
```

リスト 6: 複素数配列に窓関数を適用する C 言語プログラム

```
1 int main() {
2     int N = LENGTH; /* 信号長 */
3     double amp; /* 振幅 */
4
5     int i;
6
7     double start_sec = 2.454; /* 読み込む音声の開始位置 [秒] */
8
9     long fs = 16000; /* 標本化周波数 */
10    long bytes_per_sample = 2; /* 16bit なので 2 バイト */
11    long start_byte = (long)(start_sec * fs * bytes_per_sample); /* 指定時刻の開始位置バ
        イト */
12
13    /* 入力信号配列 */
14    short input[LENGTH];
15    Complex x[LENGTH];
16
17    FILE *fp = fopen ( "j22330.sw", "rb" );
18    if ( fp == NULL ) {
19        fprintf ( stderr, "File open error (j22330.sw)\n" );
20        exit ( 1 );
21    }
22
23    /* 読み込む位置にファイルポインタを移動 */
24    fseek(fp, start_byte, SEEK_SET);
```

リスト 7: 課題 4 で使用する main 関数の C 言語プログラム (1/2)

```
25  /* 音声データを読み込み */
26  if (fread(input, sizeof(short), N, fp) != (size_t)N) {
27      fprintf(stderr, "File read error (j22330.sw)\n");
28      fclose(fp);
29      exit(1);
30  }
31
32  /* 音声データを複素数型に変換 */
33  for(int i = 0; i < N; i++) {
34      x[i] = getComplex(input[i], 0);
35  }
36
37  /* 窓関数の適用 */
38  applyWindow(x, N);
39
40  /* FFTの実行 */
41  fft(x, N, 0);
42
43  /* 振幅スペクトルの出力 */
44  for(int i = 0; i < N; i++) {
45      amp = sqrt(x[i].r * x[i].r + x[i].i * x[i].i);
46      printf("%f\n", amp);
47  }
48
49
50  fclose(fp);
51
52  return 0;
53 }
```

```
1 #define _USE_MATH_DEFINES
2
3 #include<stdio.h>
4 #include<math.h>
5 #include<stdlib.h>
6
7 #define LENGTH 512 /* 信号長N ただし, 2の整数乗 */
8
9 /* プロトタイプ宣言 */
10 double calcAmpPower(short *x, int N);
11 void applyWindow(short *x, int N);
12
13 int main(int argc, char *argv[]) {
14     /* 信号長 */
15     int N = LENGTH;
16
17     double beforePower, afterPower;
18
19     int i;
20
21     FILE *fp = NULL;
22     if (argc > 1) {
23         fp = fopen(argv[1], "rb");
24         if (fp == NULL) {
25             fprintf(stderr, "File open error (%s)\n", argv[1]);
26             exit(1);
27         }
28     } else {
29         fprintf(stderr, "Usage: %s input-filename\n", argv[0]);
30         exit(1);
31     }
32
33     /* 入力信号配列 */
34     short input[LENGTH];
35
36     /* 音声データを読み込み */
37     if (fread(input, sizeof(short), N, fp) != (size_t)N) {
38         fprintf(stderr, "File read error (%s)\n", argv[1]);
39         fclose(fp);
40         exit(1);
41     }
```

```
42     beforePower = calcAmpPower(input, N);
43
44     applyWindow(input, N);
45
46     afterPower = calcAmpPower(input, N);
47
48     printf("BEFORE : %16f\n", beforePower);
49     printf("AFTER  : %16f\n", afterPower);
50     printf("DIFF  : %16f\n", beforePower - afterPower);
51     printf("RATIO : %16f\n", afterPower / beforePower);
52
53     fclose(fp);
54
55     return 0;
56 }
57
58 /* short 型配列の平均パワーを求める関数 */
59 double calcAmpPower(short *x, int N) {
60     double amp;
61     double power = 0.0;
62     for (int i = 0; i < N; i++) {
63         amp = x[i] * x[i];
64         power += amp;
65     }
66     return power / N;
67 }
68
69 /* 配列にハニング窓を適用 */
70 void applyWindow(short *x, int N) {
71     for (int i = 0; i < N; i++) {
72         double w = 0.5 - 0.5 * cos(2 * M_PI * i / (N - 1)); // ハニング窓
73         x[i] *= w;
74     }
75 }
```

リスト 8: 信号配列の平均パワーを求める C 言語プログラム (2/2)