平成 26 年度卒業論文

プレイリスト自動編成アルゴリズムの研究

函館工業高等専門学校　情報工学科　5 年

東海林研究室　鳴海洸樹

目次

1章　序論

1.1 研究目的

本研究では、複数の楽曲をテンポやムードなどの様々な観点から楽曲間が自然な繋がりとなるようなプレイリストを編成するアルゴリズムの研究・開発を行う。また、このアルゴリズムを組み込み、更に楽曲間を自然に繋げて再生する Android 用音楽プレイヤーの開発を行う。

本研究の目的

(1) 楽曲間の繋がりを考えたプレイリストの編成アルゴリズムの研究・開発

(2) (1)のアルゴリズムを組み込んだ Android 用音楽プレイヤーの開発

1.2 研究背景

現在世の中に出回っているプレイリストを自動編成するアルゴリズムを用いている音楽プレイヤーの大半は全ての楽曲に対して完全にランダムに選曲している。一部のプレイヤーではテンポ解析、ムード解析、協調フィルタリングなどの技術を利用して選曲しているが[1]、そのような既存のプレイヤーはプレイリストの自動編成はできるものの、楽曲再生時の音の繋がりは考慮していない。そのため、プレイリストを自動編成していても再生時に楽曲間の繋ぎ目で違和感が発生してしまうという問題点が存在する[2]。

1.3 類似研究

類似研究をいくつか調べた中で、楽曲情報を利用した多次元ベクトルを用いて音楽マップ上に配置するもの[3]、Web 上の音楽情報サイトから取得したキーワードを用いて分類するもの[4]、ユーザの嗜好を集めて強調フィルタリングによって生成するもの[4]、ユーザの再生履歴などを利用した楽曲推薦を行うもの[5]など様々な手法が考案されていたが、いずれもインターネット環境が必要であること、個人利用できるプレイヤーが無かったことから、本研究ではインターネット環境が無い状態でも個人で利用出来るアルゴリズムとプレイヤー作成を目的とした。

1.4 英文 Abstract

Study of Playlist Automatic Organization Algorithm

Abstract: In the past, some playlist automatic schedulings performed only rough categorization, and excluded the relationship between the two songs in continuous playbacks. The purpose of this study is to create a playlist that takes into connections between the two songs in addition to an existing playlist creation technology. Besides, the purpose is to play the songs in the playlist in natural connections. Our application analyzes information of the songs in order to create the playlist, and sets the characteristics of the songs as multidimensional vectors. Then this application weights the Square Euclidean distance vectors for each songs, and organizes the playlist using the vectors. Additionally, the application can play the songs in the playlist in natural connections using techniques such cross-fade. Therefore, the conclusion is that it is possible to organize the playlist that has natural connections between the two songs by applying the techniques used in this study.

Key words: playlist, song, automatic scheduling, Square Euclidean distance vector, natural connection

２章　プレイリスト編成アルゴリズムについて

2.1 プレイリスト編成手順

本研究において開発したプレイリストの編成手順について説明する。本研究ではプレイリストの編成において各楽曲の特徴を調べる必要がある。そのため、まず初めに(1)BPM の測定、(2)Chroma ベクトルの測定、(3)アルバム名などの楽曲情報の読み込みを行い、それら(1)(2)(3)の情報を元に(4)楽曲間の類似度を求める。なお、今回の研究ではプレイリストに多少のランダム性を持たせるために、全楽曲から起点となる 1 曲をランダムに選出し、その曲と類似度が近い 3 曲からランダムに 1 曲を選び、次に再生する楽曲として選択する。同様の動作を最後まで繰り返すことによってプレイリストを編成する。

2.2 BPM の測定によるテンポ解析

楽曲の特徴を調べるにあたってまず初めに、BPM（Beat Per Minutes）の測定によるテンポ解析を行う[6]。BPM とは、楽曲の 1 分間の拍数を表す数値である。この拍数が高いほど、テンポが速い楽曲であると言え、拍数が低いほど、テンポが遅い楽曲であると言える。また、楽曲間を連続再生するにあたって、このテンポの違いは聴覚上の違和感に強く影響を与える。

【具体的な計算方法について】

ある楽曲に対し、一定時間間隔で高速フーリエ変換（FFT : Fast Fourier Transform）を利用し、求めた周波数毎の振幅成分を合計することにより、一定時間間隔における楽曲の音量を求める（図 2.1）。

次に、求めた音量に対して連続する時刻間での差分を求める。つまり、ある時刻⊿ｔと⊿ｔ+1 への音量の変化量を求める。この際、変化量が負の値になった場合は変化量を 0 とすることにより、一定時間間隔における音量の増加量を求める（図 2.2）（図 2.3）。

増加量のみを求める理由は、本研究ではテンポを測定する方法の一つとして、音の出始めの間隔を計算する方法を用いたためである。音の出始めは音量の増加量が急激に上昇するため、音量の増加量を取得することによりテンポの測定が可能になる。更に位相を計算する際に拍の開始位置を合わせることも出来る。

次に、求めた音量の増加量を用いて、以下の式によって BPM の推定を行う。次式の R が最大になる bpm の値が楽曲の BPM として推定される。

$$\boldsymbol{R} = \sqrt{c\_sum^2 + s\_sum^2}$$

$$c_{sum} = \frac{1}{N}\sum_{n=0}^{N-1} D(n)\cos\left(2\pi * \frac{bpm}{60}\frac{n}{s}\right) * win\,(n)$$

$$s_{sum} = \frac{1}{N}\sum_{n=0}^{N-1} D(n)\sin\left(2\pi * \frac{bpm}{60}\frac{n}{s}\right) * win(n)$$

n：配列のサンプル数

D(n)：増加量の配列

s：サンプリングレート

bpm：推定する BPM (60～240 の範囲で可変させる)

win(n)：窓関数（端での誤差を減らす為に掛ける）

ここで本研究では、窓関数として以下のハン窓を使用した。

$$win(n) = 0.5 - 0.5 * cos\left(2\pi * \frac{n}{N}\right)$$

拍位置は以下の式によって求められる。

$$\mathrm{T} = \tan^{-1}\frac{s_{sum}}{c_{sum}}$$

なお BPM と拍位置の推定関連のソースコードは「付録 1」に示してある。

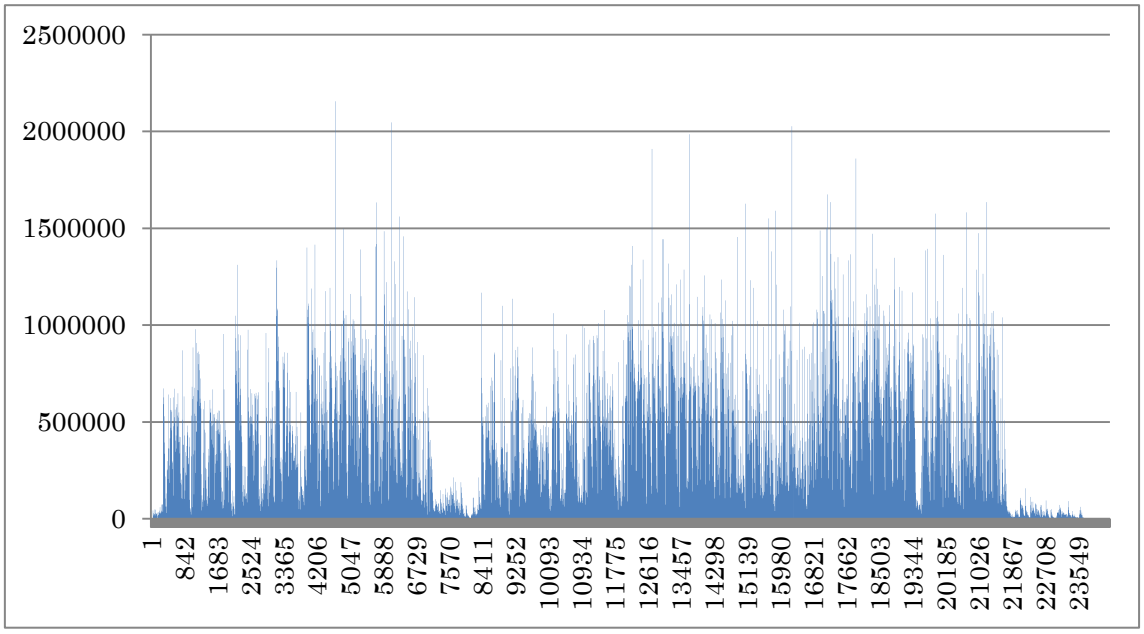

図 2.1　ある楽曲の 時間―音量グラフ

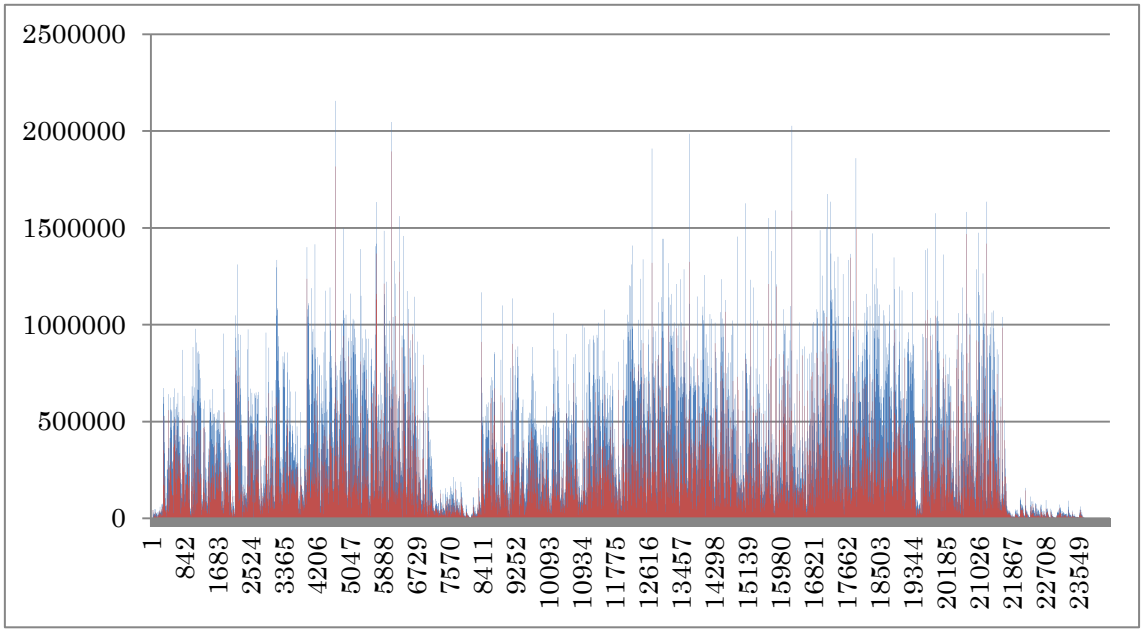

図 2.2　図 2.1 に時間―増加量グラフを加えたグラフ

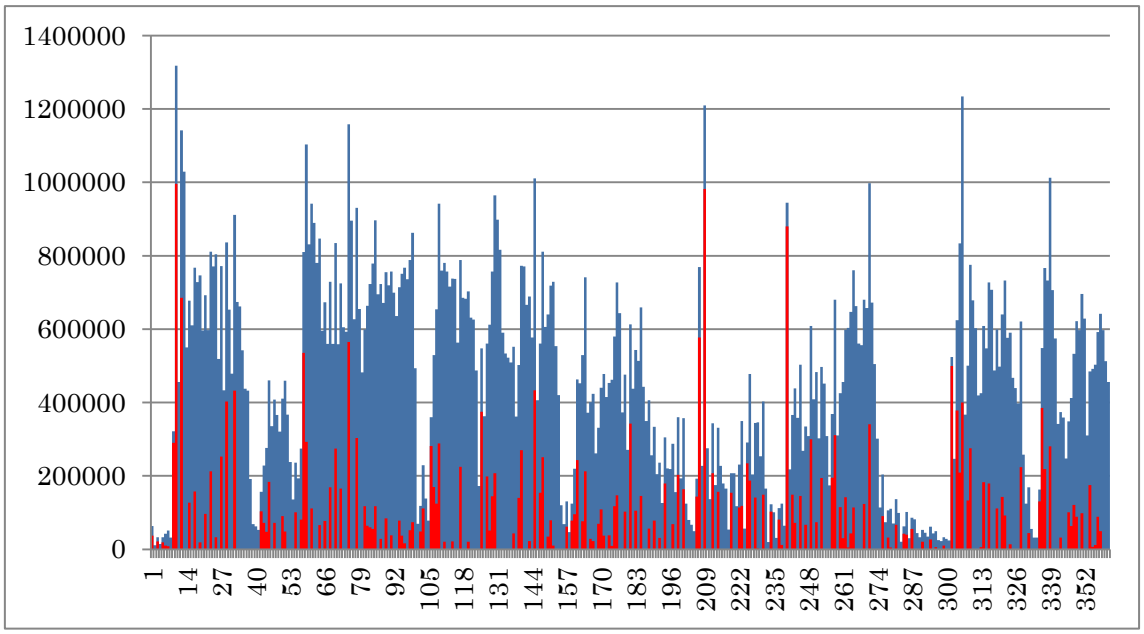

図 2.3　図 2.2 の一部分を拡大したグラフ

2.3 Chroma ベクトルの測定によるムード解析

楽曲の特徴を調べるにあたって次に、12 次元の Chroma ベクトルの測定によるムード解析を行う[7][8]。Chroma ベクトルとは、音を 12 音階で離散分布させた特徴量のベクトルであり、オクターブが違う同じ音階の音量を 1 つにまとめたベクトルである。

【具体的な計算方法について】

2.2 節と同様に楽曲に対し FFT を実施し、一定時間間隔における周波数毎の振幅成分を得る。その振幅成分に対して周波数が最も近い音階を探索し、Chroma ベクトルの対応する音階にその振幅成分を加算する。ここで Chroma ベクトルの例を図 2.4 に示す。また参考としてハ長調・オクターブ 4 における周波数と音階の対応を表 2.1 に示す。

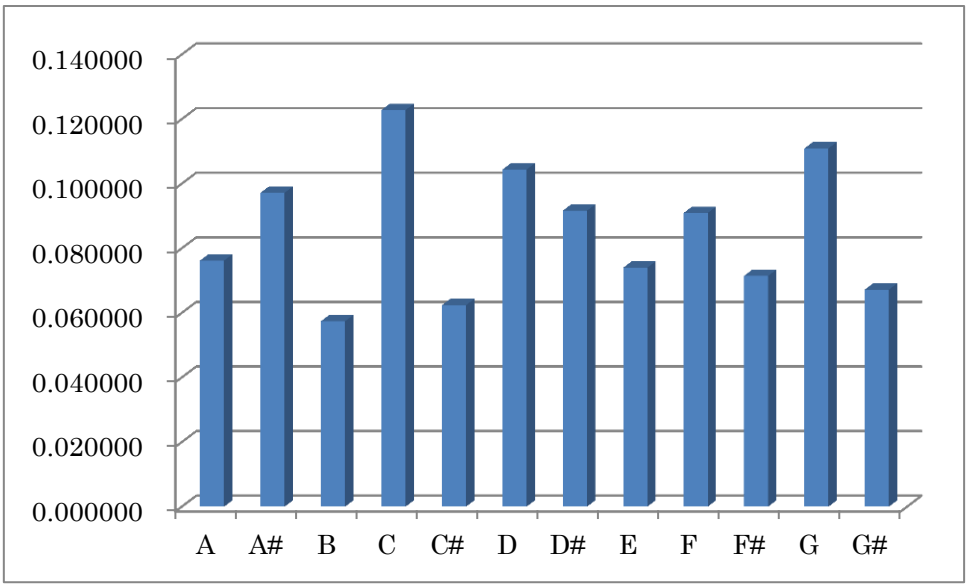

図 2.4　ある楽曲の Chroma ベクトル例

表 2.1　音階の例（ハ長調・オクターブ 4）

音階 (ハ長調)	周波数 (Hz)
C（ド）	261.6256
C#（ド#）	277.1826
D（レ）	293.6648
D#（レ#）	311.127
E（ミ）	329.6276
F（ファ）	349.2282

音階 (ハ長調)	周波数 (Hz)
F#（ファ#）	369.9944
G（ソ）	391.9954
G#（ソ#）	415.3047
A（ラ）	440
A#（ラ#）	466.1638
B（シ）	493.8833

本研究で扱う音階は 12 平均律を使用し、その音階と周波数の対応については、以下の様な計算式で求められる。

1. オクターブ 4 のラ音（o4A と表記）＝440Hz　を基準とする

2. 1 音上がるごとに前の音の周波数を $\sqrt[12]{2}$ 倍する

3. 1 音下がるごとに前の音の周波数を $\frac{1}{\sqrt[12]{2}}$ 倍する

例えば o4A から 1 オクターブ上がり o5A になると、$440Hz * \left\{\sqrt[12]{2}\right\}^{12} = 880Hz$ と丁度 2 倍になる。

なお、本研究では低音域と高音域は除外して o2A（110Hz）から o8G#（13289.75Hz）までの 7 オクターブ間の振幅成分を用いて Chroma ベクトルを作製した。また Chroma ベクトル推定関連のソースコードは「付録 2」に示してある。

2.4 楽曲間類似度の算出

次に、楽曲の類似度の算出について本実験で採用した手法について解説する。本実験では、ある 2 曲間の類似度算出において BPM, 12 次元の Chroma ベクトル, アーティスト, 作曲者, アルバムの各 16 項目での比較及び計算を行った後、それらを総合評価することによって類似度を求めた[9][10]。本研究では楽曲 x と y の間の類似度は以下の通りとした。

$$d_{bpm} = |bpm[x] - bpm[y]|^{\mathbf{1.2}} * \mathbf{0.0001}$$

$$d_{chroma} = \left\{\sum_{i=0}^{11}(Chroma[x][i] - Chroma[y][i])^2\right\}^{\mathbf{1.3}}$$

$$EQ_{artist} = \begin{cases} 0.00, & (artist[x] = artist[y]) \\ \mathbf{0.25}, & (artist[x] \neq artist[y]) \end{cases}$$

$$EQ_{composer} = \begin{cases} 0.00, & (composer[x] = composer[y]) \\ \mathbf{0.20}, & (composer[x] \neq composer[y]) \end{cases}$$

$$EQ_{album} = \begin{cases} 0.00, & (album[x] = album[y]) \\ \mathbf{0.10}, & (album[x] \neq album[y]) \end{cases}$$

$$\boldsymbol{d_{mix}} = d_{bpm} + d_{chroma} + EQ_{artist} + EQ_{composer} + EQ_{album}$$

なお今回は、BPM にはマンハッタン距離を求めた後 1.2 乗し 0.0001 倍の重み付け、Chroma ベクトルには平方ユークリッド距離を求めた後 1.3 乗している。アーティスト、作

曲者、アルバムにおいては離散距離を求めそれぞれ 0.25 倍、0.20 倍、0.10 倍の重み付けをしている。この重み付けに関して本研究では手探りで適当な値を探した。この重み付けが極端な方向に偏った場合、生成されるプレイリストも極端な特徴を持つことになる。例えば、BPM の重みを重視した場合はテンポが優先されるため曲間の繋がりは良くなるが、明るいムードの曲から暗いムードの曲へと急な転調が起こる可能性が高くなる。逆に、Chroma ベクトルの重みを重視した場合、明るいムードの曲の次も明るいムードの曲といったように曲調に関しては繋がりが良くなるが、テンポの速い曲から急にゆったりとした曲へ変わる可能性が高くなる。また、同じアルバムや同じアーティストでの連続再生を重視するか否かなども、該当項目の重み付けの変更によって調整できる。

なお楽曲間類似度算出関連のソースコードは「付録 3」に示してある。

3章　アルゴリズムの実用に向けた開発

3.1 Android 用音楽プレイヤーの開発

本研究で提案したプレイリスト自動編成アルゴリズムを組み込み、更にギャップレス再生、クロスフェード、ビートマッチング等の技術を利用して楽曲間を自然に繋げて再生するアプリケーションを開発した（図 3.1）。

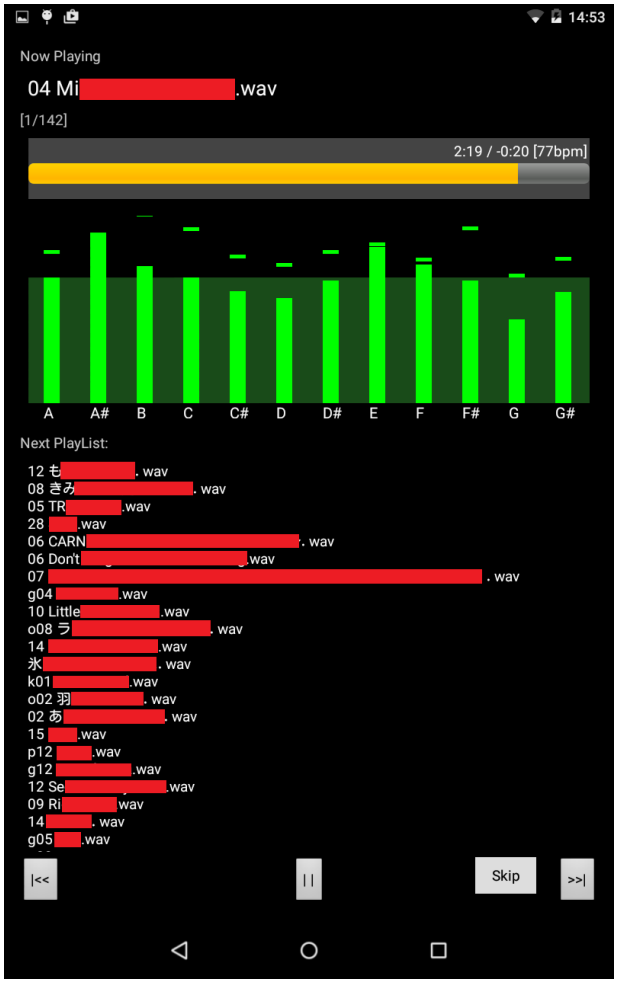

図 3.1　開発したアプリケーションの実行画面

開発環境などは以下の通りである。

【アプリケーションの動作環境】
・対象楽曲：wav 音源（16bit モノラル）
・対象端末：Android 2.2～

【開発環境】
・Eclipse Luna
・AndroidSDK 21

【使用したデータベース管理システム（DBMS）】
・SQLite（Android 端末内）

【使用したライブラリ】
・AudioTrack（AndroidAPI 内）
・JTransForms（無料配布外部ライブラリ）

楽曲情報解析に高速フーリエ変換（FFT）を利用しているが、これを高速に行う為に JTransForms を利用した[11]。また、本プレイヤーでは楽曲の連続再生を行うため、AudioTrack というクラスを利用した。楽曲の解析情報等の保存には Android 端末上で動作する SQLite を利用した。

3.2 曲間合成部分の出力

曲間の繋ぎ目部分では、自然な繋がりとなるように音楽データを合成して Stream に出力する。ここではその出力データの生成について説明する。

まず、曲の前後の無音部分をカットするためにギャップレス再生を実装した。前曲の末尾と次曲の先頭から探索し、あるしきい値以上の音量が出る地点を見つけて無音部分をカットする。無音部分を無くす事によって繋ぎ目で音が途切れるのを防ぐ事ができる。

次に、前曲から次曲へのテンポが一致する状態で合成するためにビートマッチングを実装した。2.2 節で求めた BPM と拍位置を利用し、合成する曲間で拍の位置が合うようにオフセットを設定する。

最後に、前曲から次曲へ自然と音が変わるようにするためクロスフェード（又はフェードアウトとフェードイン）を実装した。合成部において前曲の音量を徐々に下げ、次曲の音量を徐々に上げて交差させる。

これらの計算処理を行うことにより、繋ぎ目部分の出力波形を生成した（図 3.2)。具体的なプログラミング手法関連のソースコードは「付録 4」「付録 5」に示してある。

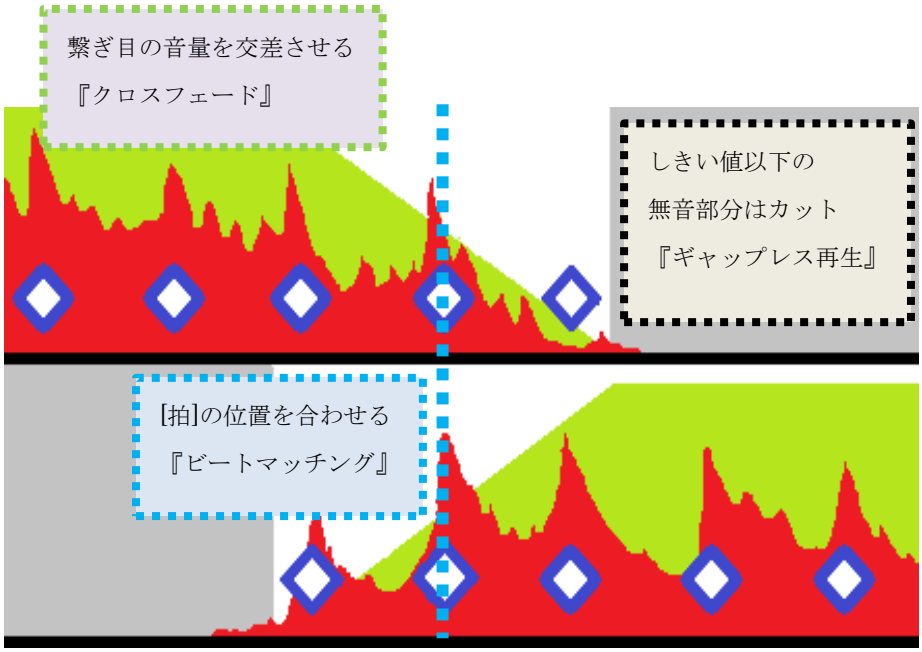

図 3.2　合成部分イメージ

3.3 マルチスレッド処理

本アプリケーションではいくつかの処理を並行に実行する為に以下の 4 スレッドを起動している。またスレッドの動作状況を図 3.3 に示す。

・楽曲情報の解析と、プレイリストの生成を行うスレッド
・プレイリストを元に楽曲を連続再生するスレッド
・曲間の繋ぎ目部分の波形を先読みして生成するスレッド
・アプリ画面にグラフなどの情報を書き込むスレッド

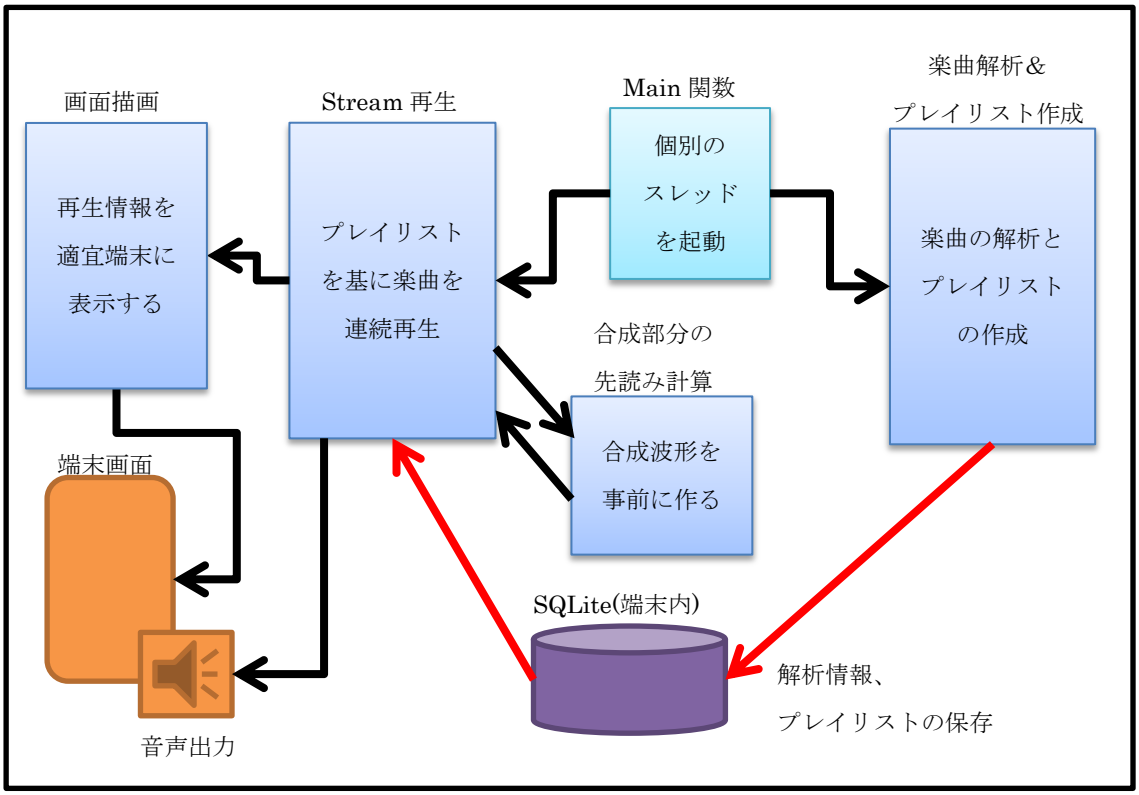

図 3.3　スレッド動作状況

4章　実験

4.1 実験概要

本研究で開発したアルゴリズムを組み込んだ音楽プレイヤーアプリケーションを用いて、楽曲間が自然な繋がりとなって再生できているかどうかの実験を行った。実験方法は以下の様に行った。

【実験方法】
・任意に選んだ142曲を対象にした
・18から20歳までの7名を被験者とした
・作成したアプリケーションを実際に利用して、あらかじめ設定した評価項目に基づいた5段階評価を行った
・他に気がづいた点について自由記述評価を行った

【評価項目】
・アプリケーションの出来ばえ（総合評価）
・似た曲での連続再生になっているか
・曲と曲のテンポが近い感じであるか
・曲のつながりに違和感はないか
・自由記述

4.2 結果

結果を表4.1に示す。総合評価の平均値は4.00という結果になった。

表 4.1 アンケート結果

被験者	アプリケーションの出来ばえ（総合評価）	似た曲での連続再生になっているか	曲と曲のテンポが近い感じであるか	曲のつながりに違和感はないか
A	4	4	5	4
B	4	4	3.5	4
C	4	4	4	4.5
D	4	4	5	4
E	4	4	4	4
F	4	4	4	4
G	4	5	3	4
平均	4.00	4.14	4.07	4.07

また自由記述評価では以下の様なコメントが得られた。

・元の曲自体がフェードイン、アウトしていると少し違和感がある
・違和感があるところはテンポも変わるように聞こえたし、感じないところはテンポも近く、曲によっては変わったことに気づかないくらい違和感はないので全体としては良かったと思う
・曲と曲のつながりがすごくきれいだった
・前の曲がフェードアウトしているとつながりに違和感があった
・テンポが近い曲同士は違和感がなかった
・歌の途中で次の曲に切り替わってしまうところが少し違和感があった
・曲同士のつながりが割りと違和感なく聞こえた
・はじまりの部分だけテンポが違う曲とのつなぎが上手く行ってないと思った
・曲調の似ている曲とのつなぎは上手いと思った

次章にてこの結果についての考察を行う。

5章　考察

今回の実験で得られた評価から、プレイリストの自動編成は概ね上手く出来ていると考えられる。しかし課題として、楽曲再生時における繋ぎ目において改善すべき点がいくつか見つかった。例えばイントロが少なく曲の出だしから歌詞があるような曲の場合は出だしの歌詞が聞こえなくなる事があり、そのような場合に違和感があった。また、イントロ部分と A メロ部分のテンポが違う曲の場合に拍が一致しない場合があった。まとめると以下の 2 つの問題点があげられる。

(1) 曲の出だしにイントロが少ない曲に対して過剰に音量を下げてしまっている
(2) 曲中でテンポが変わる曲に対する配慮が足りない

これらに対する改善策として以下の改善案を考えた。

(1) 曲の出だしのイントロの長さを測り、その長さに応じてフェードインする際の音量の減少量を調節する。これにより、イントロが短い曲に対して歌詞がはっきり聞き取れる音量にする事ができると思われる

(2) 曲調の変わり目部分でのテンポの計算を分割して行うことによって変化に対応させる。これにより、曲中でテンポが変わる曲に対応できると思われる

また、今回の実験では 142 曲をプレイリストの対象としたため特に問題にはならなかったが、再生する楽曲として登録したサンプル数が極端に少ない場合、類似度が低い楽曲が選ばれる可能性が高くなるという問題がある。これに対しては対象とする楽曲数をある程度確保する事によって解決される。

また今回の研究における次曲の選曲には、プログラムを簡略化するため類似度が最も近い 3 曲からランダムに選択したが、プレイリストの最初の方と最後の方で一定の類似性を保つには、巡回セールスマン問題と置き換える事による選曲方法が最適な有効手段の一つと言える[3][12]。

6章　まとめ

今回の研究では新たなプレイリストの編成アルゴリズムを開発した。加えて、そのアルゴリズムを組み込み、更に楽曲間を自然につなげて再生する Android 用音楽プレイヤーの開発を行った。

評価実験の結果、概ね実用化できるアルゴリズムとなっている事は証明できたが、考察の章で挙げたように改善の余地が残っていると思われる。また、本研究では対象としなかった mp3 などの圧縮形式の楽曲に対して本アルゴリズムとアプリケーションを対応させ、尚且つ楽曲情報を ID3 タグなどから読み取る仕組みを追加する事によって、更に本研究の利用の幅が広まると思われる。

参考文献

[1] 音楽情報処理 http://www.crestmuse.jp/handbookMI/.

[2] 野元 悠一: 音響的特徴を利用した楽曲間処理による音楽連続再生手法の提案, https://dspace.wul.waseda.ac.jp/dspace/bitstream/2065/13151/1/Honbun-t3606u081.pdf, (2007).

[3] Pitoyo Hartono, Ryo Yoshitake: *Journal of Signal Processing, Vol.17, No1,* Automatic Playlist Generation from Self-Organizing Music Map, p11, (2013).

[4] 渡邉 岳志, 服部 哲, 速水 治夫: 楽曲のキーワードの類似度を用いたプレイリスト作成支援システム, 研究報告グループウェアとネットワークサービス（GN）, 2001-GN-79, 14, pp1-6, (2011).

[5] 竹村 顕大朗: 再生履歴を考慮した選曲ポリシーの選択に基づく楽曲推薦システム, http://www.dl.kuis.kyoto-u.ac.jp/papers/2004/doc/bthesis-takemura.pdf, (2005).

[6] 倉島研, 金地美知彦, 畑山俊輝: *情報処理学会研究報告. [音楽情報科学]*, 楽曲の印象と好みに与えるテンポの影響, 2004(111), pp125-130, (2004)

[7] Anssi Klapuri, *Manuel Davy: Signal Processing Methods for Music Transcription*, **11**, p.346, (2006).

[8] 黒川奈桜子, 斎藤博昭: *情報科学技術フォーラム講演論文集*, E-007 スペクトルディップとクロマベクトルの併用による和音推定, Vol.11, No.2, pp165-170, (2012).

[9] 石井健一郎, 上田修功, 前田英作, 村瀬洋: *わかりやすいパターン認識*, **6**, (1998).

[10] 浜本義彦: *統計学的パターン認識入門*, **4**, (2009).

[11] JTransforms - Piotr Wendykier, https://sites.google.com/site/piotrwendykier/software/jtransforms.

[12] 津谷 篤, 田中 敦: *日本感性工学会研究論文集*, 巡回セールスマン問題解法を用いたプレイリスト作成とその効果, Vol.7, No.2, pp355-363, (2007).

付録 1. BPM 及び拍位置解析処理

```
private int FFT_SIZE = 0;

public CalcBpm(int FFTsize) {
	FFT_SIZE = FFTsize; // FFT_SIZE はインスタンス作成時に指定 (512 サンプル)
}

public int[] getBPM(WAVEFORMAT m_wavfmt, FileInputStream fis, String fname) {

	DoubleFFT_1D fft = new DoubleFFT_1D(FFT_SIZE);

	byte[] m_buf = new byte[FFT_SIZE * 2];
	double[] data = new double[FFT_SIZE * 2];
	int i;
	int row, col;

	int m_width = (m_wavfmt.data_size / 2) / FFT_SIZE;
	int m_height = FFT_SIZE / 2;

	double[] m_power = new double[m_height];// パワースペクトル(フレーム毎に更新)
	double[] FmVol = new double[m_width]; // フレーム毎の音量合計値

	// 一定区間毎に読み込み解析処理開始
	for (col = 0; col < m_width; col++) {

		// 読み込み処理
		try {
			int len = fis.read(m_buf); // 一定区間毎のファイル読み込み
			if (len < m_buf.length) {
				Log.d("CalcBpm", "read error!" + len + ":" + m_buf.length);
			}
		} catch (IOException e) {
			e.printStackTrace(); //ここでのエラー処理はほぼ実装していない
		}

		// オリジナルデータ byte[]を double[]に変換
		for (i = 0; i < FFT_SIZE; i++) {
			data[i] = (short) ((m_buf[i * 2 + 1] * 0x100) & 0xFFFF | (m_buf[i * 2] & 0xFF));
		}

		for (i = FFT_SIZE; i < FFT_SIZE * 2; i++) {
			data[i] = 0.0; // data[]の残り後半を 0 で埋める
		}

		// フーリエ変換(FFT)の実行 (data[0]は実数,[1]は虚数 ～ data[n]は実数,[n+1]は虚数)
		fft.realForwardFull(data);

		// パワースペクトル計算
		for (row = 0; row < m_height; row++) {
			m_power[row] = Math.sqrt(data[row * 2] * data[row * 2]
					+ data[row * 2 + 1] * data[row * 2 + 1]);
		}

		// フレーム毎の音量合計値を取得
		FmVol[col] = 0;
		for (row = 1; row < m_height; row++) {
			FmVol[col] += Math.pow(
					Math.sqrt(Math.pow(data[row * 2], 2.0)
							+ Math.pow(data[row * 2 + 1], 2.0)), 2.0)
					+ Math.pow(Math.sqrt(Math.pow(data[row * 2 + m_height],2.0)
							+ Math.pow(data[row * 2 + 1 + m_height], 2.0)),2.0);
		}
		FmVol[col] = Math.sqrt(Math.pow(FmVol[col], 2.0) / FFT_SIZE); // 一定区間毎の音量

	} // FFT の終了
```

```
        // BPM 取得解析処理開始
        double[] FmDiffVol = new double[m_width]; //  フレーム毎の音量差分合計値(負は足さない)

        //  フレーム毎の音量差分を取得(負は足さない)
        FmDiffVol[0] = FmVol[0];
        for (col = 1; col < m_width; col++) {
                FmDiffVol[col] = (FmVol[col] - FmVol[col - 1]);
                if (FmDiffVol[col] < 0){ FmDiffVol[col] = 0; }
        }

        double rate = m_wavfmt.rate / FFT_SIZE; //  サンプリング周波数（ヘッド読み込み）
        int Bmin = 60; //  推定する BPM の下限
        int Bmax = 240; //  推定する BPM の上限
        double[] cos_w = new double[Bmax - Bmin + 1]; // cos 用配列
        double[] sin_w = new double[Bmax - Bmin + 1]; // sin 用配列
        double[] r_w = new double[Bmax - Bmin + 1]; // BPM のマッチ度
        double c_sum, s_sum, win; //  解析用 tmp 変数

        // BPM 解析
        int mainBPM = -1;
        double BPMs = -1;
        for (int bpm = Bmin; bpm <= Bmax; bpm++) {
                c_sum = s_sum = 0;
                for (col = 0; col < m_width; ++col) {
                        win = 0.5 - 0.5 * Math.cos(2.0 * Math.PI * col / m_width); //  窓関数の一つ(ハン窓)
                        c_sum += FmDiffVol[col]
                                        * Math.cos(2.0 * Math.PI * ((double) bpm / 60.0) * col / rate) * win;
                        s_sum += FmDiffVol[col]
                                        * Math.sin(2.0 * Math.PI * ((double) bpm / 60.0) * col / rate) * win;
                }
                cos_w[bpm - Bmin] = c_sum / m_width;
                sin_w[bpm - Bmin] = s_sum / m_width;
                r_w[bpm - Bmin] = Math.sqrt(Math.pow(cos_w[bpm - Bmin], 2.0) + Math.pow(sin_w[bpm - Bmin], 2.0));
                if (BPMs < r_w[bpm - Bmin]) {
                        mainBPM = bpm;
                        BPMs = r_w[bpm - Bmin];
                }
        }
        Log.d("CalcBpm", mainBPM + "bpm" + BPMs); // mainBPM に求まった BPM,BPMs に R が格納されている

// 拍位置計算
        double theta = Math.atan2(sin_w[mainBPM - Bmin], cos_w[mainBPM - Bmin]);
        if (theta < 0) {
                theta += 2.0 * Math.PI;
        }

        int start_beat = (int) (theta / (2.0 * Math.PI) * ((double) 60 / mainBPM) * m_wavfmt.rate);

        Log.d("CalcBpm", "Beat Start : " + start_beat + " point¥n"); //  求まった拍位置
        Log.d("CalcBpm", "Beat Start : " + (double) start_beat / m_wavfmt.rate +" sec¥n"); //  拍位置を時間換算

        return new int[] { mainBPM, start_beat };
}
```

付録 2. Chroma ベクトル解析処理

```
private int FFT_SIZE = 0; // FFT_SIZE はインスタンス作成時に指定（4096 サンプル）

public CalcChroma(int FFTsize) {
	FFT_SIZE = FFTsize;
}

public double[] getChroma(WAVEFORMAT m_wavfmt, FileInputStream fis, String fname) {

	DoubleFFT_1D fft = new DoubleFFT_1D(FFT_SIZE);
	byte[] m_buf = null;
	double[] data = null;
	int i;
	int row, col;
	int point;
	double rate;

	int m_width = (m_wavfmt.data_size / 2) / FFT_SIZE;
	int m_height = FFT_SIZE / 2;

	double[] m_power = new double[m_height];// パワースペクトル(フレーム毎に更新)

	m_buf = new byte[FFT_SIZE * 2];
	data = new double[FFT_SIZE * 2];

	String[] sca = {"A","A#","B","C","C#","D","D#","E","F","F#","G","G#"};
	double[] scale = new double[12 * 7 + 1]; // A2 110Hz ～ G8 まで（while オーバー対策に+1)

	double[] AllChroma = new double[13]; // Chroma ベクトル 12 配列と全ベクトルの合計値を格納する
	long TotalChroma = 0;

	for (i = 0; i < 12; i++) {
		AllChroma[i] = 0;
	}

	// 音階の周波数の計算（scale に格納)
	scale[0] = 110; // 基準点: o2a = 110Hz
	for (i = 0; i < 7; i++) {
		if (i != 0)
			scale[i * 12] = scale[(i-1) * 12] * 2;
		for (int j = 1; j < 12; j++) {
			scale[i * 12 + j] = scale[i * 12] * Math.pow(2, (double) j / 12);
		}
	}

	// 読み込み解析処理開始
	for (col = 0; col < m_width; col++) {

		// 読み込み処理
		try {
			int len = fis.read(m_buf);
			if (len < m_buf.length) {
				Log.d("CalcChroma", "read error!" +len+":"+ m_buf.length);
			}
		} catch (IOException e) {
			e.printStackTrace(); //ここでのエラー処理はほぼ実装していない
		}

		// オリジナルデータ byte[]を double[]に変換
		for (i = 0; i < FFT_SIZE; i++) {
			data[i] = (short) ((m_buf[i * 2 + 1] * 0x100) & 0xFFFF | (m_buf[i * 2] & 0xFF));
		}

		for (i = FFT_SIZE; i < FFT_SIZE * 2; i++) {
			data[i] = 0.0;
		} // data[]の残り後半を 0 で埋める
```

```
            // フーリエ変換(FFT)の実行 (data[0]は実数,[1]は虚数 ～ data[n]は実数,[n+1]は虚数)
            fft.realForwardFull(data);

            // パワースペクトル計算
            for (row = 0; row < m_height; row++) {
                  m_power[row] = Math.sqrt(data[row * 2] * data[row * 2]
                              + data[row * 2 + 1] * data[row * 2 + 1]);
            }

            // フレーム毎の周波数毎のスペクトルをクロマベクトルに格納
            point = 0;
            for (row = 0; row < m_height; row++) {
                  rate = (double) m_wavfmt.rate / FFT_SIZE * row;
                  if (rate > 107.70833) {
                        while ((point < (12*7)) && (rate > (double)(scale[point] + scale[point+1]) / 2)) {
                              point++;
                        }
                        if (point < 12 * 7) { // 対応する音階に音量を格納する処理
                              AllChroma[point % 12] += m_power[row];
                              TotalChroma += m_power[row];
                        }
                  }
            }
      } // 計算の終了

      // 求めた Chroma ベクトル出力処理
      Log.d("CalcChroma", "TotalChromaVector(o2a~o8g)¥n");
      for (i = 0; i < 12; i++) {
            Log.d("CalcChroma", sca[i] + ":" + AllChroma[i]);
      }

      AllChroma[12] = TotalChroma; // return 用に 12 ベクトルの合計値を 13 番目に格納

      return AllChroma;
}
```

付録 3. 楽曲間類似度の算出処理

```
public void getSimilar() {
	int N = MainActivity.dbHelper.count_musicList(); // 楽曲数 N

	// 項目の変数宣言
	String cname[] = new String[N]; // 楽曲ファイル一覧（主キーとして利用）
	int[] bpm = new int[N];
	int[] beat = new int[N];
	double[][] chroma = new double[N][12];
	String artist[] = new String[N];
	String composer[] = new String[N];
	String album[] = new String[N];

	// 各項目の SQLite からの読み込み
	cname = MainActivity.dbHelper.get_AllMusicList(N);

	for (int i = 0; i < N; i++) {
		bpm[i] = MainActivity.dbHelper.get_bpm(cname[i]);
		beat[i] = MainActivity.dbHelper.get_beat(cname[i]);
		chroma[i] = MainActivity.dbHelper.get_chroma(cname[i]);
		artist[i] = MainActivity.dbHelper.get_info_artist(cname[i]);
		composer[i] = MainActivity.dbHelper.get_info_composer(cname[i]);
		album[i] = MainActivity.dbHelper.get_info_album(cname[i]);
	}

	int n = N;

	//差分を計算する為の変数
	int[] diff_b = new int[N];
	double[] diff_ch = new double[N];
	double[] diff_mix = new double[N];

// 自動プレイリスト生成処理の開始
	int[] use = new int[n];
	for (int i = 0; i < n; i++) { use[i] = 0; }

	// ランダムに開始楽曲を決定する
	int pt = (int) (Math.random() * n);
	use[pt] = 1;

	// 1 曲目のデータベースへの登録
	MainActivity.dbHelper.write_playList(0, cname[pt]);

	// 2 曲目以降の選択処理
	for (int j = 1; j < n; j++) {
		for (int i = 0; i < n; i++) {
			if (pt != i && use[i] != 1) {
				diff_b[i] = bpm_dist(bpm[pt], bpm[i]); // 下部で計算
				diff_ch[i] = chroma_dist(chroma[pt], chroma[i]); // 下部で計算

				// 重み付けして楽曲間類似度を算出
				diff_mix[i] = Math.pow(diff_b[i], 1.2) * 0.0001
					+ Math.pow(diff_ch[i], 1.3)
					+ ((artist[pt].equals(artist[i])) ? 0 : 0.000025)
					+ ((composer[pt].equals(composer[i])) ? 0 : 0.000020)
					+ ((album[pt].equals(album[i])) ? 0 : 0.000010);
			} else { //既に選択済みの場合はスキップ
				diff_b[i] = -1;
				diff_ch[i] = -1;
				diff_mix[i] = -1;
			}
		}
		// 最も類似度の高い 3 曲を取得し 1 曲をランダム選択する
		int px[] = new int[3];
		px = UtilFunc.find_min3(diff_mix, n); // diff_mix から最も値の小さい 3 曲の番号を取得
		int rand = (j < n - 2) ? (int) (Math.random() * 3) : 0;
```

```
                pt = px[rand];
                use[pt] = 1;

                //2 曲目以降のデータベースへの登録
                MainActivity.dbHelper.write_playList(j, cname[pt]);
        }
}

private int bpm_dist(int b1, int b2) {
        return Math.abs(b1 - b2);
}

private double chroma_dist(double ch1[], double ch2[]) {
        double dist = 0.0;
        for (int i = 0; i < 12; i++) {
                dist += (ch1[i] - ch2[i]) * (ch1[i] - ch2[i]);
        }
        return dist;
}
```

付録 4. 2 曲間合成部分の合成位置算出処理

```
private int rate = 0; // サンプリングレート

CalcClossfade() {
	rate = MainActivity.ms.getRate(); // 設定値(44100)を取得
}

public int[] getClossFadePoint(String cname1, String cname2) {

// 重ね合わせを行う場合と行わない場合(最終曲)での分岐
	if(cname2 != null){
		// 楽曲の読み込み準備
		File fin1 = new File(cname1);
		File fin2 = new File(cname2);
		FileInputStream fis1 = null;
		FileInputStream fis2 = null;

		try {
			fis1 = new FileInputStream(fin1);
			fis2 = new FileInputStream(fin2);
			fis2.skip(44);
		} catch (FileNotFoundException e) {
			e.printStackTrace();
		} catch (IOException e) {
			e.printStackTrace();
		}

		ReadWaveHeader head1 = new ReadWaveHeader(fis1);

		// DB から bpm と beat の取得
		int m1bpm = MainActivity.dbHelper.get_bpm(cname1);
		int m1beat = MainActivity.dbHelper.get_beat(cname1);
		int m2bpm = MainActivity.dbHelper.get_bpm(cname2);
		int m2beat = MainActivity.dbHelper.get_beat(cname2);

	// m1 の拍の一致地点の計算---------->最終的に m1eq を求める
		int m1cycle = (int) ((double) 60 / m1bpm * rate);
		int m1eq = ((head1.size() / m1cycle) * m1cycle) + m1beat;
		int m1end = head1.size();

		short[] buf1 = new short[m1cycle];

		boolean ch1 = false;
		int skip1 = 0;
		try {
			while (ch1 == false) {
				fis1.skip((m1end - skip1 * m1cycle) * 2);
				buf1 = UtilFunc.fread(fis1, m1cycle); // ファイルの読み込み処理

			// ギャップレス再生関連処理
				int i = m1cycle - 1; // m1 の拍の位置の計算
				while ((i >= 0) && (Math.abs(buf1[i]) <= MainActivity.ms.getTh1())) {
					i--;
				}

				if ((i + (m1end - skip1 * m1cycle)) > m1eq) {
					ch1 = true;
					m1eq -= m1cycle;
					skip1+=20;//20 周期減らして m1eq をずらす（重ね合わせる周期を 20 としている）
					m1eq -= m1cycle*20;
				} else {
					skip1++;
					m1eq -= m1cycle;

					fis1.close(); // 開き直し
					fis1 = new FileInputStream(fin1);
```

```
                        fis1.skip(44);
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        // m2 の一致点の計算---------->最終的に m2eq を求める

            int m2cycle = (int) ((double) 60 / m2bpm * rate);
            int m2eq = m2beat;

            short[] buf2 = new short[m2cycle];

            boolean ch2 = false;
            int skip2 = 0;
            while (ch2 == false) {
                buf2 = UtilFunc.fread(fis2, m2cycle);

            //  ギャップレス再生関連処理
                int i = 0; // m2 の拍の位置の計算
                while ((i < m2cycle)
                        && (Math.abs(buf2[i]) <= MainActivity.ms.getTh2())) {
                    i++;
                }

                if ((i + skip2 * m2cycle) < m2eq) {
                    ch2 = true;
                    skip2+=10;//10 周期増やして m2eq をずらす
                    m2eq += m2cycle*10;
                } else {
                    skip2++;
                    m2eq += m2cycle;
                }
            }
        // ----------------------------
            try {
                fis1.close();
                fis2.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

         // 2 曲の一致点とオフセットを結果として返す
            return new int[] { m1eq, m2eq, m1cycle };

//重ね合わせを行わない場合
        }else{
            FileInputStream fis1 = null;
            ReadWaveHeader head1 = null;

            try {
                fis1 = new FileInputStream(cname1);
                head1 = new ReadWaveHeader(fis1);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }

            // DB から bpm と beat の取得
            int m1bpm = MainActivity.dbHelper.get_bpm(cname1);
            int m1beat = MainActivity.dbHelper.get_beat(cname1);

            int m1cycle = (int) ((double) 60 / m1bpm * rate);
            int m1eq = ((head1.size() / m1cycle) * m1cycle) + m1beat;
            int m1end = head1.size();

            short[] buf1 = new short[m1cycle];
```

```
            boolean ch1 = false;
            int skip1 = 0;
            try {
                while (ch1 == false) {
                    fis1.skip((m1end - skip1 * m1cycle) * 2);
                    buf1 = UtilFunc.fread(fis1, m1cycle);

                // ギャップレス再生関連処理
                    int i = m1cycle - 1; // m1 の拍の位置の計算
                    while ((i >= 0)
                            && (Math.abs(buf1[i]) <= MainActivity.ms.getTh1())) {
                        i--;
                    }

                    if ((i + (m1end - skip1 * m1cycle)) > m1eq) {
                        ch1 = true;
                    } else {
                        skip1++;
                        m1eq -= m1cycle;

                        fis1.close(); // 開き直し
                        fis1 = new FileInputStream(cname1);
                        fis1.skip(44);
                    }
                }
            } catch (IOException e) {
                e.printStackTrace();
            }

        // ------------------------------

            try {
                fis1.close();
            } catch (IOException e) {
                e.printStackTrace();
            }

         // 2 曲の一致点とオフセットを結果として返す
            return new int[] { m1eq, 0, m1cycle };
        }
}
```

付録 5．2 曲間合成部分の合成波形出力処理

```
public class LookAheadThread extends Thread {

	private int startPoint;
	private int endPoint;
	private int beatPoint;
	private int m1cycle;
	private String cname;
	private String cname2;
	private String cname3 = null;

	private short[] out_buf;
	private int[] ret_point;

	private CalcClossfade cc;

	public LookAheadThread(int var3, int var4, int var5, int var6, String cn1, String cn2, String cn3) {
		cc = new CalcClossfade();

		startPoint = var3;
		endPoint = var4;
		beatPoint = var5;
		m1cycle = var6;
		cname = cn1;
		cname2 = cn2;
		cname3 = cn3;
	}

	public void run() {
		out_buf = lookAhead(startPoint, endPoint, beatPoint, m1cycle);
	}

	private short[] lookAhead(int startPoint, int endPoint, int beatPoint, int m1cycle) {
		FileInputStream fis1_2 = null; // 現在再生中の曲の最後を取得
		FileInputStream fis2 = null; // 次に再生予定の曲の先頭を取得

		short[] buf_out = null;

		if(cname2 != null){
			try {
				fis1_2 = new FileInputStream(cname);
				fis2 = new FileInputStream(cname2);

				fis1_2.skip(startPoint); //オフセットまで飛ばす
				fis2.skip(44); // ヘッダ分飛ばす
				beatPoint -= 44;

				byte[] buf = new byte[44100];
				while (buf.length < endPoint - m1cycle*20) { //20 周期の重ね合わせとする
					fis1_2.skip(buf.length);
					endPoint -= buf.length;
				}

				short[] buf1 = UtilFunc.fread(fis1_2, fis1_2.available() / 2);
				short[] buf2 = UtilFunc.fread(fis2, beatPoint / 2);
				buf_out = new short[endPoint / 2];
				int n = buf_out.length;
				for (int i = 0; i < n; i++) {
				// クロスフェード処理
					short tmp1 = (i < buf1.length) ? buf1[i] : 0;
					short tmp2 = (i < (endPoint - beatPoint) / 2) ? 0 : buf2[i - (endPoint - beatPoint) / 2];
					buf_out[i] = (short) (tmp1 * ((double) (n - i) / n) + tmp2 * ((double) i / n));
				}

			} catch (FileNotFoundException e) {
				e.printStackTrace();
```

```
                } catch (IOException e) {
                        e.printStackTrace();
                }

                //次とその次に再生予定の曲で交差点を計算
                ret_point = cc.getClossFadePoint(cname2, cname3); //  付録 5 の関数を呼び出し
                Log.d("StreamWrite", "GetClossFadePoint finish!");


        //クロスフェードしない場合
        } else {
                try {
                        fis1_2 = new FileInputStream(cname);

                        fis1_2.skip(startPoint);
                        beatPoint -= 44;

                        endPoint -= startPoint;

                        byte[] buf = new byte[44100];
                        while (buf.length < endPoint - m1cycle) {
                                fis1_2.skip(buf.length);
                                endPoint -= buf.length;
                        }

                        short[] buf1 = UtilFunc.fread(fis1_2, fis1_2.available() / 2);
                        buf_out = new short[endPoint / 2];
                        int n = buf_out.length;
                        for (int i = 0; i < n; i++) {
                                short tmp1 = (i < buf1.length) ? buf1[i] : 0;
                                buf_out[i] = (short) (tmp1 * ((double) (n - i) / n));
                        }

                } catch (FileNotFoundException e) {
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                ret_point = new int[] { 0, 0, 0 };
        }
        return buf_out;
    }

    public short[] get_buf() {
        return out_buf;
    }

    public byte[] get_buf_b(int offset) {
        byte[] buf = new byte[44100];
        int i=0;
        while(i<22050 && (i+offset)<out_buf.length){
                buf[i] = (byte) ( out_buf[i+offset]& 0xFF    /2);
                buf[i+1] = (byte) ( out_buf[i+offset]& 0xFF00    /2);
                i++;
        }
        return buf;
    }

    public int get_buflen() {
        return out_buf.length;
    }

    public int[] get_ret() {
        return ret_point;
    }
}
```