1 はじめに

この記事は、統計的声質変換について基礎から解説し、Python による実装を示しながら、固有声という概念を用いた多対多の声質変換まで辿り着くことを目的としている。ただし、混合ガウスモデルや EM アルゴリズムなどの機械学習や統計処理に関する部分や、音声特徴量の抽出など音声分析や音声合成に関わる部分には深く立ち入らず、次章で軽く触れるのみとする。

また、この記事で用いる Python コードはすべて Python 2.7.10 で動作を確認しており、ライブラリについては以下の通りである。

- matplotlib: 1.5.0
- numpy: 1.9.2
- scikit-learn: 0.17
- scipy: 0.15.1

2 統計的声質変換の基礎知識

2.1 統計的声質変換とは

統計的声質変換とは、人間の声から抽出された特徴量を統計的手法によって変換することによって、ある人の音声データを元に、別の人が同じ内容を話したかのような音声データを生成する仕組みのことである。基本的には、変換元となる人と変換先となる人が同じ内容を話しているデータ（以降、「パラレルデータ」という）を用いて、統計的モデルを学習することによって、変換を行っている。

ただし、この統計的モデルが扱えるのはあくまで数値データであり、音声データをそのまま学習に用いることはできない[1]。

そこで、音声データを少ないパラメータで表すために、音声分析による特徴量抽出が必要になる。もちろん、統計的モデルによって変換されたパラメータから、音声合成によって音声データを復元する作業も必要になる。

つまり、統計的声質変換は、以下のような仕組みとなっている。

1 厳密に言えば、音声データをそのまま数値化して扱うことはできるものの、データ量が大きすぎて学習に時間が掛かり過ぎる。

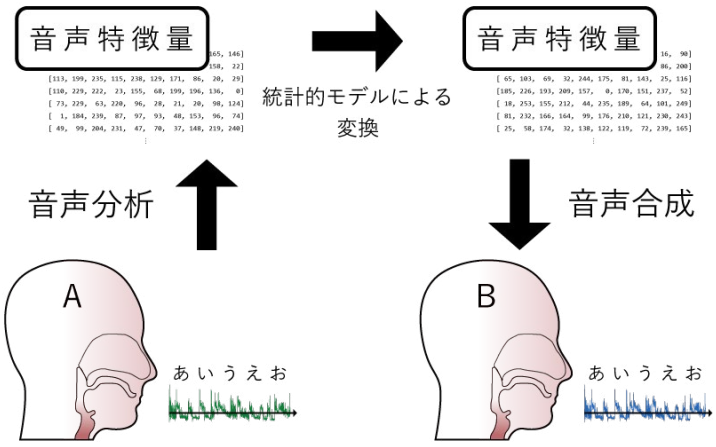

図2.1　統計的声質変換の仕組み

2.2　音声分析による特徴量抽出

音声分析のベースには、ソースフィルタモデルと呼ばれる概念が取り入れられている。ソースフィルタモデルにおいては、声帯の振動によって生み出された音を声道や口腔の形状によって変化させることで発声しているという考え方に基づき、声帯からの音源（ソース）を表すパラメータと、声道や口腔での変化（フィルタ）を表すパラメータを用いて音声の特徴を表す。この際、ソースのパラメータは声の高さや太さ、フィルタのパラメータは「あ」「い」といった発声内容や声の個人性（人それぞれの特徴）などを表すこととなる。つまり、このフィルタを表すパラメータを統計的モデルによって変換することによって声質変換ができる。

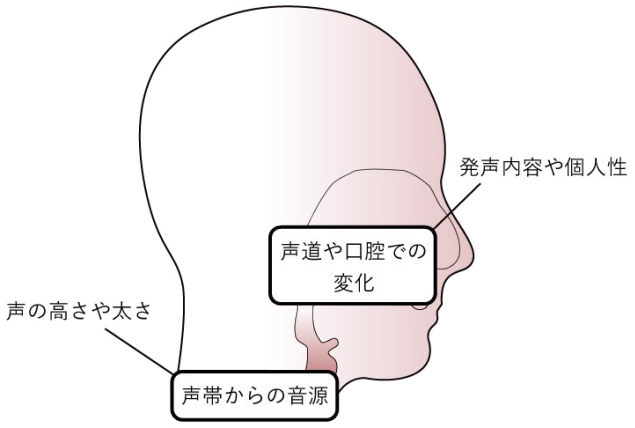

図2.2　ソースフィルタモデル

今回は、このパラメータの抽出については、TANDEM-STRAIGHT[1]というシステムを利用する。これは、山梨大学の森勢助教が公開している[2]音声分析・合成システムであり、音声データを分析し、ソースを表すパラメータである基本周波数、フィルタを表すパラメータであるスペクトル包絡、そして、音声のかすれや雑音を表すパラメータである非周期成分という 3 つのパラメータを抽出することができる。もちろん、この 3 つのパラメータから音声データを合成することも可能である。

さらに、メル周波数ケプストラム係数（以降、「MFCC」という）というものを導入する。これは、スペクトル包絡をより人間の知覚に沿うような形で抽出した特徴量であり、具体的には、低い音に対してはその音程の細かな違いに気づくが、高い音になるほど音程の違いに気づきにくくなるという人間の特性を利用している。MFCC は、スペクトル包絡に対して低い周波数帯がより強調されるようなフィルタ（メルバンクフィルタという）を適用し、離散コサイン変換を行うことによって得られる。

2.3 EM アルゴリズム

統計的声質変換には、後述する混合ガウスモデルがよく使われるのだが、それだけに限らず様々な場面で使われるのが EM アルゴリズムである。EM アルゴリズムは、確率モデルのパラメータを推定するときに使われる手法で、ある観測データ $\boldsymbol{X}$、その観測データと潜在変数 $\boldsymbol{Z}$ との同時分布 $P(\boldsymbol{X}, \boldsymbol{Y}|\boldsymbol{\theta})$ が与えられた時に、$P(\boldsymbol{X}|\boldsymbol{\theta}\}$ を最大化するようなパラメータ $\boldsymbol{\theta}$ を推定することができる。式で表すと、以下の通りとなる。

$$\begin{aligned}\hat{\boldsymbol{\theta}} &= \underset{\boldsymbol{\theta}}{argmax}\ P(\boldsymbol{X}|\boldsymbol{\theta}) \\ &= \underset{\boldsymbol{\theta}}{argmax} \prod_{all\ \boldsymbol{Z}} P(\boldsymbol{X}, \boldsymbol{Z}|\boldsymbol{\theta})\end{aligned}$$

ここで、Q 関数と呼ばれる関数を導入する。

$$\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{old}) = \sum_{all\ \boldsymbol{Z}} P(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{\theta}^{old}) \log P(\boldsymbol{X}, \boldsymbol{Z}|\boldsymbol{\theta})$$

すると、以下のステップによって、$\boldsymbol{\theta}$ を推定できる。

- パラメータの初期値 $\boldsymbol{\theta}^{old}$ を適当に定める
- Eステップ: $P(\boldsymbol{Z}|\boldsymbol{X}, \boldsymbol{\theta}^{old})$ を計算する。
- Mステップ: Eステップで得られた値を元に $\boldsymbol{\theta}^{new} = \underset{\boldsymbol{\theta}}{argmax}\mathcal{Q}(\boldsymbol{\theta}, \boldsymbol{\theta}^{new})$ を計算する。
- $\boldsymbol{\theta}$ が収束するまで $\boldsymbol{\theta}^{old} \leftarrow \boldsymbol{\theta}^{new}$ としてEステップとMステップを繰り返す。

なぜこの方法によって $\boldsymbol{\theta}$ の最尤推定ができるのかという詳しい説明は専門書に譲るとして、図

2 http://ml.cs.yamanashi.ac.jp/straight/

を用いて簡単に説明をしておく。以下の図は、$\boldsymbol{\theta}$と対数尤度関数$\log P(\boldsymbol{X}|\boldsymbol{\theta})$のグラフである。

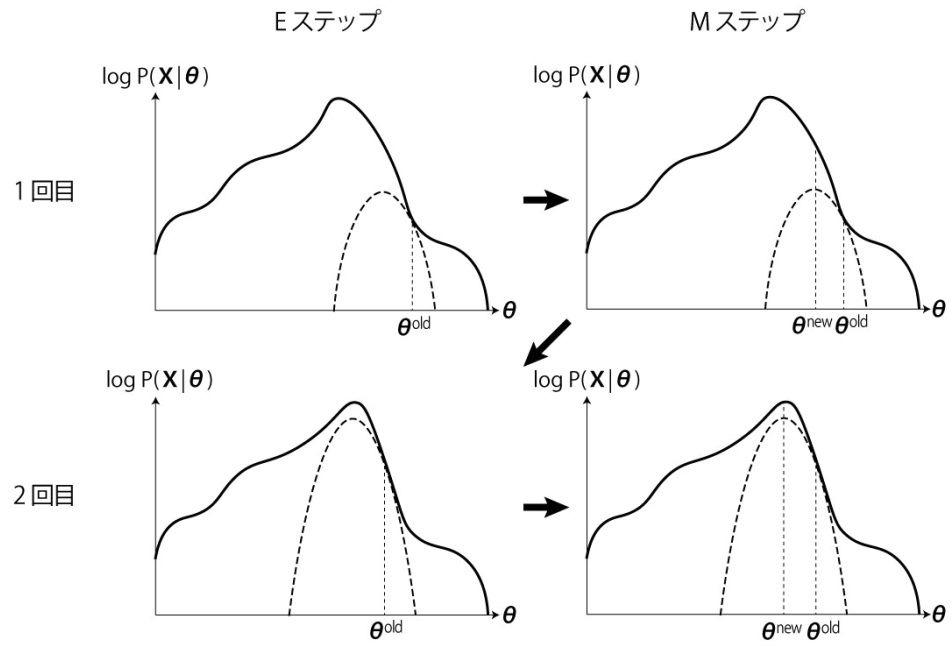

図2.3　EMアルゴリズムのイメージ

実は、E ステップは対数尤度関数の $\boldsymbol{\theta}^{old}$ における下界を求めるというのと対応し、M ステップはその下界を最大化するパラメータ $\boldsymbol{\theta}^{new}$ を求めるというのと対応している。図を見れば、それを繰り返すことによって対数尤度関数を最大化するようにパラメータが動いていることが分かるだろう。

2.4　混合ガウスモデル

混合ガウスモデル (以降、「GMM」という) は、複数のガウス分布を組み合わせることによって表されるモデルである。扱うデータをD次元のベクトル$\boldsymbol{x}$としたときの定義は以下の通りである。

$$P(\boldsymbol{x}|\boldsymbol{\lambda}) = \sum_{m=1}^{M} w_m \mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$$

ここで、$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m)$は、平均ベクトルが$\boldsymbol{\mu}_m$、分散共分散行列が$\boldsymbol{\Sigma}_m$のガウス分布である。

$$\mathcal{N}(\boldsymbol{x}; \boldsymbol{\mu}_m, \boldsymbol{\Sigma}_m) = \frac{1}{(2\pi)^{\frac{d}{2}}\boldsymbol{\Sigma}_m} \exp(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_m)^\top \boldsymbol{\Sigma}_m^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_m))$$

すなわち、GMM は M 個のガウス分布を線形結合したものであり、確率密度関数となるために以下の制約を持つ。

$$w_1, ..., w_m \geq 0, \sum_{m=1}^{M} w_m = 1$$

つまり、GMM のパラメータは、それぞれのガウスモデルの重みと平均ベクトル、分散共分散行列からなり、まとめて $\boldsymbol{\lambda}$ として表される。

混合ガウスモデルの最尤推定

GMM の学習は、学習データのそれぞれに対して推定される確率の積が最大となるようなパラメータを見つけることである。つまり、N個の学習データに対し、尤度関数 $L(\boldsymbol{\lambda})$ を用いて以下のように定式化される。

$$L(\boldsymbol{\lambda}) := \prod_{n=1}^{N} P(\boldsymbol{x}_n, \boldsymbol{\lambda})$$

$$\hat{\boldsymbol{\lambda}} := \underset{\boldsymbol{\lambda}}{argmax}\ L(\boldsymbol{\lambda}) \text{ subject to} \begin{cases} w_1, ..., w_m \geq 0 \\ \sum_{m=1}^{M} w_m = 1 \end{cases}$$

このとき、最尤推定量 $\hat{\boldsymbol{\lambda}}$ は次式を満たす。

$$\left. \frac{\partial}{\partial \boldsymbol{\lambda}} L(\boldsymbol{\lambda}) \right|_{\boldsymbol{\lambda}=\hat{\boldsymbol{\lambda}}} = 0$$

これは、重み、平均ベクトル、分散共分散行列について、それぞれ以下を満たす。

$$\hat{w}_m = \frac{1}{N} \sum_{n=1}^{N} \hat{\eta}_{n,m}$$

$$\hat{\boldsymbol{\mu}}_m = \frac{\sum_{n=1}^{N} \hat{\eta}_{n,m} \boldsymbol{x}_n}{\sum_{n=1}^{N} \hat{\eta}_{n,m}}$$

$$\hat{\boldsymbol{\Sigma}}_m = \frac{\sum_{n=1}^{N} \hat{\eta}_{n,m} (\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_m)^\top (\boldsymbol{x}_n - \hat{\boldsymbol{\mu}}_m)}{d \sum_{n=1}^{N} \hat{\eta}_{n,m}}$$

ただし、$\hat{\eta}_{n,m}$ は次式の通りである。

$$\hat{\eta}_{n,m} := \frac{\hat{w}_m \mathcal{N}(x_n; \hat{\boldsymbol{\mu}}_m, \hat{\boldsymbol{\Sigma}}_m)}{\sum_{m'=1}^{M} \hat{w}_m \mathcal{N}(x_n; \hat{\boldsymbol{\mu}}_{m'}, \hat{\boldsymbol{\Sigma}}_{m'})}$$

これは、$\hat{\eta}_{n,m}$ の定義にそれぞれのパラメータが用いられているため、パラメータを解析的に求

めることは難しいので、EM アルゴリズムを用いることで推定することができる。この場合は、各ステップは以下のようになる。

- パラメータの初期値 $\hat{w}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}$ を適当に定める。
- Eステップ: 現在のパラメータから $\hat{\eta}_{n,m}$ を計算する。
- Mステップ: 現在の $\hat{\eta}_{n,m}$ から $\hat{w}, \hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}$ を計算する。
- 収束するまでEステップとMステップを繰り返す。

これにより、学習データに対する尤度を最大化するようなパラメータを推定することができる。その他、EM アルゴリズムの導出や GMM についての具体的な仕組みなどは、C.M. ビショップ著「パターン認識と機械学習」[2]や杉山 将著「統計的機械学習」[3]などを参照されたい。

3 声質変換のための様々なモジュールの実装

この章では、声質変換のために必要となる様々なモジュールの実装と簡単な解説を行う。

3.1 STF ファイルの読み書き

声質変換を行うためには、まず TANDEM-STRAIGHT で抽出された特徴量を Python で扱える形式に変換しなければならない。しかし、Python から TANDEM-STRAIGHT をライブラリとして呼び出し、結果を受け取るのはかなり煩雑なので、TANDEM-STRAIGHT の音声データから抽出した特徴量を STF ファイルという独自形式で保存する機能を利用する。

まず、音声データを STF ファイルに変換するには、配布されている TANDEM-STRAIGHT ライブラリ内の GenerateSTF という実行ファイルを利用できる。そうして得られる STF ファイルを、Python で扱いやすいようデータを読み込んで、numpy の配列に変換するようなライブラリを実装する。

STF ファイルについての具体的な仕様は配布されているライブラリ内のドキュメントに詳しく記載されているが、簡単に説明すると、1 つの STF ファイルは複数のチャンクからなり、それぞれのチャンクは格納しているデータの種類を表すチャンク名、チャンクサイズ、そしてデータから構成されている。チャンクのうち、特徴量に直接関係のない制御用のものは以下の 3 つである。

- STFT: エンディアンやサンプリング周波数などがヘッダ情報を表すチャンク
- CHKL: そのSTFファイルに含まれるチャンクのリストを表すチャンク
- NXFL: 複数ファイルに亘る場合に用いられるチャンク

ただし、GenerateSTF で得られる STF ファイルでは NXFL チャンクは用いられないので、STFT と CHKL のみに対して処理を行えばよい。また、特徴量を表すチャンクはいくつかあるが、以下の 3 つのチャンクが存在していれば合成が可能なので、これら以外は無視してもよい。

- F0: 音声データの基本周波数
- SPEC: 音声データのスペクトル包絡
- APSG: 音声データの非周期成分をシグモイド関数のパラメータとして表したもの

これを Python で実装したのが、以下のソースコードである。

リスト3.1　stf.py

```
 1: #!/usr/bin/env python
 2:
 3: import numpy
 4: import os
 5: import struct
 6: import sys
 7:
 8: class STF:
 9:     def __init__(self, filename = None):
10:         self.endian = '>'
11:         self.chunks = ['APSG', 'F0  ', 'SPEC']
12:
13:     def loadfile(self, filename):
14:         with open(filename, 'rb') as stf_file:
15:             self.load(stf_file)
16:
17:     def load(self, stf_file):
18:         filesize = os.fstat(stf_file.fileno()).st_size
19:
20:         while stf_file.tell() < filesize:
21:             chunk = stf_file.read(4)
22:
23:             if chunk == 'STRT':
24:                 if stf_file.read(2) == '\xff\xfe':
25:                     self.endian = '<'
26:                 chunk_size, self.version, self.channel, self.frequency \
27:                     = struct.unpack(self.endian + 'IHHI', stf_file.read(12))
28:             else:
29:                 chunk_size, = struct.unpack(self.endian + 'I', stf_file.read(4))
30:
31:                 if chunk == 'CHKL' or chunk == 'NXFL':
32:                     data = stf_file.read(chunk_size)
33:                     if chunk == 'CHKL':
34:                         self.chunks += [data[i: i + 4] \
35:                             for i in range(0, chunk_size, 4) \
36:                             if data[i: i + 4] not in self.chunks]
37:                 else:
38:                     self.shift_length, frame_count, argument, \
39:                     self.bit_size, self.weight, data_size \
40:                         = struct.unpack(self.endian + 'dIIHdI', stf_file.read(30))
41:                     data = stf_file.read(data_size)
42:
43:                     element = data_size / (self.bit_size / 8)
44:                     matrix = numpy.fromstring(data, count = element)
45:
46:                     for c in self.chunks:
47:                         if chunk == c:
```

```
48:                             if element / frame_count == 1:
49:                                 self.__dict__[c.strip()] = matrix
50:                             else:
51:                                 self.__dict__[c.strip()] \
52:                                     = matrix.reshape( \
53:                                         (frame_count, element / frame_count))
54:                             break
55:
56:         for c in self.chunks:
57:             if c.strip() not in self.__dict__:
58:                 self.__dict__[c.strip()] = None
59:
60:     def savefile(self, filename):
61:         with open(filename, 'wb') as stf_file:
62:             self.save(stf_file)
63:
64:     def save(self, stf_file):
65:         stf_file.write('STRT')
66:         if self.endian == '>':
67:             stf_file.write('\xfe\xff')
68:         elif self.endian == '<':
69:             stf_file.write('\xff\xfe')
70:         stf_file.write(struct.pack(self.endian + 'IHHI', 8, \
71:             self.version, self.channel, self.frequency))
72:
73:         stf_file.write('CHKL')
74:         stf_file.write(struct.pack(self.endian + 'I', \
75:             len(''.join(self.chunks))) + ''.join(self.chunks))
76:
77:         for c in self.chunks:
78:             if self.__dict__[c.strip()] is None:
79:                 continue
80:
81:             matrix = self.__dict__[c.strip()]
82:             if len(matrix.shape) == 1:
83:                 argument = 1
84:             else:
85:                 argument = matrix.shape[1]
86:             data_size = matrix.shape[0] * argument * 8
87:
88:             header = struct.pack(self.endian + 'dIIHdI', self.shift_length, \
89:                 matrix.shape[0], argument, self.bit_size, self.weight, data_size)
90:             stf_file.write(c + \
91:                 struct.pack(self.endian + 'I', len(header) + data_size) + header)
92:
93:             for i in xrange(matrix.shape[0]):
94:                 if argument == 1:
95:                     stf_file.write(struct.pack(self.endian + 'd', matrix[i]))
96:                 else:
97:                     for j in xrange(matrix.shape[1]):
98:                         stf_file.write(struct.pack(self.endian + 'd', \
99:                                                    matrix[i, j]))
100:
101: if __name__ == '__main__':
102:     if len(sys.argv) < 2:
103:         print 'Usage: %s <stf_file>' % sys.argv[0]
104:         sys.exit()
105:
106:     stf = STF()
107:     stf.loadfile(sys.argv[1])
108:     print stf.F0
```

基本的に、struct モジュールを用いてデータの読み込み・書き込みを行っている。STF クラスの self.chunks に含まれているチャンクは、インスタンス変数として numpy の配列に変換されるようになっている。

3.2 MFCC の抽出

次に、STF ファイルから読み込んだスペクトル包絡から MFCC を抽出するライブラリを実装する。また、声質変換後の音声合成のために、MFCC からスペクトル包絡を復元する機能も実装する。なお、実装にあたっては、「人工知能に関する断創録 [4]」を参考にした。

まず、周波数を MFCC の算出に用いるメル尺度という音高の知覚的尺度に変換する関数と、その逆の処理をする関数を用意する。hz2mel によって周波数をメル尺度に、mel2hz によってメル尺度を周波数に変換する。

周波数とメル尺度の変換

```
    def hz2mel(self, f):
        return 1127.01048 * numpy.log(f / 700.0 + 1.0)

    def mel2hz(self, m):
        return 700.0 * (numpy.exp(m / 1127.01048) - 1.0)
```

次に、メルフィルタバンクを計算する。これは、メル尺度上で等間隔にならぶバンドパスフィルタを並べたものであり、このフィルタをスペクトル包絡に適用することで人間が知覚しにくい領域の重みが小さくなる。

メルフィルタバンクの導出

```
    def melFilterBank(self):
        # サンプリング周波数の半分（ナイキスト周波数）までを対象とする
        fmax = self.frequency / 2
        melmax = self.hz2mel(fmax)

        # 周波数に合わせて、サンプル数の半分の標本数で計算する
        nmax = self.nfft / 2
        df = self.frequency / self.nfft

        # フィルタごとの中心となるメル尺度を計算する
        dmel = melmax / (self.channels + 1)
        melcenters = numpy.arange(1, self.channels + 1) * dmel
        fcenters = self.mel2hz(melcenters)

        # それぞれの標本が対象とする周波数の範囲を計算する
        indexcenter = numpy.round(fcenters / df)
        indexstart = numpy.hstack(([0], indexcenter[0: self.channels - 1]))
        indexstop = numpy.hstack((indexcenter[1: self.channels], [nmax]))

        # フィルタごとに indexstart を始点、indexcenter を頂点、
        # indexstop を終点とする三角形のグラフを描くように計算する
        filterbank = numpy.zeros((self.channels, nmax))
        for c in numpy.arange(0, self.channels):
            increment = 1.0 / (indexcenter[c] - indexstart[c])
            for i in numpy.arange(indexstart[c], indexcenter[c]):
                filterbank[c, i] = (i - indexstart[c]) * increment
            decrement = 1.0 / (indexstop[c] - indexcenter[c])
            for i in numpy.arange(indexcenter[c], indexstop[c]):
                filterbank[c, i] = 1.0 - ((i - indexcenter[c]) * decrement)
            filterbank[c] /= (indexstop[c] - indexstart[c]) / 2

        return filterbank, fcenters
```

ここで、self.channels はメルフィルタバンクに用いられるバンドパスフィルタの数を示しており、後に出てくる MFCC の次元数と同じか、より大きくする必要がある。

このメルフィルタバンクをスペクトル包絡に適用した後に、離散コサイン変換によって得られる係数が MFCC である。つまり、MFCC を得る関数は以下のようになる。

MFCCの導出

```
    def mfcc(self, spectrum):
        # スペクトル包絡として負の値が与えられた場合は、0 として扱う
        spectrum = numpy.maximum(numpy.zeros(spectrum.shape), spectrum)
        # スペクトル包絡とメルフィルタバンクの積の対数を取る
        mspectrum = numpy.log10(numpy.dot(spectrum, self.filterbank.transpose()))
        # scipy を用いて離散コサイン変換をする
        return scipy.fftpack.dct(mspectrum, norm = 'ortho')[:self.dimension]
```

ここで、self.dimension は離散コサイン変換の結果のうち低次の係数から何次元だけ用いるかを示しており、一般には 16 次元の係数を用いることが多いが、より精度を求める場合には 32 次元や 64 次元と設定することもある。

また、MFCC からスペクトル包絡への逆変換は、離散コサイン変換の逆変換を用いて、以下のように実装できる。この実装においては、逆変換の後に scipy.interpolate を用いてスプライン補間をしている。

MFCCからのスペクトル包絡の導出

```
    def imfcc(self, mfcc):
        # MFCC の削られた部分に 0 を代入した上で、逆離散コサイン変換をする
        mfcc = numpy.hstack([mfcc, [0] * (self.channels - self.dimension)])
        mspectrum = scipy.fftpack.idct(mfcc, norm = 'ortho')
        # 得られた離散的な値をスプライン補間によって連続的にする
        tck = scipy.interpolate.splrep(self.fcenters, numpy.power(10, mspectrum))
        return scipy.interpolate.splev(self.fscale, tck)
```

そして、今後のために動的変化量を求める関数を実装しておく。これは、あるフレームの前後数フレームでの MFCC の変化量を微分したもの（回帰係数）であり、ここでは簡単のため、1 つ前のフレームと 1 つ後のフレームの差を 2 で割ったものを用いることとする。

MFCCの動的変化量の導出

```
    def delta(self, mfcc):
        # データの開始部と終了部は同じデータが続いているものとする
        mfcc = numpy.concatenate([[mfcc[0]], mfcc, [mfcc[-1]]])

        delta = None
        for i in xrange(1, mfcc.shape[0] - 1):
            # 前後のフレームの差を 2 で割ったものを動的変化量とする
            slope = (mfcc[i + 1] - mfcc[i - 1]) / 2
            if delta is None:
                delta = slope
            else:
                delta = numpy.vstack([delta, slope])

        return delta
```

これまでのソースコードを全部組み合わせた mfcc.py は以下の通りである。なお、このモジュールをコマンドライン引数に STF ファイルを与えて実行すると、MFCC の出力及び元データと MFCC からの逆変換で得られるデータの波形の違いを確認することができる。

リスト3.2　mfcc.py

```
 1: #!/usr/bin/env python
 2:
 3: import numpy
 4: import scipy.fftpack
 5: import scipy.interpolate
 6: import scipy.linalg
 7: import sys
 8:
 9: from stf import STF
10:
11: class MFCC:
12:     '''
13:     MFCC computation from spectrum information
14:
15:     Reference
16:     ---------
17:      - http://aidiary.hatenablog.com/entry/20120225/1330179868
18:     '''
19:
20:     def __init__(self, nfft, frequency, dimension = 16, channels = 20):
21:         self.nfft = nfft
22:         self.frequency = frequency
23:         self.dimension = dimension
24:         self.channels = channels
25:
26:         self.fscale = \
27:             numpy.fft.fftfreq(self.nfft, d = 1.0 / self.frequency)[: self.nfft / 2]
28:         self.filterbank, self.fcenters = self.melFilterBank()
29:
30:     def hz2mel(self, f):
31:         return 1127.01048 * numpy.log(f / 700.0 + 1.0)
32:
33:     def mel2hz(self, m):
34:         return 700.0 * (numpy.exp(m / 1127.01048) - 1.0)
35:
36:     def melFilterBank(self):
37:         <省略>
38:
39:     def mfcc(self, spectrum):
40:         <省略>
41:
42:     def delta(self, mfcc):
43:         <省略>
44:
45:     def imfcc(self, mfcc):
46:         <省略>
47:
48: if __name__ == '__main__':
49:     if len(sys.argv) < 2:
50:         print 'Usage: %s <stf_file>' % sys.argv[0]
51:         sys.exit()
52:
53:     stf = STF()
54:     stf.loadfile(sys.argv[1])
55:
56:     mfcc = MFCC(stf.SPEC.shape[1] * 2, stf.frequency)
57:     res = mfcc.mfcc(stf.SPEC[stf.SPEC.shape[0] / 5])
58:     spec = mfcc.imfcc(res)
59:
60:     print res
61:
62:     import pylab
63:
64:     pylab.subplot(211)
65:     pylab.plot(stf.SPEC[stf.SPEC.shape[0] / 5])
66:     pylab.ylim(0, 1.2)
```

```
67:     pylab.subplot(212)
68:     pylab.plot(spec)
69:     pylab.ylim(0, 1.2)
70:     pylab.show()
```

以下は、同じ音声データを MFCC に変換し、逆変換した結果を、16 次元まで求めたものと、64 次元まで求めたもので比べたものである。16 次元でも低周波数ではある程度復元できているものの、64 次元の方がより元のスペクトル包絡に近くなっていることがわかる。また、64 次元のものでも、高周波数帯になるにつれて線が滑らかになってしまっているのがわかるだろう。

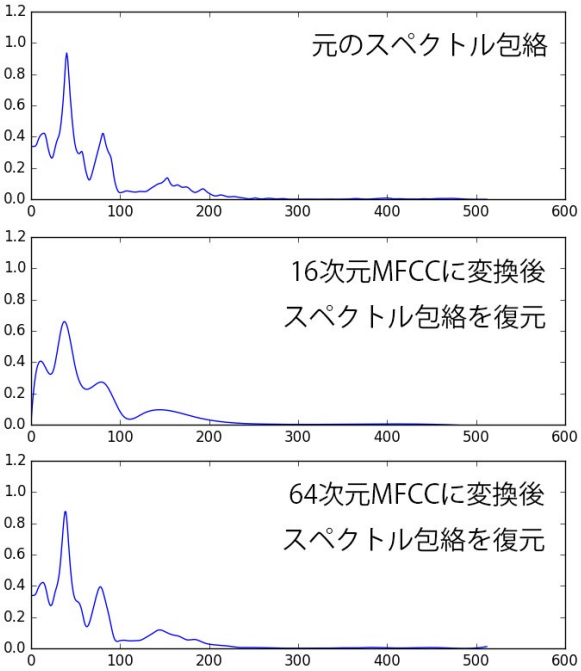

図3.1　MFCCの次元数による精度の違い

3.3　動的時間伸縮の実装

GMM を使った学習処理を行う前に、もう 1 つ実装しておくべき機能が、動的時間伸縮（Dynamic Time Warping、以下、「DTW」という）である。これは、学習に使うパラレルデータを作るために欠かせない処理であり、同じ発話内容の音声データであっても話すスピードや間の長さの違いによってフレームごとでは対応が取れていない場合が多いので、DP マッチングに基づいてデータを伸縮させるという処理である。つまり、以下の図のように、同じ音を発している部分の対応を見つけ、両方が同じフレーム数になるように伸縮させるという処理である。

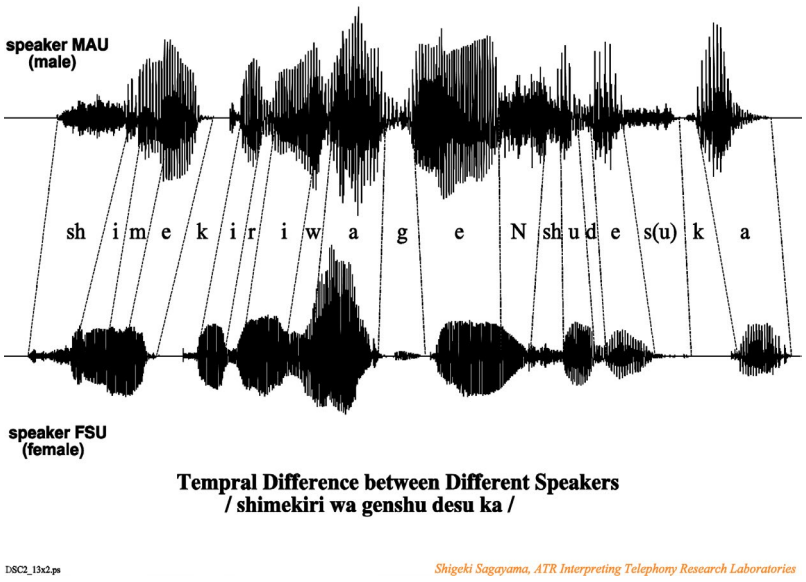

図3.2　話者による波形の違い([5]より引用)

そのためには、まずDPマッチングを実装する必要がある。DPマッチングは、その名の通り動的計画法を用いた手法で、以下のように説明できる。

- 1つ目のデータを $a_1, a_2, ..., a_N$ 、2つ目のデータを $b_1, b_2, ..., b_M$ として、N×Mのグリッドグラフを用意する。
- 頂点 (i, j) に対し、a_i と b_j の類似度をコストとして設定する。
- 頂点 $(1, 1)$ から頂点 (N, M) への最小コストのパスを検索する。
- 最小コストの値からパスを逆順に求めていき、頂点 (i, j) を通過したならば、a_i と b_j が対応していることがわかる。

図で表すと以下の様なイメージとなる。同じ内容の発話をしているフレームに対応する頂点を、赤いパスが通過していることがわかる。

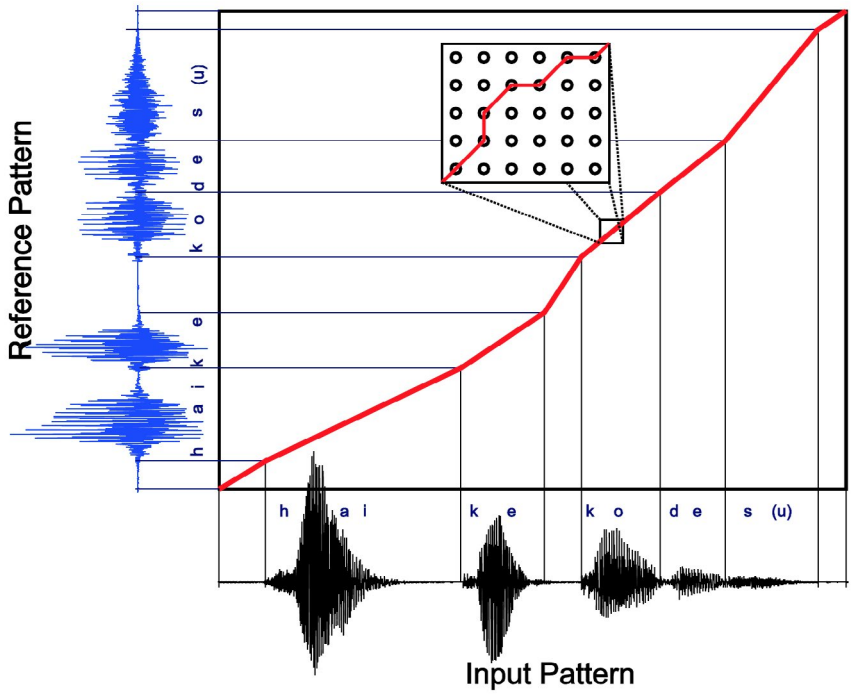

図3.3　DPマッチングのイメージ([5]より引用)

Python での実装は以下の通りとなる。なお、self.distance はフレーム間の類似度を算出する関数である。

DPマッチング

```
    def dtw(self):
        M, N = len(self.source), len(self.target)
        cost = sys.maxint * numpy.ones((M, N))

        # グリッドグラフの 1 行目、1 列目だけ先に処理しておく
        cost[0, 0] = self.distance(self.source[0], self.target[0])
        for i in range(1, M):
            cost[i, 0] = cost[i - 1, 0] + self.distance(self.source[i], self.target[0])
        for i in range(1, N):
            cost[0, i] = cost[0, i - 1] + self.distance(self.source[0], self.target[i])

        # 各頂点までの最短パスの長さを計算する
        for i in range(1, M):
            # 各フレームの前後 self.window フレームだけを参照する
            for j in range(max(1, i - self.window), min(N, i + self.window)):
                cost[i, j] = \
                        min(cost[i - 1, j - 1], cost[i, j - 1], cost[i - 1, j]) + \
                        self.distance(self.source[i], self.target[j])

        m, n = M - 1, N - 1
        self.path = []

        # 最短パスの経路を逆順に求めていく
        while (m, n) != (0, 0):
            self.path.append((m, n))
            m, n = min((m - 1, n), (m, n - 1), (m - 1, n - 1), \
                                    key = lambda x: cost[x[0], x[1]])
            if m < 0 or n < 0:
                break
```

```
        self.path.append((0, 0))
```

ここで注意すべき点は、self.window による探索範囲の制限である。これは、あるフレームに対して比較対象を前後 self.window フレームのみに限定するというもので、探索時間と、同じ発話内容が繰り返し現れた時に異なる繰り返しを参照してしまう可能性の低減を目的としている。これまで使ってみた限りでは、self.window を 2 つのデータのフレーム数の差の 2 倍程度にちょうど設定するくらいがよいであろう。

次に、DP マッチングの結果に応じてデータを伸縮させる処理を実装する。この際、単に対応するフレームを用いるのではなく、1 つ前のフレームが対応するフレームとの間で最も類似度が高いものを選ぶようにしている。つまり、DP マッチングの説明の時と同じ記号を用いると、パスが (i, j) の次に $(i+1, k)$ を通るとすると、a_{i+1} に対応するフレームは、$b_j, b_{j+1}, ..., b_k$ の中で最も a_{i+1} に類似度が高いフレームを選ぶ。

DPマッチングによる伸縮

```
    def align(self, data, reverse = False):
        # reverse = True の時は、target のフレーム数に合わせるようにする
        if reverse:
            path = [(t[1], t[0]) for t in self.path]
            source = self.target
            target = self.source
        else:
            path = self.path
            source = self.source
            target = self.target

        path.sort(key = lambda x: (x[1], x[0]))

        shape = tuple([path[-1][1] + 1] + list(data.shape[1:]))
        alignment = numpy.ndarray(shape)

        idx = 0
        frame = 0
        candicates = []

        while idx < len(path) and frame < target.shape[0]:
            if path[idx][1] > frame:
                # 候補となっているフレームから最も類似度が高いフレームを選ぶ
                candicates.sort(key = lambda x: \
                                self.distance(source[x], target[frame]))
                alignment[frame] = data[candicates[0]]

                candicates = [path[idx][0]]
                frame += 1
            else:
                candicates.append(path[idx][0])
                idx += 1

        if frame < target.shape[0]:
            candicates.sort(key = lambda x: self.distance(source[x], target[frame]))
            alignment[frame] = data[candicates[0]]

        return alignment
```

DTW の実装をすべてまとめてモジュールとしたものが以下のソースコードである。この実装では、ユークリッド距離とコサイン距離のどちらかから類似度計算関数を選択できるようにしている。

また、このモジュールをコマンドライン引数に 2 つの STF ファイルを与えて実行すると、DTW 前と後での 2 つの MFCC の第 1 次係数のズレの違いが確認できるようになっている。

リスト3.3 dtw.py

```
 1: #!/usr/bin/env python
 2:
 3: import numpy
 4: import scipy
 5: import scipy.linalg
 6: import sys
 7:
 8: class DTW:
 9:     def __getstate__(self):
10:         d = self.__dict__.copy()
11:
12:         if self.distance == self.cosine:
13:             d['distance'] = 'cosine'
14:         elif self.distance == self.euclidean:
15:             d['distance'] = 'euclidean'
16:
17:         return d
18:
19:     def __setstate__(self, dict):
20:         self.__dict__ = dict
21:
22:         if dict['distance'] == 'cosine':
23:             self.distance = self.cosine
24:         elif dict['distance'] == 'euclidean':
25:             self.distance = self.euclidean
26:
27:     def cosine(self, A, B):
28:         return scipy.dot(A, B.transpose()) / scipy.linalg.norm(A) \
29:                                                 / scipy.linalg.norm(B)
30:
31:     def euclidean(self, A, B):
32:         return scipy.linalg.norm(A - B)
33:
34:     def __init__(self, source, target, distance = None, window = sys.maxint):
35:         self.window = window
36:         self.source = source
37:         self.target = target
38:
39:         if distance:
40:             self.distance = distance
41:         else:
42:             self.distance = self.euclidean
43:
44:         self.dtw()
45:
46:     def dtw(self):
47:         < 省略 >
48:
49:     def align(self, data, reverse = False):
50:         < 省略 >
51:
52: if __name__ == '__main__':
53:     if len(sys.argv) < 3:
54:         print 'Usage: %s <source stf> <target stf>' % sys.argv[0]
55:         sys.exit()
56:
57:     from stf import STF
58:     source, target = STF(), STF()
59:     source.loadfile(sys.argv[1])
60:     target.loadfile(sys.argv[2])
```

```
61:
62:     from mfcc import MFCC
63:     mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency)
64:     source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
65:                         for frame in xrange(source.SPEC.shape[0])])
66:     mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency)
67:     target_mfcc = numpy.array([mfcc.mfcc(target.SPEC[frame]) \
68:                         for frame in xrange(target.SPEC.shape[0])])
69:
70:     dtw = DTW(source_mfcc, target_mfcc, \
71:         window = abs(source_mfcc.shape[0] - target_mfcc.shape[0]) * 2)
72:     warp_mfcc = dtw.align(source_mfcc)
73:
74:     import pylab
75:     pylab.subplot(211)
76:     pylab.plot(source_mfcc[:, 0])
77:     pylab.plot(target_mfcc[:, 0])
78:     pylab.subplot(212)
79:     pylab.plot(warp_mfcc[:, 0])
80:     pylab.plot(target_mfcc[:, 0])
81:     pylab.show()
```

このモジュールを同じ発話内容を異なる話者が発声したデータを使って実行した結果である。上のグラフだと、最初から大きくずれが発生し、終了位置も異なるが、下のグラフだと、波形がマッチしており、長さも等しくなっていることが分かる。

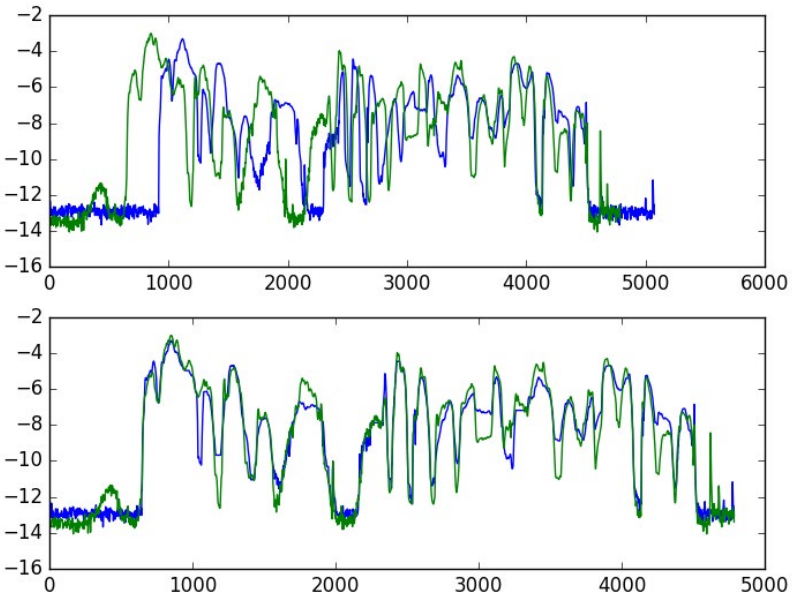

図3.4　DTWの実行結果

4 混合ガウスモデルによる声質変換

この章では、GMM を用いた最も基本的な声質変換の解説及び実装を行う。なお、声質変換の手法については、戸田氏の論文「Voice Conversion Based on Maximum Likelihood Estimation of Spectral Parameter Trajectory[6]」を、実装については、r9y9 氏のブログ [7]を大いに参照した。

4.1 混合ガウスモデルの学習処理

学習の処理はかなり単純である。変換元の特徴量と変換先の特徴量を各フレームごとに結合して得られるデータに対して、学習処理を行って GMM のパラメータを推定する。つまり、$\boldsymbol{x}_t, \boldsymbol{y}_t$ をそれぞれ t 番目のフレームにおける変換元、変換先の特徴量とすると、$\boldsymbol{z}_t = \left[\boldsymbol{x}_t^\top, \boldsymbol{y}_t^\top\right]^\top$ を学習データとして用いることとなる。ここで、特徴量として D 次元の MFCC を用いるとすると、学習データは $2D$ 次元となる。式で表すと以下のようになる。

$$P(\boldsymbol{z}_t|\boldsymbol{\lambda}^{(z)}) = \sum_{m=1}^{M} w_l \mathcal{N}(\boldsymbol{z}_t; \boldsymbol{\mu}_m^{(z)}, \boldsymbol{\Sigma}_m^{(z)})$$

$$\hat{\boldsymbol{\lambda}}^{(z)} = \underset{\boldsymbol{\lambda}^{(z)}}{argmax} \prod_{t=1}^{T} P(\boldsymbol{z}_t, \boldsymbol{\lambda}^{(z)})$$

ここで、$\boldsymbol{z}_t$ は、$\boldsymbol{x}_t, \boldsymbol{y}_t$ の結合特徴量であることから、平均ベクトルと分散共分散行列は以下のように表すことができる。

$$\boldsymbol{\mu}_m^{(z)} = \begin{bmatrix} \boldsymbol{\mu}_m^{(x)} \\ \boldsymbol{\mu}_m^{(y)} \end{bmatrix}, \boldsymbol{\Sigma}_m^{(z)} = \begin{bmatrix} \boldsymbol{\Sigma}_m^{(xx)} & \boldsymbol{\Sigma}_m^{(xy)} \\ \boldsymbol{\Sigma}_m^{(yx)} & \boldsymbol{\Sigma}_m^{(yy)} \end{bmatrix}$$

$\boldsymbol{\mu}_m^{(x)}, \boldsymbol{\mu}_m^{(y)}$ は、それぞれ m 番目のガウス分布における変換元、変換先の平均ベクトルを意味し、$\boldsymbol{\Sigma}_m^{(xx)}, \boldsymbol{\Sigma}_m^{(yy)}$ は、それぞれ m 番目のガウス分布における変換元、変換先の分散共分散行列、そして $\boldsymbol{\Sigma}_m^{(xy)}, \boldsymbol{\Sigma}_m^{(yx)}$ は、それぞれ変換元、変換先間の相互共分散行列と表している。

これは、学習したパラメータをプロットしてみるとよく分かる。以下の図は、変換元、変換先の特徴量をそれぞれ 16 次元の MFCC として、混合数 32 の GMM で学習した際の平均ベクトルをプロットしたものである。縦軸が混合数、横軸が特徴量の次元であり、左半分が $\boldsymbol{\mu}_m^{(x)}$、右半分が $\boldsymbol{\mu}_m^{(y)}$ を表しているが、大まかに似たような分布になっていることが分かる。

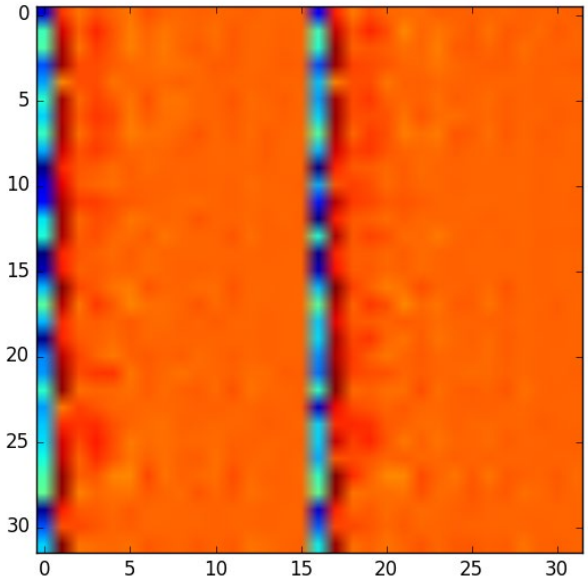

図4.1　学習したGMMの平均ベクトル

以下の図は、同様にして得られた分散共分散行列のうち、1 番目と 2 番目のガウス分布のものをプロットしたものである。4 つの領域で同様の分布になっており、さらにそれぞれが対称行列のようになっていることがわかる。

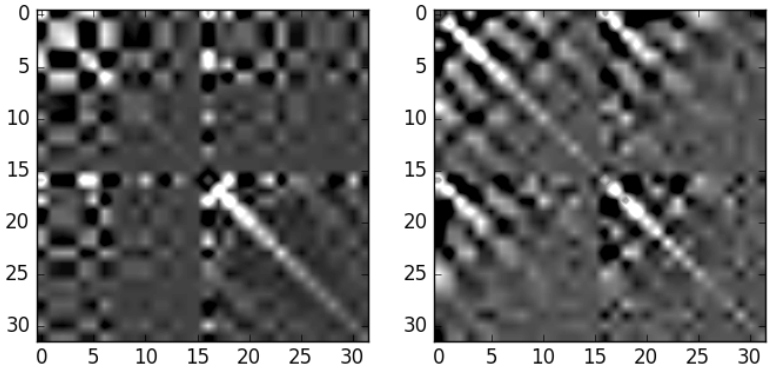

図4.2　学習したGMMの分散共分散行列

学習処理の実装

GMMMap というクラスに学習済み GMM インスタンスを与えることによって変換処理を行う

インスタンスを作るものとして、まず複数の STF データを読みこんで、結合特徴量を作り、GMM に学習させる処理までを実装する。

GMM の学習処理には、scikit-learn の GMM 実装である sklearn.mixture.GMM を用いる。ここで、気をつけるべきポイントはコンストラクタに covariance_type = 'full'を指定するという点である。scikit-learn は、デフォルトではパラメータの学習の際に、分散共分散行列を対角行列に制限することで計算量を削減しているが、より自然な音声を生成するにはその制約を用いずに行列をすべて更新するのがよい。

また、学習結果は Python の pickle モジュールを利用してシリアライズし、保存することとする。同様に、DTW はフレーム数に比例して計算量が大きくなるが、同じ音声データに対しての DTW の結果は常に一致することから、計算結果を pickle でキャッシュすることができる。

実装したソースは、以下の通りである。特に複雑な処理を実装しているわけではないので、詳細についてはコメントを参照していただきたい。

リスト4.1　learn_gmmmap.py

```
 1: #!/usr/bin/env python
 2:
 3: import math
 4: import numpy
 5: import os
 6: import pickle
 7: import re
 8: import sklearn
 9: import sklearn.mixture
10: import sys
11:
12: from gmmmap import GMMMap
13:
14: from stf import STF
15: from mfcc import MFCC
16: from dtw import DTW
17:
18: D = 16  # 利用する MFCC の次元数
19: M = 32  # GMM の混合数
20:
21: if __name__ == '__main__':
22:     if len(sys.argv) < 5:
23:         print ('Usage: %s [list of source stf] [list of target stf] ' + \
24:                               '[dtw cache directory] [output file]') % sys.argv[0]
25:         sys.exit()
26:
27:     # 対応する STF ファイルのパスが同じ行に書かれたリストを入力として受け取る
28:     source_list = open(sys.argv[1]).read().strip().split('\n')
29:     target_list = open(sys.argv[2]).read().strip().split('\n')
30:
31:     assert len(source_list) == len(target_list)
32:
33:     learn_data = None
34:
35:     for i in xrange(len(source_list)):
36:         # 変換元の STF を読み取る
37:         source = STF()
38:         source.loadfile(source_list[i])
39:
40:         # 変換元のスペクトル包絡から各フレームごとに MFCC を計算する
41:         mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = D)
```

```
42:         source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
43:                                    for frame in xrange(source.SPEC.shape[0])])
44:
45:         # 変換先の STF を読み取る
46:         target = STF()
47:         target.loadfile(target_list[i])
48:
49:         # 変換先のスペクトル包絡から各フレームごとに MFCC を計算する
50:         mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency, dimension = D)
51:         target_mfcc = numpy.array([mfcc.mfcc(target.SPEC[frame]) \
52:                                    for frame in xrange(target.SPEC.shape[0])])
53:
54:         # DTW のキャッシュが存在しない場合は DP マッチングの計算処理を行う
55:         cache_path = os.path.join(sys.argv[3], '%s_%s.dtw' % \
56:             tuple(map(lambda x: re.sub('[./]', '_', re.sub('^[./]*', '', x)), \
57:                                         [source_list[i], target_list[i]])))
58:         if os.path.exists(cache_path):
59:             dtw = pickle.load(open(cache_path))
60:         else:
61:             dtw = DTW(source_mfcc, target_mfcc, \
62:                     window = abs(source.SPEC.shape[0] - target.SPEC.shape[0]) * 2)
63:             with open(cache_path, 'wb') as output:
64:                 pickle.dump(dtw, output)
65:
66:         # DTW により変換元の MFCC のフレーム数を変換先と合わせる
67:         warp_mfcc = dtw.align(source_mfcc)
68:
69:         # 変換元と変換先の MFCC を結合し、各フレームごとに 2D 次元の特徴量となるようにする
70:         data = numpy.hstack([warp_mfcc, target_mfcc])
71:         # STF ファイルごとの結合特徴量を時間方向に繋げて 1 つの行列とする
72:         if learn_data is None:
73:             learn_data = data
74:         else:
75:             learn_data = numpy.vstack([learn_data, data])
76:
77:     # GMM の学習処理を行う
78:     gmm = sklearn.mixture.GMM(n_components = M, covariance_type = 'full')
79:     gmm.fit(learn_data)
80:     gmmmap = GMMMap(gmm)
81:
82:     # 学習済みインスタンスを pickle でシリアライズする
83:     with open(sys.argv[4], 'wb') as output:
84:         pickle.dump(gmmmap, output)
```

4.2 混合ガウスモデルによる変換処理

変換処理は、入力として変換元の特徴量 $\boldsymbol{x}_t$ が与えられた時の、変換後の特徴量の条件付き確率密度関数 $P(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})$ を求め、それが最大化されるような $\boldsymbol{y}_t$ を求めることによって行われる。

ここで、

$$P(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)}) = \sum_{m=1}^{M} P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)}) P(\boldsymbol{y}_t|\boldsymbol{x}_t, m, \boldsymbol{\lambda}^{(z)})$$

となるが、ここで $P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})$ は事後確率として以下のように導出できる。

$$P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)}) = \frac{w_m \mathcal{N}(\boldsymbol{x}_t; \boldsymbol{\mu}_m^{(z)}, \boldsymbol{\Sigma}_m^{(z)})}{\sum_{n=1}^{M} w_n \mathcal{N}(\boldsymbol{x}_t, \boldsymbol{\mu}_n^{(z)}, \boldsymbol{\Sigma}_n^{(z)})}$$

さらに、$P(\boldsymbol{y}_t|\boldsymbol{x}_t, m, \boldsymbol{\lambda}^{(z)})$ も GMM でモデル化することができ、その平均ベクトル $\boldsymbol{E}_{m,t}^{(y)}$ 及び分散共分散行列 $\boldsymbol{D}_m^{(y)}$ は以下のように表される[3]。

$$\boldsymbol{E}_{m,t}^{(y)} = \boldsymbol{\mu}_m^{(y)} + \boldsymbol{\Sigma}_m^{(yx)}{\boldsymbol{\Sigma}_m^{(xx)}}^{-1}(\boldsymbol{x}_t - \boldsymbol{\mu}_m^{(x)})$$
$$\boldsymbol{D}_m^{(y)} = \boldsymbol{\Sigma}_m^{(yy)} + \boldsymbol{\Sigma}_m^{(yx)}{\boldsymbol{\Sigma}_m^{(xx)}}^{-1}\boldsymbol{\Sigma}_m^{(xy)}$$

このとき、最小平均二乗誤差推定 (MMSE) によって変換後の特徴量を求めるならば、推定される特徴量を $\hat{\boldsymbol{y}}_t$ として、以下の通りとなる。

$$\hat{\boldsymbol{y}}_t = E[\boldsymbol{y}_t|\boldsymbol{x}_t] = \int P(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})\boldsymbol{y}_t d\boldsymbol{y}_t$$

これに、得られた $P(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})$ を代入する。

$$\begin{aligned}\int P(\boldsymbol{y}_t|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})\boldsymbol{y}_t d\boldsymbol{y}_t &= \int \sum_{m=1}^{M} P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})P(\boldsymbol{y}_t|\boldsymbol{x}_t, m, \boldsymbol{\lambda}^{(z)})\boldsymbol{y}_t d\boldsymbol{y}_t \\ &= \sum_{m=1}^{M} P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)}) \int \mathcal{N}(\boldsymbol{x}_t, \boldsymbol{\mu}_n^{(z)}, \boldsymbol{\Sigma}_n^{(z)})\boldsymbol{y}_t d\boldsymbol{y}_t \\ &= \sum_{m=1}^{M} P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})\boldsymbol{E}_{m,t}^{(y)}\end{aligned}$$

以上より、$P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})$ と $\boldsymbol{E}_{m,t}^{(y)}$ の算出を実装すれば変換処理ができることがわかる。

変換処理の実装

変換処理についても、学習処理と同じくコメントにて解説を入れる。基本的な構造としては、コンストラクタで学習済みの GMM を引数として取り、変換元データと独立な部分を算出した後、convert メソッドで変換元データを引数として取って、変換処理を行う。

1 つ注意すべきポイントは、$P(m|\boldsymbol{x}_t, \boldsymbol{\lambda}^{(z)})$ の算出を、前章の定義をそのまま実装するのではなく、sklearn.mixture.GMM の predict_proba メソッドを用いて行っているという点である。この predict_proba メソッドでは GMM から事後確率を計算することができるので、コンストラクタ内で $\boldsymbol{\mu}_m^{(x)}$ を平均ベクトル、$\boldsymbol{\Sigma}_m^{(xx)}$ を分散共分散行列とする GMM インスタンスを作成している。

また、sklearn.mixture.GMM では、それぞれのガウス分布の平均ベクトルと分散共分散行列をまとめて多次元配列として扱っているということも把握しておく必要がある。つまり、平均ベクトルは $M \times 2D$ 次元、分散共分散行列は $M \times 2D \times 2D$ 次元の配列となっている。

3 この平均ベクトル・分散共分散行列の導出については、「パターン認識と機械学習」の「2.3.1 条件付きガウス分布」を参照されたい。

リスト4.2　gmmmap.py

```
#!/usr/bin/python
# coding: utf-8

import numpy as np
from sklearn.mixture import GMM

class GMMMap(object):
    def __init__(self, gmm, swap = False):
        # GMM の学習に用いられるのは結合特徴量なので、その半分が MFCC の次元数となる
        self.M, D = gmm.means_.shape[0], gmm.means_.shape[1] / 2
        self.weights = gmm.weights_

        # 学習済み GMM の平均ベクトルを x と y に分ける
        self.src_means = gmm.means_[:, :D]
        self.tgt_means = gmm.means_[:, D:]

        # 学習済み GMM の分散共分散行列を xx, xy, yx, yy の 4 つに分ける
        self.covarXX = gmm.covars_[:, :D, :D]
        self.covarXY = gmm.covars_[:, :D, D:]
        self.covarYX = gmm.covars_[:, D:, :D]
        self.covarYY = gmm.covars_[:, D:, D:]

        # GMM の学習時と逆に変換先の話者から変換元の話者へと変換する場合は
        # 平均ベクトルと分散共分散行列を逆に扱えばよい
        if swap:
            self.tgt_means, self.src_means = self.src_means, self.tgt_means
            self.covarYY, self.covarXX = self.covarXX, self.covarYY
            self.covarYX, self.covarXY = self.covarXY, self.covarYX

        # 事後確率の計算のために、それぞれのガウス分布の重みはそのままで
        # x の平均ベクトルと xx の分散共分散行列を用いた GMM のインスタンスを生成する
        self.px = GMM(n_components = self.M, covariance_type = "full")
        self.px.means_ = self.src_means
        self.px.covars_ = self.covarXX
        self.px.weights_ = self.weights

    def convert(self, src):
        D = len(src)

        # ベクトル E をすべてのガウス分布についてまとめて計算する
        E = np.zeros((self.M, D))
        for m in range(self.M):
            # 逆行列に行列を掛け合わせる処理は numpy.linalg.solve を使うと高速である
            xx = np.linalg.solve(self.covarXX[m], src - self.src_means[m])
            E[m] = self.tgt_means[m] + self.covarYX[m].dot(xx.transpose())

        # 事後確率 P(m|x) を計算する
        posterior = self.px.predict_proba(np.atleast_2d(src))

        # 事後確率と E の積が求める特徴量となる
        return posterior.dot(E)
```

4.3　変換特徴量から音声データへの変換

ここまでで、GMM を用いて特徴量を変換することはできたが、最後に得られた特徴量を音声データへと変換する処理を実装する必要がある。GMM によって変換される特徴量は MFCC なので、スペクトル包絡へと逆変換した後に、変換元データとして与えられた STF ファイルのスペクトル包絡を上書きして保存する。そして、TANDEM-STRAIGHT によって、変換後の STF ファイルを

音声データへと変換する。

この時に、F0 周波数も変換することで、より変換先の話者に似せることができる。F0 の変換については、複雑な処理は行わずに以下のように線形変換をする。

$$\hat{\boldsymbol{y}}_t = \frac{\rho^{(y)}}{\rho^{(x)}}(\boldsymbol{x}_t - \mu^{(x)}) + \mu^{(y)}$$

ここで、$\boldsymbol{x}_t, \boldsymbol{y}_t$ は対数尺度での変換元、変換先の F0 周波数とし、$\mu^{(x)}, \rho^{(x)}$ はそれぞれ変換元の対数 F0 周波数の平均及び標準偏差、同様に、$\mu^{(y)}, \rho^{(y)}$ はそれぞれ変換先の対数 F0 周波数の平均及び標準偏差とする。これもスペクトル包絡と同様に、変換元 STF の F0 データを変換したデータで上書きする。

F0 変換パラメータの算出処理

まず、F0 の変換に用いる、対数 F0 周波数の平均と標準偏差の算出処理を実装する。算出に用いる STF ファイルが 1 つとは限らないので、ファイルごとに対数 F0 周波数の平均と二乗平均を更新していき、最後に二乗平均と平均の二乗の差の平方根を取ることで標準偏差を算出している。

算出結果は、タプルとして pickle でシリアライズして保存する。

リスト4.3　learn_f0.py

```
 1: #!/usr/bin/env python
 2:
 3: import math
 4: import numpy
 5: import pickle
 6: import sklearn
 7: import sys
 8:
 9: from stf import STF
10: from mfcc import MFCC
11: from dtw import DTW
12:
13: if __name__ == '__main__':
14:     if len(sys.argv) < 4:
15:         print 'Usage: %s [list of source stf] ' + \
16:                 '[list of target stf] [output file]' % sys.argv[0]
17:         sys.exit()
18:
19:     source_list = open(sys.argv[1]).read().strip().split('\n')
20:     target_list = open(sys.argv[2]).read().strip().split('\n')
21:
22:     assert len(source_list) == len(target_list)
23:
24:     f0_count = [0, 0]
25:     f0_mean = [0.0, 0.0]
26:     f0_square_mean = [0.0, 0.0]
27:
28:     for i in xrange(len(source_list)):
29:         source = STF()
30:         source.loadfile(source_list[i])
31:
32:         target = STF()
33:         target.loadfile(target_list[i])
```

```
34:
35:         for idx, stf in enumerate([source, target]):
36:             count = (stf.F0 != 0).sum()
37:             f0_mean[idx] = (f0_mean[idx] * f0_count[idx] + \
38:                     numpy.log(stf.F0[stf.F0 != 0]).sum()) / (f0_count[idx] + count)
39:             f0_square_mean[idx] = (f0_square_mean[idx] * f0_count[idx] + \
40:                     (numpy.log(stf.F0[stf.F0 != 0]) ** 2).sum()) / \
41:                     (f0_count[idx] + count)
42:             f0_count[idx] += count
43:
44:     f0_deviation = [math.sqrt(f0_square_mean[i] - f0_mean[i] ** 2) \
45:                                                 for i in xrange(2)]
46:     f0 = (tuple(f0_mean), tuple(f0_deviation))
47:
48:     print f0
49:     output = open(sys.argv[3], 'wb')
50:     pickle.dump(f0, output)
51:     output.close()
```

STF ファイルへの変換処理

保存された GMMMap インスタンスを用いて特徴量を変換し、MFCC の逆変換によってスペクトル包絡を復元した後に STF ファイルに保存する処理を実装する。

リスト4.4　convert_gmmmap.py

```
 1: #!/usr/bin/env python
 2:
 3: import math
 4: import numpy
 5: import pickle
 6: import sklearn
 7: import sys
 8:
 9: from gmmmap import GMMMap
10:
11: from stf import STF
12: from mfcc import MFCC
13: from dtw import DTW
14:
15: K = 32
16: DIMENSION = 16
17:
18: if __name__ == '__main__':
19:     if len(sys.argv) < 5:
20:         print 'Usage: %s [gmmmap] [f0] [input] [output]' % sys.argv[0]
21:         sys.exit()
22:
23:     # 保存された GMMMap インスタンスを読み込む
24:     gmm_file = open(sys.argv[1], 'rb')
25:     gmmmap = pickle.load(gmm_file)
26:     gmm_file.close()
27:
28:     # F0 の変換パラメータを読み込む
29:     f0_file = open(sys.argv[2], 'rb')
30:     f0 = pickle.load(f0_file)
31:     f0_file.close()
32:
33:     source = STF()
34:     source.loadfile(sys.argv[3])
35:     # F0 の有声部分について、パラメータに基づいて変換する
36:     source.F0[source.F0 != 0] = \
37:         numpy.exp((numpy.log(source.F0[source.F0 != 0]) - f0[0][0]) \
```

```
38:                                         * f0[1][1] / f0[1][0] + f0[0][1])
39:
40:     # 変換元の MFCC を計算する
41:     mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = DIMENSION)
42:     source_data = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
43:                                         for frame in xrange(source.SPEC.shape[0])])
44:
45:     # GMMMap で特徴量を変換し、MFCC の逆変換でスペクトル包絡を復元する
46:     output_mfcc = numpy.array([gmmmap.convert(source_data[frame])[0] \
47:                                         for frame in xrange(source_data.shape[0])])
48:     output_spec = numpy.array([mfcc.imfcc(output_mfcc[frame]) \
49:                                         for frame in xrange(output_mfcc.shape[0])])
50:
51:     # STF ファイルのスペクトル包絡を上書きして保存する
52:     source.SPEC = output_spec
53:     source.savefile(sys.argv[4])
```

4.4 混合ガウスモデルによる変換処理の結果

以下の画像は、GMM による変換の結果を表したグラフである。それぞれのグラフは MFCC の第 1 次係数（数値が小さい方）及び第 2 次係数（数値が大きい方）の推移を表しており、上から、変換先話者のデータ、GMM による変換結果のデータ、GMM による変換に用いた変換元話者のデータを DTW によって伸縮させたものである。非常にわかりにくいが、下段のグラフよりは中段のグラフの方が、上段のグラフに近いように見えなくもない。また、この手法ではフレームごとに独立して変換を行っているため、変換結果のグラフがかなりギザギザになっていることもわかる。

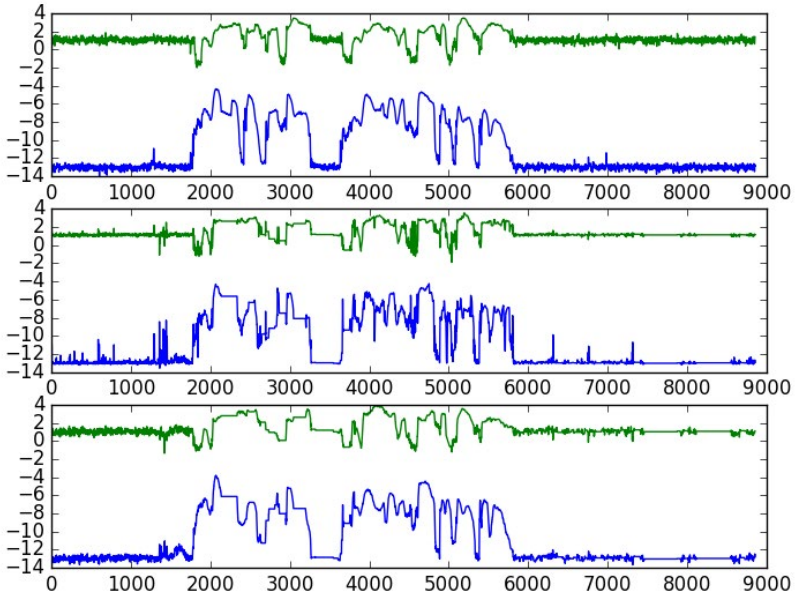

図4.3 GMMによる変換結果と変換先話者データの比較

5 トラジェクトリベースな声質変換

前章では、最も素朴な声質変換手法について説明したが、これには 1 つ大きな問題点がある。それは、前後フレームとの関連性を全く考慮せずにフレームごとに独立に変換しているために、グラフでも推移がギザギザになっていた通り、自然性（声のナチュラルさ）が失われてしまっているという点である。

この章では、前後フレームとのつながりを考慮した、トラジェクトリベースな変換と呼ばれる変換手法について解説する。なお、この章についても、戸田氏の論文 [6]及び、r9y9 氏のブログ [7] を参照した。

5.1 動的特徴量の導入

前後フレームとのつながりを考慮した変換を行うために、MFCC の抽出の際に実装した動的特徴量を導入する。ここでは、動的特徴量 $\Delta \boldsymbol{x}_t$ を以下のように定義する。

$$\Delta \boldsymbol{x}_t = \frac{1}{2}(\boldsymbol{x}_{t+1} - \boldsymbol{x}_{t-1})$$

そして、前章での変換元、変換先の特徴量の代わりに、動的変化量を結合した特徴量 $\boldsymbol{X}_t = [\boldsymbol{x}_t^\top, \Delta \boldsymbol{x}_t^\top], \boldsymbol{Y}_t = [\boldsymbol{y}_t^\top, \Delta \boldsymbol{y}_t^\top]$ を用いる。時系列で結合した全体の特徴量は、以下の通りとなる。

$$\boldsymbol{X} = [\boldsymbol{X}_1^\top, \boldsymbol{X}_2^\top, ..., \boldsymbol{X}_t^\top, ..., \boldsymbol{X}_T^\top],\ \boldsymbol{Y} = [\boldsymbol{Y}_1^\top, \boldsymbol{Y}_2^\top, ..., \boldsymbol{Y}_t^\top, ..., \boldsymbol{Y}_T^\top]$$

この時、$\boldsymbol{x}_t, \boldsymbol{y}_t$が共に$D$次元のベクトルだとすると、$\boldsymbol{X}, \boldsymbol{Y}$ は共に $2D \times T$ 次元の行列となる。

また、前章と同様に変換元と変換先の特徴量を結合した $4D$ 次元の特徴量 $\boldsymbol{Z}_t = [\boldsymbol{X}_t^\top, \boldsymbol{Y}_t^\top]$ を用いて GMM の学習処理を行い、$\boldsymbol{\lambda}^{(Z)}$ を求める。

学習処理の実装

学習処理については、特徴量に動的変化量を結合する以外には、前章との大きな違いはない。ここでも、変換処理は TrajectoryGMMMap というクラスで行うものとして、まず学習処理のみを実装する。

リスト5.1　learn_trajectory.py

```
1: #!/usr/bin/env python
2:
3: import math
4: import numpy
```

```
 5: import os
 6: import pickle
 7: import re
 8: import sklearn
 9: import sklearn.mixture
10: import sys
11:
12: from trajectory import TrajectoryGMMMap
13:
14: from stf import STF
15: from mfcc import MFCC
16: from dtw import DTW
17:
18: D = 16
19: M = 32
20:
21: if __name__ == '__main__':
22:     if len(sys.argv) < 5:
23:         print 'Usage: %s [list of source stf] [list of target stf] ' + \
24:                             [dtw cache directory] [output file]' % sys.argv[0]
25:         sys.exit()
26:
27:     source_list = open(sys.argv[1]).read().strip().split('\n')
28:     target_list = open(sys.argv[2]).read().strip().split('\n')
29:
30:     assert len(source_list) == len(target_list)
31:
32:     learn_data = None
33:
34:     for i in xrange(len(source_list)):
35:         source = STF()
36:         source.loadfile(source_list[i])
37:
38:         mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = D)
39:         source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
40:                                    for frame in xrange(source.SPEC.shape[0])])
41:
42:         target = STF()
43:         target.loadfile(target_list[i])
44:
45:         mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency, dimension = D)
46:         target_mfcc = numpy.array([mfcc.mfcc(target.SPEC[frame]) \
47:                                    for frame in xrange(target.SPEC.shape[0])])
48:
49:         cache_path = os.path.join(sys.argv[3], '%s_%s.dtw' \
50:             % tuple(map(lambda x: re.sub('[./]', '_', re.sub('^[./]*', '', x)), \
51:                                                 [source_list[i], target_list[i]])))
52:         if os.path.exists(cache_path):
53:             dtw = pickle.load(open(cache_path))
54:         else:
55:             dtw = DTW(source_mfcc, target_mfcc, \
56:                     window = abs(source.SPEC.shape[0] - target.SPEC.shape[0]) * 2)
57:             with open(cache_path, 'wb') as output:
58:                 pickle.dump(dtw, output)
59:
60:         warp_mfcc = dtw.align(source_mfcc)
61:
62:         # 変換元、変換先共に、動的変化量を結合する
63:         warp_data = numpy.hstack([warp_mfcc, mfcc.delta(warp_mfcc)])
64:         target_data = numpy.hstack([target_mfcc, mfcc.delta(target_mfcc)])
65:
66:         data = numpy.hstack([warp_data, target_data])
67:         if learn_data is None:
68:             learn_data = data
69:         else:
70:             learn_data = numpy.vstack([learn_data, data])
71:
72:     gmm = sklearn.mixture.GMM(n_components = M, covariance_type = 'full')
```

```
73:     gmm.fit(learn_data)
74:
75:     gmmmap = TrajectoryGMMMap(gmm, learn_data.shape[0])
76:
77:     with open(sys.argv[4], 'wb') as output:
78:         pickle.dump(gmmmap, output)
```

5.2 トラジェクトリベースな変換処理

時系列に結合した特徴量に対する確率密度関数

前章では、フレームごとに確率密度関数を求め、変換処理を行っていたが今回は前後でのつながりを考えるために、$\boldsymbol{X}, \boldsymbol{Y}$ に関する確率密度関数を考える。 $\boldsymbol{X}, \boldsymbol{Y}$ は単純にフレームごとの特徴量を時系列でつなげたものなので、前章で用いた確率密度関数のすべてのフレームにおける積を考えればよい。

$$P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)}) = \prod_{t=1}^{T} \sum_{m=1}^{M} P(m|\boldsymbol{X}_t, \boldsymbol{\lambda}^{(Z)}) P(\boldsymbol{Y}_t|\boldsymbol{X}_t, m, \boldsymbol{\lambda}^{(Z)})$$

このとき、前章と同様に、$P(m|\boldsymbol{X}_t, \boldsymbol{\lambda}^{(Z)})$ 及び $P(\boldsymbol{Y}_t|\boldsymbol{X}_t, m, \boldsymbol{\lambda}^{(Z)})$ は以下のように表すことができる。

$$P(m|\boldsymbol{X}_t, \lambda^{(Z)}) = \frac{w_m \mathcal{N}(\boldsymbol{X}_t; \boldsymbol{\mu}_m^{(Z)}, \boldsymbol{\Sigma}_m^{(Z)})}{\sum_{n=1}^{M} w_n \mathcal{N}(\boldsymbol{X}_t, \boldsymbol{\mu}_n^{(Z)}, \boldsymbol{\Sigma}_n^{(Z)})}$$

$$P(\boldsymbol{Y}_t|\boldsymbol{X}_t, m, \boldsymbol{\lambda}^{(Z)}) = \mathcal{N}(\boldsymbol{Y}_t; \boldsymbol{E}_{m,t}^{(Y)}, \boldsymbol{D}_m^{(Y)})$$

ここで、$\boldsymbol{E}_{m,t}^{(y)}$ 及び $\boldsymbol{D}_m^{(y)}$ は以下の通りである。

$$\boldsymbol{E}_{m,t}^{(Y)} = \boldsymbol{\mu}_m^{(Y)} + \boldsymbol{\Sigma}_m^{(YX)} \boldsymbol{\Sigma}_m^{(XX)^{-1}} (\boldsymbol{X}_t - \boldsymbol{\mu}_m^{(X)})$$
$$\boldsymbol{D}_m^{(Y)} = \boldsymbol{\Sigma}_m^{(YY)} + \boldsymbol{\Sigma}_m^{(YX)} \boldsymbol{\Sigma}_m^{(XX)^{-1}} \boldsymbol{\Sigma}_m^{(XY)}$$

また、確率密度関数 $P(\boldsymbol{X}|\boldsymbol{Y}, \boldsymbol{\lambda}^{(Z)})$ は、分布系列 $\boldsymbol{m} = \{m_1, m_2, ..., m_t, ..., m_T\}$ を用いて、以下のように表すことができる。

$$P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)}) = \sum_{all\ \boldsymbol{m}} P(\boldsymbol{m}|\boldsymbol{X}_t, \boldsymbol{\lambda}^{(Z)}) P(\boldsymbol{Y}_t|\boldsymbol{X}_t, \boldsymbol{m}, \boldsymbol{\lambda}^{(Z)})$$

この確率密度関数に基づいて、求める特徴量 $\hat{\boldsymbol{y}}$ は以下のように定式化できる。

$$\hat{\boldsymbol{y}} = \underset{\boldsymbol{y}}{argmax}\ P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)})$$

ただし、この確率密度関数は $\boldsymbol{X}$ が与えられた時の $\boldsymbol{Y}$ についての確率を記述したものなので、$\boldsymbol{y}$ と $\boldsymbol{Y}$ の関係性を与える必要がある。そこで、以下のように定義される変換行列 $\boldsymbol{W}$ を導入する。

$$\boldsymbol{W} = [\boldsymbol{W}_1, \boldsymbol{W}_2, ..., \boldsymbol{W}_t, ..., \boldsymbol{W}_T]^\top \otimes \boldsymbol{I}_{\boldsymbol{D}\times\boldsymbol{D}}$$

$$\boldsymbol{W}_t = [\boldsymbol{w}_t^{(0)}, \boldsymbol{w}_t^{(1)}]$$

$$\boldsymbol{w}_t^{(0)} = [\overset{1st}{0}, \overset{2nd}{0}, ..., \overset{t-1\ th}{0}, \overset{t\ th}{1}, \overset{t+1\ th}{0}, ..., \overset{T\ th}{0}]$$

$$\boldsymbol{w}_t^{(1)} = [\overset{1st}{0}, \overset{2nd}{0}, ..., \overset{t-1\ th}{-0.5}, \overset{t\ th}{0}, \overset{t+1\ th}{0.5}, ..., \overset{T\ th}{0}]$$

ここで、各フレームについて $\boldsymbol{W}_t$ が $\boldsymbol{y}_t$ から $\boldsymbol{Y}_t$ への変換を担っているが、$\boldsymbol{w}_t^{(0)}$ が $\boldsymbol{y}_t$ をそのまま移し、$\boldsymbol{w}_t^{(1)}$ が、この章の冒頭で定義した動的特徴量と対応していることが分かる。ただし、実装に合わせるために、$\Delta\boldsymbol{x}_0 = \frac{1}{2}(\boldsymbol{x}_1 - \boldsymbol{x}_0)$ 及び $\Delta\boldsymbol{x}_T = \frac{1}{2}(\boldsymbol{x}_T - \boldsymbol{x}_{T-1})$ とし、変換行列についても、$\boldsymbol{W}_0, \boldsymbol{W}_T$ を以下の図のように設定する。

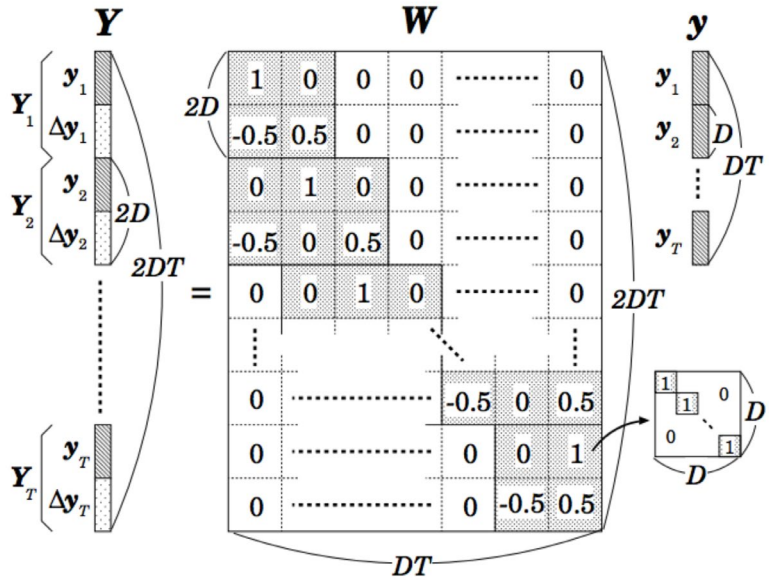

図5.1　変換行列Wのイメージ([6]にあった図を元に編集)

元論文 [6]では、EM アルゴリズムを用いて、この定式化に基づいた変換パラメータの導出方法も紹介されているが、ここでは準最適な分布系列を用いて計算量を削減する手法を用いる。これは、それぞれのフレームごとに用いるガウス分布を 1 つにするということで、つまり、分布系列 $\boldsymbol{m}$ を準最適な $\hat{\boldsymbol{m}} = [\hat{m}_1, \hat{m}_2, ..., \hat{m}_t, ..., \hat{m}_T]$ の 1 つに固定することによって、以下のように近似する。

$$P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)}) \simeq P(\hat{\boldsymbol{m}}|\boldsymbol{X}_t, \boldsymbol{\lambda}^{(Z)}) P(\boldsymbol{Y}_t|\boldsymbol{X}_t, \hat{\boldsymbol{m}}, \boldsymbol{\lambda}^{(Z)})$$

このとき、$\hat{\boldsymbol{m}}$ は以下のように決定される。

$$\hat{\boldsymbol{m}} = \underset{\boldsymbol{m}}{argmax}\ P(\boldsymbol{m}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)})$$

これらをまとめると、求める特徴量は以下のように表される。

$$\begin{aligned}\hat{\boldsymbol{y}} &= \underset{\boldsymbol{y}}{argmax}\ P(\boldsymbol{Y}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)})\ subject\ to\ \boldsymbol{Y} = \boldsymbol{W}\boldsymbol{y} \\ &\simeq P(\hat{\boldsymbol{m}}|\boldsymbol{X}_t, \boldsymbol{\lambda}^{(Z)}) P(\boldsymbol{Y}_t|\boldsymbol{X}_t, \hat{\boldsymbol{m}}, \boldsymbol{\lambda}^{(Z)})\ subject\ to\ \boldsymbol{Y} = \boldsymbol{W}\boldsymbol{y} \\ &\qquad where\ \hat{\boldsymbol{m}} = \underset{\boldsymbol{m}}{argmax}\ P(\boldsymbol{m}|\boldsymbol{X}, \boldsymbol{\lambda}^{(Z)})\end{aligned}$$

変換特徴量の導出

求めた確率密度関数に基づいて、最尤推定を行い、変換後の特徴量を導出する。対数尤度を考えることによって、以下のように特徴量を求めることができるが、具体的な導出手法については筆者の理解の浅さと 7 時間後に迫った締め切りのために割愛させていただく。詳しくは、元論文 [6]の Appendix を参照してほしい。

$$\hat{\boldsymbol{y}} = (\boldsymbol{W}^\top {\boldsymbol{D}_{\hat{\boldsymbol{m}}}^{(Y)}}^{-1} \boldsymbol{W})^{-1} \boldsymbol{W}^\top {\boldsymbol{D}_{\hat{\boldsymbol{m}}}^{(Y)}}^{-1} \boldsymbol{E}_{\hat{\boldsymbol{m}}}^{(Y)}$$

ここで、$\boldsymbol{E}_{\hat{\boldsymbol{m}}}^{(Y)}, {\boldsymbol{D}_{\hat{\boldsymbol{m}}}^{(Y)}}^{-1}$ は以下のように与えられる。

$$\begin{aligned}\boldsymbol{E}_{\hat{\boldsymbol{m}}}^{(Y)} &= [\boldsymbol{E}_{\hat{m}_1,1}^{(Y)}, \boldsymbol{E}_{\hat{m}_2,2}^{(Y)}, ..., \boldsymbol{E}_{\hat{m}_t,t}^{(Y)}, ..., \boldsymbol{E}_{\hat{m}_T,T}^{(Y)}] \\ {\boldsymbol{D}_{\hat{\boldsymbol{m}}}^{(Y)}}^{-1} &= diag\ [{\boldsymbol{D}_{\hat{\boldsymbol{m}}_1}^{(Y)}}^{-1}, {\boldsymbol{D}_{\hat{\boldsymbol{m}}_2}^{(Y)}}^{-1}, ..., {\boldsymbol{D}_{\hat{\boldsymbol{m}}_t}^{(Y)}}^{-1}, ..., {\boldsymbol{D}_{\hat{\boldsymbol{m}}_T}^{(Y)}}^{-1}]\end{aligned}$$

変換処理の実装

導出結果に基づいて、変換処理の実装を行う。具体的には、学習処理の項でも述べたように、GMMMap クラスを継承した TrajectoryGMMMap クラスを実装する。

まず、コンストラクタにて、変換元のデータと独立で求められる $\boldsymbol{D}_m^{(Y)}$ の計算処理を行う。

行列Dの算出

```
    def __init__(self, gmm, swap = False):
        # GMMMap のコンストラクタを呼び出す
        super(TrajectoryGMMMap, self).__init__(gmm, swap)

        D = gmm.means_.shape[1] / 2

        # すべての m について P(Y|X) における分散共分散行列 D をまとめて 1 つの変数として扱う
        self.D = np.zeros((self.M, D, D))
        for m in range(self.M):
            xx_inv_xy = np.linalg.solve(self.covarXX[m], self.covarXY[m])
            self.D[m] = self.covarYY[m] - np.dot(self.covarYX[m], xx_inv_xy)
```

次に、変換処理を実装するにあたり、$\boldsymbol{y}$ から $\boldsymbol{Y}$ への変換行列 $\boldsymbol{W}$ を生成するメソッドを実装する。$\boldsymbol{W}$ の生成は、T 回のループで $\boldsymbol{w}_t^{(0)}, \boldsymbol{w}_t^{(1)}$ を生成し、つなげていくことで行っている。

また、ここで注意すべきなのは scipy.sparse モジュールを使うことで高速化を図っているという点である。scipy.sparse は疎行列を扱うためのモジュールで、$\boldsymbol{W}$ は図 5.1を見れば分かるように対角成分付近以外はほとんどが 0 の疎行列なので、これを用いることで計算量や必要なメモリを減らすことができる。

scipy.sparse モジュールには幾つかの疎行列の実装があるが、ここでは lil_matrix と csr_matrix の 2 つを用いている。lil_matrix ではインデックスを指定してデータを割り当てができる一方で、行列に対する計算操作は csr_matrix の方が高速なので、lil_matrix で行列の生成をした後に csr_matrix への変換を行っている。

行列Wの算出

```
    def __construct_weight_matrix(self, T, D):
        W = None

        for t in range(T):
            # 図の各行に対応する行列を生成する
            w0 = scipy.sparse.lil_matrix((D, D * T))
            w1 = scipy.sparse.lil_matrix((D, D * T))

            # scipy.sparse.diags を使って図の「1」のマスに該当する部分の対角成分に 1 を代入する
            w0[0:, t * D: (t + 1) * D] = scipy.sparse.diags(np.ones(D), 0)

            # 図の「-0.5」のマスに該当する部分の対角成分に -0.5 を代入する
            tmp = np.zeros(D).fill(-0.5)
            if t == 0:
                w1[0:, :D] = scipy.sparse.diags(tmp, 0)
            else:
                # t == 0 でない場合は t - 1 番目のマスに代入する
                w1[0:, (t - 1) * D: t * D] = scipy.sparse.diags(tmp, 0)

            # 図の「0.5」のマスに該当する部分の対角成分に 0.5 を代入する
            tmp = np.zeros(D).fill(-0.5)
            if t == T - 1:
                w1[0:, t * D:] = scipy.sparse.diags(tmp, 0)
            else:
                # t == 1 でない場合は t + 1 番目のマスに代入する
                w1[0:, (t + 1) * D: (t + 2) * D] = scipy.sparse.diags(tmp, 0)

            # w0 と w1 を結合したものを積み重ねていく
            W_t = scipy.sparse.vstack([w0, w1])
            if W == None:
                W = W_t
            else:
                W = scipy.sparse.vstack([W, W_t])

        # 最後に lil_matrix を csr_matrix へと変換する
        return W.tocsr()
```

最後に、変換処理本体を実装する。まず、変換行列 $\boldsymbol{W}$ を生成した後に、$\hat{\boldsymbol{m}}$ を求め、それに基づいて $\boldsymbol{E}_{\hat{\boldsymbol{m}}}^{(Y)}, \boldsymbol{D}_{\hat{\boldsymbol{m}}}^{(Y)^{-1}}$ を計算する。そして、求めた行列の積を scipy.sparse.linalg.spsolve によって計算し、特徴量を求める。

注意すべき点の 1 つは、準最適な分布系列 $\hat{\boldsymbol{m}}$ を求める際に、sklearn.mixture.GMM の predict メ

ソッドを用いているということである。$\hat{m}_t$ は t 番目のフレームにおける、最も事後確率が高いガウス分布のインデックスと同義なので、predict メソッドが返すラベルをそのまま用いることができるのだ。

変換処理

```
    def convert(self, src):
        T, D = src.shape[0], src.shape[1] / 2
        W = self.__construct_weight_matrix(T, D)

        # 準最適な分布系列を求める
        optimum_mix = self.px.predict(src)

        # 行列 E を用意する
        E = np.zeros((T, D * 2))
        for t in range(T):
            m = optimum_mix[t]
            # フレームごとに Et を代入する
            xx = np.linalg.solve(self.covarXX[m], src[t] - self.src_means[m])
            E[t] = self.tgt_means[m] + np.dot(self.covarYX[m], xx)
        E = E.flatten()

        # コンストラクタで計算した self.D を元に D^-1 の対角要素を計算する
        D_inv = np.zeros((T, D * 2, D * 2))
        for t in range(T):
            m = optimum_mix[t]
            # フレームごとに分布系列に対応する self.D の逆行列を代入する
            D_inv[t] = np.linalg.inv(self.D[m])
        # 計算した要素を対角成分とする
        D_inv = scipy.sparse.block_diag(D_inv, format = 'csr')

        # 計算した行列を用いて変換後の特徴量を求める
        mutual = W.T.dot(D_inv)
        covar = mutual.dot(W)
        mean = mutual.dot(E)
        # numpy.linalg.solve に対応する疎行列向けのメソッドを使う
        y = scipy.sparse.linalg.spsolve(covar, mean, use_umfpack = False)

        return y.reshape((T, D))
```

以上で変換処理が実装できる。念のため、ほぼ情報量はないが、TrajectoryGMMMap の実装全体を載せておく。

リスト5.2　trajectory.py

```
 1: #!/usr/bin/python
 2: # coding: utf-8
 3:
 4: import numpy as np
 5:
 6: from sklearn.mixture import GMM
 7:
 8: import scipy.sparse
 9: import scipy.sparse.linalg
10:
11: from gmmmap import GMMMap
12:
13: class TrajectoryGMMMap(GMMMap):
14:     def __init__(self, gmm, swap = False):
15:         <省略>
16:
17:     def __construct_weight_matrix(self, T, D):
```

```
18:         < 省略 >
19:
20:     def convert(self, src):
21:         < 省略 >
```

STF ファイルへの変換

変換処理を呼び出し、結果を STF ファイルに保存するスクリプトは前章とほとんど変わらない。MFCC の動的変化量を結合するのと、フレームごとに呼び出していた変換処理をまとめて呼び出すように変更するのみである。

以下に、前章のスクリプトとトラジェクトリベースな変換のためのスクリプトの diff を載せておく。

リスト5.3　convert_gmmmap.pyとconvert_trajectory.pyのdiff

```
 1: 9c9
 2: < from gmmmap import GMMMap
 3: ---
 4: > from trajectory import TrajectoryGMMMap
 5: 38c38
 6: <     source_data = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
 7: ---
 8: >     source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
 9: 39a40
10: >     source_data = numpy.hstack([source_mfcc, mfcc.delta(source_mfcc)])
11: 41,42c42
12: <     output_mfcc = numpy.array([gmmmap.convert(source_data[frame])[0] \
13: <                                         for frame in xrange(source_data.shape[0])])
14: ---
15: >     output_mfcc = gmmmap.convert(source_data)
```

5.3　トラジェクトリベースな変換処理の結果

以下の画像は、トラジェクトリベースな変換の結果を表したグラフである。前章と同様に、それぞれのグラフは MFCC の第 1 次係数 (数値が小さい方) 及び第 2 次係数 (数値が大きい方) の推移を表しており、上から、変換先話者のデータ、トラジェクトリベースな変換を行ったデータ、GMM によるフレームごとに独立な変換処理を行ったデータを DTW によって伸縮させたものである。中段のグラフの方が、下段よりなめらかに推移し、より上段の変換先話者のデータに近づいていることが分かるだろう。

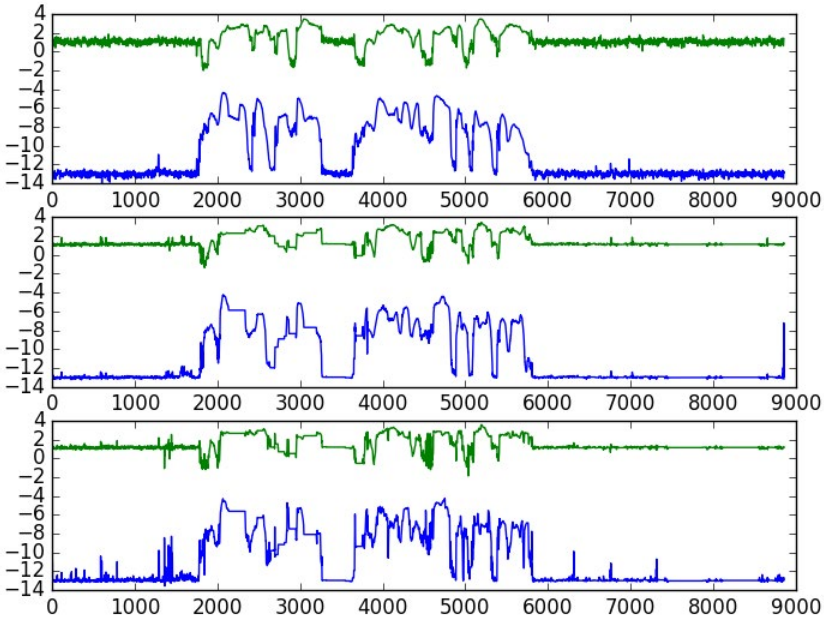

図5.2　トラジェクトリベースな変換とフレームごとに独立な変換による結果データの比較

このように、前後フレームとのつながりを考えた変換手法を採用することで、より自然な変換音声を生成することができる。

6　固有声に基づく多対多声質変換

ここまで、変換元と変換先の話者のパラレルデータが存在していることを前提に、一対一の変換モデルを学習していく仕組みを紹介してきたが、ここからは、固有声 (Eigenvoice) という概念に基づいた、変換元と変換先の話者のパラレルデータがなくても変換できるような仕組みを紹介し、実装していく。まずは、一対多の変換、つまりある特定の人の声質を任意の相手の声質に変換できる仕組みについて説明した後に、多対一の変換、つまり任意の人の声質を特定の相手の声質に変換できる仕組みについて説明し、その 2 つを組み合わせることで多対多の変換を実現するものとする。

固有声による声質変換については「Eigenvoice Conversion Based on Gaussian Mixture Model[8]」を参照した。

6.1　固有声の導入

固有声とは、顔画像認識で用いられている固有顔という概念を元にしたものある。固有顔とは、顔画像を主成分分析した際に得られる固有ベクトルを指し、様々な顔画像の、この固有ベクトルで

形成される部分空間において類似度を取ることによって、低計算量かつ高精度で顔認識を実現することができる。これと同様に、声質変換においても、主成分分析を導入することによって一対多及び多対一の変換を実現する。ただし、ここで主成分分析の対象とするのは、音声の特徴量ではなく、GMM の平均ベクトルであるという大きな違いがある。

ここからは、まず一対多の声質変換について説明する。一対多の場合は、モデル構築のために変換元の話者と第三者 (以降、「事前学習用出力話者」という) のパラレルデータが多数用意されている状況を仮定する。このとき、変換元話者と事前学習用出力話者のそれぞれとの間で、共通の重みと分散共分散行列を用いるという制約の下、GMM に学習させて平均ベクトルを求めておく。そして、この平均ベクトルの出力成分に対して主成分分析をすることで固有ベクトル、つまり、いわゆる固有声を求めることができる。すると、任意の話者に対しても、この固有声空間に射影してやれば、変換に用いる平均ベクトルを求められるようになる。

つまり、変換元話者と s 番目の事前学習用出力話者とのパラレルデータから学習した GMM の i 番目のガウス分布における平均ベクトルを $\boldsymbol{\mu}_i(s) = [\boldsymbol{\mu}_i^{(x)}(s)^\top, \boldsymbol{\mu}_i^{(y)}(s)^\top]^\top$ とすると、$\boldsymbol{\mu}_i^{(y)}(s)$ に対して主成分分析を行うことで以下のように表すことができる。

$$\boldsymbol{\mu}_i^{(y)}(s) \simeq \boldsymbol{B}_i \boldsymbol{w}^{(s)} + \boldsymbol{b}_i^{(0)}$$

ここで、$\boldsymbol{b}_i^{(0)}$ がバイアスベクトル、$\boldsymbol{B}_i = [\boldsymbol{b}_{i,1}, \boldsymbol{b}_{i,2}, ..., \boldsymbol{b}_{i,J}]$ が固有ベクトルである。この $\boldsymbol{B}_i$ で張られる部分空間においては、$\boldsymbol{\mu}_i^{(y)}(s)$ は J 次元の重みベクトル $\boldsymbol{w}_i^{(s)} = [w_{i,1}, w_{i,2}, ..., w_{i,J}]^\top$ によって表される。そして、変換先話者に対応する重みベクトル $\boldsymbol{w}^{(tar)}$ を推定することができれば、平均ベクトル $\boldsymbol{\mu}_i(tar)$ を得ることができ、GMM を用いて特徴量の変換ができる。

ここまでの説明を図で表すと以下の通りとなる。これを見ると、固有声に基づく変換は事前学習処理と話者適応処理の 2 つからなることが分かるだろう。

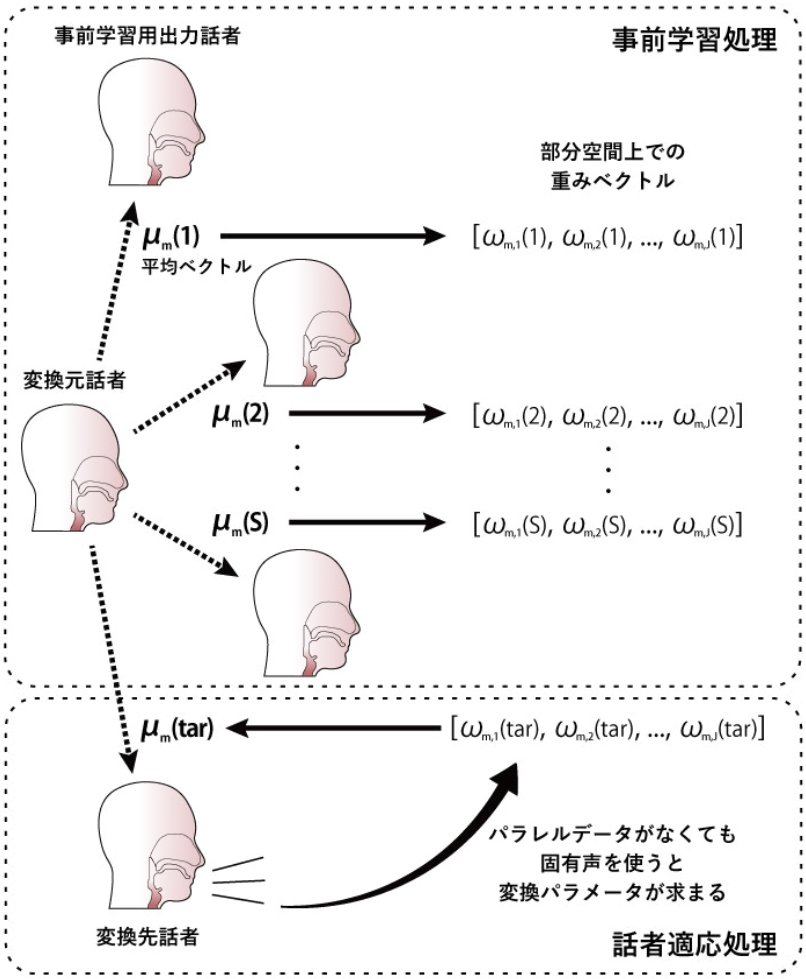

図6.1　固有声を用いた変換の仕組み

以上が、一対多の声質変換の基本的な仕組みである。多対一の場合は、GMM の学習の際に変換元話者と事前学習用出力話者を入れ替えた上で、平均ベクトルの入力成分に対して主成分分析してやれば同様に実現することができる。

6.2　固有声による声質変換の事前学習

まず、固有声による声質変換 (以降、「固有声 GMM」という) の事前学習処理について説明する。事前学習処理は、以下の 3 つのステップからなる。

- すべての事前学習用出力話者とのGMMで共通に用いられる重みと分散共分散行列の推定
- それぞれの事前学習用出力話者に対する平均ベクトルの推定
- 主成分分析による固有ベクトルとバイアスベクトルの決定

具体的な説明にあたって、変換元話者の特徴量を $\boldsymbol{X}_t$、s 番目の事前学習用出力話者の特徴量を

$\boldsymbol{Y}_t^{(s)}$、それを結合したものを $\boldsymbol{Z}_t^{(s)} = [\boldsymbol{X}_t^\top, \boldsymbol{Y}_t^{(s)\top}]^\top$ とし、事前学習用出力話者は S 人いるものとする。

まず、重みと分散共分散行列の推定を行う。この際は、すべての事前学習用出力話者の特徴量に対して尤度が最大となるように学習してやればよい。つまり、推定されるパラメータを $\boldsymbol{\lambda}^{(0)}$ とすると、以下の通りである。

$$\hat{\boldsymbol{\lambda}}^{(0)} = \underset{\boldsymbol{\lambda}}{argmax} \prod_{s=1}^{S} \prod_{t=1}^{T} P(\boldsymbol{Z}_t^{(s)} | \boldsymbol{\lambda})$$

ここで、T は学習データのフレーム数を表している。

次に、それぞれの事前学習用出力話者に対する平均ベクトルの推定を行う。ここで注意すべきなのは、$\boldsymbol{\lambda}^{(0)}$ の重みと分散共分散行列はそのままで、平均ベクトルのみ更新するという点である。

$$\hat{\boldsymbol{\lambda}}^{(s)} = \underset{\boldsymbol{\lambda}}{argmax} \prod_{t=1}^{T} P(\boldsymbol{Z}_t^{(s)} | \boldsymbol{\lambda}^{(s)})$$

最後に、主成分分析によって、固有ベクトルとバイアスベクトルの推定を行う。まず、それぞれの事前学習用出力話者に対して $\boldsymbol{\lambda}^{(s)}$ の出力平均ベクトルを $\boldsymbol{\mu}_i^{(Y)}(s)$ として、$2DM$ 次元のスーパーベクトル $SV^{(s)} = [\boldsymbol{\mu}_1^{(Y)}(s)^\top, \boldsymbol{\mu}_2^{(Y)}(s)^\top, ..., \boldsymbol{\mu}_M^{(Y)}(s)^\top]^\top$ を求める。そして、全出力話者のスーパーベクトルに対して主成分分析を行うことによって、バイアスベクトル $\boldsymbol{b}_i^{(0)}$ 及び固有ベクトル $\boldsymbol{B}_i$ を決定する。このとき、$SV^{(s)}$ は以下のように表すことができる。

$$SV^{(s)} \simeq [\boldsymbol{B}_1^\top, \boldsymbol{B}_2^\top, ..., \boldsymbol{B}_M^\top]^\top \boldsymbol{w}^{(s)} + [\boldsymbol{b}_1^{(0)}, \boldsymbol{b}_2^{(0)}, ..., \boldsymbol{b}_M^{(0)}]$$

$$where\ \boldsymbol{b}_i^{(0)} = \frac{1}{S} \sum_{s=1}^{S} \boldsymbol{\mu}_i^{(Y)}(s)$$

学習データの構築処理

固有声 GMM についても、特徴量を受け取って変換処理を担うクラスと、与えられた STF ファイルのリストから特徴量を抽出して学習データを構築し、事前学習処理を呼び出す部分、そして保存された学習済みインスタンスを読み込んで変換処理を呼び出し、結果を STF ファイルとして保存する部分の 3 つに分けて実装する。

まずは、学習データの構築と学習処理の呼び出しを実装する。今回は、一対多及び多対一の両方に対応できるようにする。はじめに、変換元話者と事前学習用出力話者のリストを受け取って、一対多の学習データを構築する関数を実装する。事前学習用出力話者の数だけ繰り返している以外はこれまでの学習処理と大きな差はないはずである。

一対多声質変換のための学習データの構築

```
def one_to_many(source_list, target_list, dtw_cache):
    source_mfcc = []

    for i in xrange(len(source_list)):
        source = STF()
        source.loadfile(source_list[i])

        mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = D)
        source_mfcc.append(numpy.array([mfcc.mfcc(source.SPEC[frame]) \
                                    for frame in xrange(source.SPEC.shape[0])]))

    total_data = []

    for i in xrange(len(target_list)):
        learn_data = None

        for j in xrange(len(target_list[i])):
            print i, j

            target = STF()
            target.loadfile(target_list[i][j])

            mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency, dimension = D)
            target_mfcc = numpy.array([mfcc.mfcc(target.SPEC[frame]) \
                                       for frame in xrange(target.SPEC.shape[0])])

            cache_path = os.path.join(dtw_cache, '%s_%s.dtw' % \
                tuple(map(lambda x: re.sub('[./]', '_', re.sub('^[./]*', '', x)), \
                                                [source_list[j], target_list[i][j]])))
            if os.path.exists(cache_path):
                dtw = pickle.load(open(cache_path))
            else:
                dtw = DTW(source_mfcc[j], target_mfcc, \
                    window = abs(source_mfcc[j].shape[0] - target_mfcc.shape[0]) * 2)
                with open(cache_path, 'wb') as output:
                    pickle.dump(dtw, output)

            warp_data = dtw.align(target_mfcc, reverse = True)

            data = numpy.hstack([source_mfcc[j], warp_data])
            if learn_data is None:
                learn_data = data
            else:
                learn_data = numpy.vstack([learn_data, data])

        total_data.append(learn_data)

    return total_data
```

次に、多対一の場合の処理を実装する。

多対一声質変換のための学習データの構築

```
def many_to_one(source_list, target_list, dtw_cache):
    target_mfcc = []

    for i in xrange(len(target_list)):
        target = STF()
        target.loadfile(target_list[i])

        mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency, dimension = D)
        target_mfcc.append(numpy.array([mfcc.mfcc(target.SPEC[frame]) \
                                    for frame in xrange(target.SPEC.shape[0])]))

    total_data = []
```

```
    for i in xrange(len(source_list)):
        learn_data = None

        for j in xrange(len(source_list[i])):
            print i, j

            source = STF()
            source.loadfile(source_list[i][j])

            mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = D)
            source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
                                       for frame in xrange(source.SPEC.shape[0])])

            cache_path = os.path.join(sys.argv[3], '%s_%s.dtw' % \
                tuple(map(lambda x: re.sub('[./]', '_', re.sub('^[./]*', '', x)), \
                                                [source_list[i][j], target_list[j]])))
            if os.path.exists(cache_path):
                dtw = pickle.load(open(cache_path))
            else:
                dtw = DTW(source_mfcc, target_mfcc[j], \
                    window = abs(source_mfcc.shape[0] - target_mfcc[j].shape[0]) * 2)
                with open(cache_path, 'wb') as output:
                    pickle.dump(dtw, output)

            warp_data = dtw.align(source_mfcc)

            data = numpy.hstack([warp_data, target_mfcc[j]])
            if learn_data is None:
                learn_data = data
            else:
                learn_data = numpy.vstack([learn_data, data])

        total_data.append(learn_data)

    return total_data
```

最後に、実行時引数として与えられた話者データのリストから、一対多か多対一かどうかを判定し、学習データを構築した後に EVGMM クラスを生成する部分を実装する。学習データの構築の部分と合わせると、以下のようになる。

リスト6.1 learn_evgmm.py

```
 1: #!/usr/bin/env python
 2: # coding: utf-8
 3:
 4: from stf import STF
 5: from mfcc import MFCC
 6: from dtw import DTW
 7: from evgmm import EVGMM
 8:
 9: import numpy
10: import os
11: import pickle
12: import re
13: import sys
14:
15: D = 16
16:
17: def one_to_many(source_list, target_list, dtw_cache):
18:     <省略>
19:
20: def many_to_one(source_list, target_list, dtw_cache):
```

```
    <省略>

if __name__ == '__main__':
    if len(sys.argv) < 5:
        print 'Usage: %s [list of source stf] [list of target] ' + \
                    '[dtw cache directory] [output file]' % sys.argv[0]
        sys.exit()

    source_list = open(sys.argv[1]).read().strip().split('\n')
    target_list = open(sys.argv[2]).read().strip().split('\n')

    if len(filter(lambda s: not s.endswith('.stf'), source_list)) == 0:
        target_list = [open(target).read().strip().split('\n') \
                                                for target in target_list]
        total_data = one_to_many(source_list, target_list, sys.argv[3])
        evgmm = EVGMM(total_data)
    elif len(filter(lambda s: not s.endswith('.stf'), target_list)) == 0:
        source_list = [open(source).read().strip().split('\n') \
                                                for source in source_list]
        total_data = many_to_one(source_list, target_list, sys.argv[3])
        evgmm = EVGMM(total_data, True)

    with open(sys.argv[4], 'wb') as output:
        pickle.dump(evgmm, output)
```

事前学習処理の実装

続いて、変換のためのクラス EVGMM を実装する。このコンストラクタによって事前学習を行う。

ここで注意すべきなのは、それぞれの事前学習用出力話者に対する平均ベクトルを推定する際に、平均ベクトルのみを更新するよう GMM のコンストラクタに init_params = "と params = 'm'を指定しているという点である。init_params = "とすることで、初期化の際にすでに代入した $\boldsymbol{\lambda}^{(0)}$ のパラメータを上書きしないように設定し、params = 'm'とすることで、学習の際に平均ベクトルのみを更新するように設定することができるのである。

また、主成分分析には sklearn.decomposition.PCA を用いている。

固有声GMMの事前学習処理

```
    # 学習データは [[Xt, Yt(1)], [Xt, Yt(2)], ..., [Xt, Yt(S)]] の S 個の要素を持つリスト
    def __init__(self, learn_data, swap = False):
        S = len(learn_data)
        D = learn_data[0].shape[1] / 2

        # すべての学習データについてパラメータを推定する
        initial_gmm = GMM(n_components = M, covariance_type = 'full')
        initial_gmm.fit(np.vstack(learn_data))

        # λ (0) から得たパラメータを保存しておく
        self.weights = initial_gmm.weights_
        self.source_means = initial_gmm.means_[:, :D]
        self.target_means = initial_gmm.means_[:, D:]
        self.covarXX = initial_gmm.covars_[:, :D, :D]
        self.covarXY = initial_gmm.covars_[:, :D, D:]
        self.covarYX = initial_gmm.covars_[:, D:, :D]
        self.covarYY = initial_gmm.covars_[:, D:, D:]

        # スーパーベクトルはすべての出力話者についてまとめて S * 2DM 次元の行列とする
        sv = []
```

```
        # 各出力話者について平均ベクトルを推定する
        for i in xrange(S):
            # 平均ベクトル以外は更新しないように設定する
            gmm = GMM(n_components = M, params = 'm', init_params = '', \
                                                covariance_type = 'full')
            gmm.weights_ = initial_gmm.weights_
            gmm.means_ = initial_gmm.means_
            gmm.covars_ = initial_gmm.covars_
            gmm.fit(learn_data[i])

            # 平均ベクトルを結合したスーパーベクトルを更新する
            sv.append(gmm.means_)

        sv = np.array(sv)

        # スーパーベクトルの入力平均ベクトルにあたる部分を主成分分析にかける
        source_pca = PCA()
        source_pca.fit(sv[:, :, :D].reshape((S, M * D)))

        # スーパーベクトルの出力平均ベクトルにあたる部分を主成分分析にかける
        target_pca = PCA()
        target_pca.fit(sv[:, :, D:].reshape((S, M * D)))

        # 入力平均ベクトルと出力平均ベクトルに対する固有ベクトルのタプル
        self.eigenvectors = source_pca.components_.reshape((M, D, S)), \
                                target_pca.components_.reshape((M, D, S))
        # 入力平均ベクトルと出力平均ベクトルに対するバイアスベクトルのタプル
        self.biasvectors = source_pca.mean_.reshape((M, D)), \
                                target_pca.mean_.reshape((M, D))

        # 話者適応の際に更新するようの平均ベクトルの変数を用意しておく
        self.fitted_source = self.source_means
        self.fitted_target = self.target_means

        self.swap = swap
```

6.3 固有声による声質変換の話者適応処理

固有声 GMM における話者適応処理は、変換先話者の特徴量から固有声の部分空間における重みベクトル $\boldsymbol{w}^{(tar)}$ を求めることによって行われる。つまり、話者適応によって得られる GMM の変換パラメータを $\boldsymbol{\lambda}^{(tar)}$ とすると、その出力平均ベクトル $\boldsymbol{\mu}_i^{(X)}(tar)$ は以下のように表すことができる。

$$\boldsymbol{\mu}_i^{(X)}(tar) = \boldsymbol{B}_i w_i^{(tar)} + \boldsymbol{b}_i^{(0)}$$

このとき、変換先話者の特徴量を $\boldsymbol{Y}^{(tar)}$ として、求める重みベクトル $\boldsymbol{w}^{(tar)}$ は以下のように表される。

$$\begin{aligned}\hat{\boldsymbol{w}}^{(tar)} &= \underset{\boldsymbol{w}}{argmax} \int P([\boldsymbol{X}^\top, {\boldsymbol{Y}^{(tar)}}^\top]^\top, \boldsymbol{\lambda}^{(tar)}) d\boldsymbol{X} \\ &= \underset{\boldsymbol{w}}{argmax}\ P(\boldsymbol{Y}^{(tar)}|\boldsymbol{\lambda}^{(tar)})\end{aligned}$$

ここで、確率密度関数 $P(\boldsymbol{Y}^{(tar)}|\boldsymbol{\lambda}^{(tar)})$ は GMM でモデル化されているので、EM アルゴリズ

ムに基づき、以下の Q 関数を最大化することで求められる。

$$\mathcal{Q}(\boldsymbol{w}^{(tar)}, \hat{\boldsymbol{w}}^{(tar)}) = \sum_{all\ \boldsymbol{m}} P(\boldsymbol{m}|\boldsymbol{Y}^{(tar)}, \boldsymbol{\lambda}^{(tar)})\ log\ P(\boldsymbol{Y}^{(tar)}, \boldsymbol{m}|\hat{\boldsymbol{\lambda}}^{(tar)})$$

このとき、$\hat{\boldsymbol{w}}^{(tar)}$ は以下のように求められる。

$$\hat{\boldsymbol{w}}^{(tar)} = \left\{ \sum_{i=1}^{M} \overline{\gamma}_i^{(tar)} \boldsymbol{B}_i^{\top} {\boldsymbol{\Sigma}_i^{(yy)}}^{-1} \boldsymbol{B}_i \right\}^{-1} \sum_{m=1}^{M} \boldsymbol{B}_i^{\top} {\boldsymbol{\Sigma}_i^{(yy)}}^{-1} \overline{\boldsymbol{Y}}_i^{(tar)}$$

ただし、$\overline{\gamma}_i^{(tar)}, \overline{\boldsymbol{Y}}_i^{(tar)}$ は以下の通りである。

$$\overline{\gamma}_i^{(tar)} = \sum_{t=1}^{T} P(m_i|\boldsymbol{Y}_t^{(tar)}, \boldsymbol{\lambda}^{(tar)})$$

$$\overline{\boldsymbol{Y}}_i^{(tar)} = \sum_{t=1}^{T} P(m_i|\boldsymbol{Y}_t^{(tar)}, \boldsymbol{\lambda}^{(tar)})(\boldsymbol{Y}_t^{(tar)} - \boldsymbol{b}_i^{(0)})$$

以上を繰り返すことによって、$\hat{\boldsymbol{w}}^{(tar)}$ が求められる。ただし、$\boldsymbol{\lambda}^{(tar)}$ の初期パラメータとしては $\boldsymbol{\lambda}^{(0)}$ を用いるものとする。

話者適応処理の実装

話者適応処理は EVGMM のメソッドとして実装し、コンストラクタで求めた事前学習パラメータと引数として与えられる特徴量から適応済みパラメータを求めることとする。まずは、一対多の声質変換に関する適応処理を実装する。

一対多の声質変換に関する話者適応処理

```
    def fit_target(self, target, epoch):
        # P(m|Y) を算出するための GMM インスタンスを生成する
        py = GMM(n_components = M, covariance_type = 'full')
        py.weights_ = self.weights
        py.means_ = self.target_means
        py.covars_ = self.covarYY

        for x in xrange(epoch):
            # P(m|Y) を算出する
            predict = py.predict_proba(np.atleast_2d(target))
            # Y を算出する
            y = np.sum([predict[:, i: i + 1] * (target - self.biasvectors[1][i]) \
                                                    for i in xrange(M)], axis = 1)
            # γを算出する
            gamma = np.sum(predict, axis = 0)

            # 重みベクトル w を算出する
            left = np.sum([gamma[i] * np.dot(self.eigenvectors[1][i].T, \
                        np.linalg.solve(py.covars_, self.eigenvectors[1])[i]) \
                                                    for i in xrange(M)], axis = 0)
            right = np.sum([np.dot(self.eigenvectors[1][i].T, \
                    np.linalg.solve(py.covars_, y)[i]) for i in xrange(M)], axis = 0)
            weight = np.linalg.solve(left, right)
```

```
            # 重みベクトルから平均出力ベクトルを求め、GMM のパラメータを更新する
            self.fitted_target = np.dot(self.eigenvectors[1], weight) \
                                                            + self.biasvectors[1]
            py.means_ = self.fitted_target
```

同様にして、多対一の声質変換に関する適応処理を実装する。用いる固有ベクトルとバイアスベクトルを平均入力ベクトルによるものにすればよい。

多対一の声質変換に関する話者適応処理

```
    def fit_source(self, source, epoch):
        # P(m|X) を算出するための GMM インスタンスを生成する
        px = GMM(n_components = M, covariance_type = 'full')
        px.weights_ = self.weights
        px.means_ = self.source_means
        px.covars_ = self.covarXX

        for x in xrange(epoch):
            # P(m|X) を算出する
            predict = px.predict_proba(np.atleast_2d(source))
            # X を算出する
            x = np.sum([predict[:, i: i + 1] * (source - self.biasvectors[0][i]) \
                                                    for i in xrange(M)], axis = 1)
            # γ を算出する
            gamma = np.sum(predict, axis = 0)

            # 重みベクトル w を算出する
            left = np.sum([gamma[i] * np.dot(self.eigenvectors[0][i].T, \
                        np.linalg.solve(px.covars_, self.eigenvectors[0])[i]) \
                                                    for i in xrange(M)], axis = 0)
            right = np.sum([np.dot(self.eigenvectors[0][i].T, \
                    np.linalg.solve(px.covars_, x)[i]) for i in xrange(M)], axis = 0)
            weight = np.linalg.solve(left, right)

            # 重みベクトルから平均入力ベクトルを求め、GMM のパラメータを更新する
            self.fitted_source = np.dot(self.eigenvectors[0], weight) \
                                                            + self.biasvectors[0]
            px.means_ = self.fitted_source
```

6.4 固有声による声質変換処理

ここまでで、変換に必要なパラメータをすべて求めることができたので、変換処理を実装する。ここで注意したいのは、EVGMM で新たに実装した処理は変換のパラメータを求めるための処理であり、パラメータが求まった後の変換処理はこれまでと全く変わらないという点である。変換処理は、一対一の声質変換と同様に実装すればよい。

変換処理の実装

変換処理の実装は、第 2 章で紹介した実装とほとんど同じである。ただし、$\boldsymbol{E}_{m,t}$ を求める際に、平均ベクトルとして話者適応処理で得られたものを使う必要があるので、その部分のみ変更を加えている。

ここまでの実装と合わせて、EVGMM クラス全体を載せておく。

EVGMMの実装

```
class EVGMM(object):
```

```
    def __init__(self, learn_data, swap = False):
        <省略>

    def fit(self, data, epoch = 1000):
        if self.swap:
            self.fit_source(data, epoch)
        else:
            self.fit_target(data, epoch)

    def fit_source(self, source, epoch):
        <省略>

    def fit_target(self, target, epoch):
        <省略>

    def convert(self, source):
        D = source.shape[0]

        # 話者適応処理で得たパラメータからEを算出する
        E = np.zeros((M, D))
        for m in xrange(M):
            xx = np.linalg.solve(self.covarXX[m], source - self.fitted_source[m])
            E[m] = self.fitted_target[m] + np.dot(self.covarYX[m], xx)

        px = GMM(n_components = M, covariance_type = 'full')
        px.weights_ = self.weights
        px.means_ = self.source_means
        px.covars_ = self.covarXX

        posterior = px.predict_proba(np.atleast_2d(source))
        return np.dot(posterior, E)
```

トラジェクトリベースの変換処理の実装

先ほど説明したように、EVGMMにおける変換処理は第2章で扱った変換手法とほとんど同じで、相違点は、平均ベクトルの代わりに話者適応で得られたパラメータを用いているのみである。つまり、学習や変換に用いる特徴量に動的特徴量を結合してやれば、全く同じようにトラジェクトリベースの声質変換を適用することができる。

以下に、EVGMMクラスを継承して、トラジェクトリベースの声質変換を行うTrajectoryEVGMMクラスの実装を載せる。ただし、convertメソッドは$\boldsymbol{E}_{m,t}$の導出以外は前章と変わらないので、該当部分以外は省略している。

EVGMMとTrajectoryGMMの実装を合わせたevgmm.pyは以下の通りである。

リスト6.2　evgmm.py

```
 1: #!/usr/bin/env python
 2: # coding: utf-8
 3:
 4: import numpy as np
 5:
 6: from sklearn.decomposition import PCA
 7: from sklearn.mixture import GMM
 8:
 9: import scipy.sparse
10: import scipy.sparse.linalg
11:
12: M = 32
```

```
13:
14: class EVGMM(object):
15:     <省略>
16:
17: class TrajectoryEVGMM(EVGMM):
18:     def __construct_weight_matrix(self, T, D):
19:         <省略>
20:
21:     def convert(self, src):
22:         <省略>
23:
24:         # 話者適応処理で得たパラメータからEを算出する
25:         E = np.zeros((T, D * 2))
26:         for t in range(T):
27:             m = optimum_mix[t]
28:             xx = np.linalg.solve(self.covarXX[m], src[t] - self.fitted_source[m])
29:             E[t] = self.fitted_target[m] + np.dot(self.covarYX[m], xx)
30:         E = E.flatten()
31:
32:         <省略>
33:         return y.reshape((T, D))
```

最後に、変換先話者のSTFファイルを読み込んで話者適応をした後に、変換元話者の読み込んで変換処理を行い、結果をSTFファイルとして保存するという部分を実装する。ここでは、一対多か多対一かを実行時引数の数で判定している。というのも、一対多の場合は話者適応のために変換先話者の特徴量が必要だが、多対一の場合は変換に用いる変換元話者の特徴量で話者適応も行うことができるからである。

リスト6.3　convert_trajevgmm.py

```
 1: #!/usr/bin/env python
 2:
 3: import math
 4: import numpy
 5: import pickle
 6: import sklearn
 7: import sys
 8:
 9: from evgmm import GMM
10:
11: from stf import STF
12: from mfcc import MFCC
13: from dtw import DTW
14:
15: D = 16
16:
17: if __name__ == '__main__':
18:     if len(sys.argv) < 5:
19:         print 'Usage: %s [gmmmap] [f0] [source speaker stf] ' + \
20:                                 (target speaker stf) [output]' % sys.argv[0]
21:         sys.exit()
22:
23:     with open(sys.argv[1], 'rb') as infile:
24:         evgmm = pickle.load(infile)
25:
26:     with open(sys.argv[2], 'rb') as infile:
27:         f0 = pickle.load(infile)
28:
29:     source = STF()
30:     source.loadfile(sys.argv[3])
31:
32:     mfcc = MFCC(source.SPEC.shape[1] * 2, source.frequency, dimension = D)
```

```
33:     source_mfcc = numpy.array([mfcc.mfcc(source.SPEC[frame]) \
34:                                    for frame in xrange(source.SPEC.shape[0])])
35:     source_data = numpy.hstack([source_mfcc, mfcc.delta(source_mfcc)])
36:
37:     if len(sys.argv) == 5:
38:         evgmm.fit(source_data)
39:     else:
40:         target = STF()
41:         target.loadfile(sys.argv[4])
42:
43:         mfcc = MFCC(target.SPEC.shape[1] * 2, target.frequency, dimension = D)
44:         target_mfcc = numpy.array([mfcc.mfcc(target.SPEC[frame]) \
45:                                    for frame in xrange(target.SPEC.shape[0])])
46:         target_data = numpy.hstack([target_mfcc, mfcc.delta(target_mfcc)])
47:
48:         evgmm.fit(target_data)
49:
50:     output_mfcc = evgmm.convert(source_data)
51:     output_spec = numpy.array([mfcc.imfcc(output_mfcc[frame]) \
52:                                    for frame in xrange(output_mfcc.shape[0])])
53:
54:     source.SPEC = output_spec
55:     source.F0[source.F0 != 0] = numpy.exp((numpy.log(source.F0[source.F0 != 0]) - \
56:                                        f0[0][0]) * f0[1][1] / f0[1][0] + f0[0][1])
57:
58:     if len(sys.argv) == 5:
59:         source.savefile(sys.argv[4])
60:     else:
61:         source.savefile(sys.argv[5])
```

6.5 固有声に基づく多対多声質変換

ここまでで扱った多対一と一対多の声質変換を組み合わせることによって、多対多の声質変換を実現することができる。つまり、任意の話者の発話データを多対一変換にて特定の話者に変換し、一対多変換によってその特定の話者から任意の話者へと変換することができる。この仕組みを図で表すと以下のようになる。

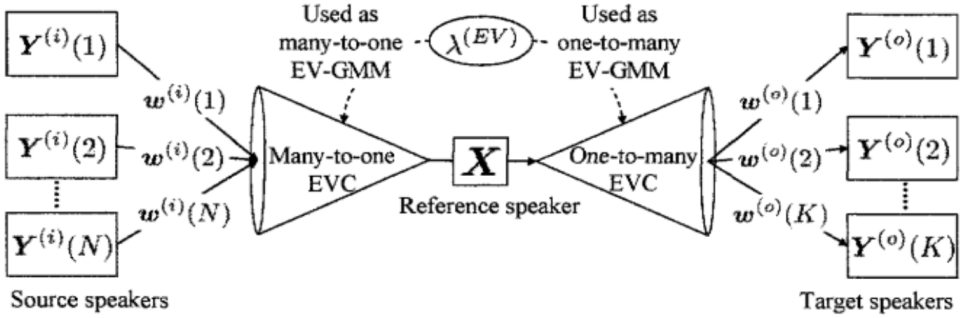

図6.2　多対一変換及び一対多変換を組み合わせた多対多声質変換の仕組み([9]より引用)

紙面と締め切りの都合上、詳細な説明は割愛するが、詳しくは「Many-to-many eigenvoice conversion with reference voice[9]」という論文を参照してほしい。

6.6 固有声に基づく声質変換の変換処理の結果

以下の画像は、固有声に基づく多対一の声質変換の結果を表したグラフである。前章と同様に、それぞれのグラフは MFCC の第 1 次係数（数値が小さいほう）及び第 2 次係数（数値が大きいほう）の推移を表しており、上から、変換先話者のデータ、変換元話者と変換先話者のパラレルデータを用いずに固有声に基づく多対一変換を行ったデータ、パラレルデータを用いて一対一変換を行ったデータを DTW によって伸縮させたものである。

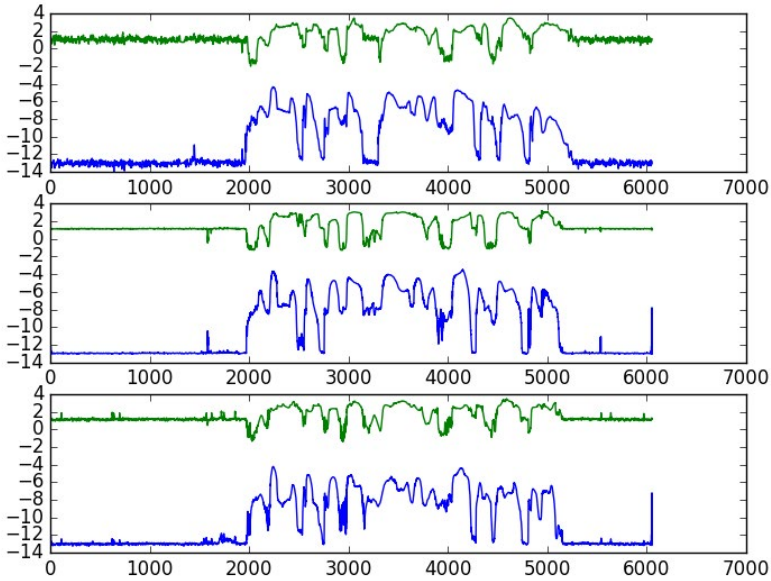

図6.3 多対一の声質変換処理と一対一の声質変換処理の結果データの比較

ここでは、モデルの学習に 11 人の事前学習用入力話者についてそれぞれ 10 文のパラレルデータを用いた。また、話者適応には 1 文のみ、変換元の特徴量をそのまま用いた。図を見るとわかるように、パラレルデータを用いた一対一の声質変換に比べても遜色ない精度で変換出来ていることがわかる。

しかし、MFCC の第 2 次係数を見ると、多対一変換においては推移が滑らかになりすぎていることがわかる。これは過剰な平滑化と呼ばれる現象で、GMM をベースとした声質変換の 1 つの問題点である。これを解決する手法としては、系列内変動 (Global Variance) や変調スペクトル (Modulation Spectrum) といった仕組みが考案されているが、ここでは説明を割愛する。

7 あとがき

送る月日に関守なしという諺の通りどんどん時間は流れる一方で、締め切りという関所は函谷関のようにずっしりと構えたまま破ることもできず、文章での分かりやすい解説というよりも、実装ベースの解説が多くを占めるような形になってしまいました。ただ、私のように、数式で具体的に説明されるよりもソースコードのほうがわかりやすいという人も幾ばくかはいるかと思いますので、そういう人のお役に立てればなと思っています。

個人的に 2015 年 1 月ごろから興味を持って取り組み始め、大学進学後は情報科学特別演習という枠組みの中で、単位のタネとしながら細々と続けていたのですが、とりあえず、解説記事という形でこれまで取り組んできた内容の 1 つを形にすることができました。この場を設けてくれた SunPro 及び @hakatashi に感謝しています。

もちろん、固有声に基づく変換処理の実装がゴールではなく、現在は、声質を好きに調整できる重回帰混合ガウスモデルの実装にも取り組んでいます。（バグって EM アルゴリズムを繰り返していくと分散共分散行列が非対称になったりしていますが）そちらについても何らかの形で面白い Web サービスとかに落とし込めればなと思っています。

思ったより長い記事になってしまいましたが、最後までお読みいただきありがとうございました。

8 参考文献

- [1] H. Kawahara and M. Morise, "Technical foundations of TANDEM-STRAIGHT, a speech analysis, modification and synthesis framework," SADHANA - Academy Proceedings in Engineering Sciences, vol. 36, no. 5 , pp. 713-728, Oct. 2011.
- [2] C.M. ビショップ , " パターン認識と機械学習 ," 丸善出版 , Apr. 2012.
- [3] 杉山 将 , " 統計的機械学習—生成モデルに基づくパターン認識 ," オーム社 , Sep. 2009.
- [4] http://aidiary.hatenablog.com/entry/20120225/1330179868
- [5] 亀岡 弘和 , " 東京大学工学部 講義資料「応用音響学 第 5 回」", May. 2014. http://hil.t.u-tokyo.ac.jp/~kameoka/aa/AA14_05.pdf
- [6] T. Toda, A. W. Black, and K. Tokuda, "Voice conversion based on maximum likelihood estimation of spectral parameter trajectory," IEEE Trans. Audio, Speech, Lang. Process, vol. 15, no. 8, pp. 2222–2235, Nov. 2007.

- [7] http://r9y9.github.io/blog/2014/07/05/statistical-voice-conversion-muzui/
- [8] T. Toda, Y. Ohtani, and K. Shikano, "Eigenvoice conversion based on Gaussian mixture model," Ninth International Conference on Spoken Language Processing. 2006.
- [9] Y. Ohtani, et al. "Many-tomany eigenvoice conversion with reference voice." Proc. INTERSPEECH. 2009.