# 音声で動作するオーディオプレイヤの製作

14EC004 飯田頌平

2016 年 2 月 16 日

## 目次



## 1　序文

音声認識システムは日々進歩している．組込み機器に音声による対話システムを組み込むことはいまや珍しいことではなくなり，それどころか人との自然な会話でさえある程度こなすロボットまでもが登場しているのだ．Siri[*1]などは，音声認識が一般人にも身近な技術となったことの好例である．

そのような時代の中, わずか数インチのコンピュータ Raspberry Pi が発売された．Raspberry Pi は UNIX ベースのディストリビューションに対応しており,Python によって動かすことができる．Python の汎用性は素晴らしく，GPIO ピンの操作から CGI の作成，果ては科学的な計算にも対応しており，フロントエンドにおける大抵のユーザの要求に応える能力を持つ．

この Python と Linux 向けに開発されたソフトウェア（音声認識ソフトや音声合成ソフト）を活用することによって，Raspberry Pi を音声で動かすことができる．また Raspberry Pi は携帯性に優れたサイズであり，Linux 標準の wav ファイル再生コマンドなどを活用すれば，ウォークマンを模したオーディオプレイヤを Raspberry Pi によって再現することもできる．既存の PC はボードのサイズが比較的大きく，ウォークマンのようにポケットに入れることはできなかった．

そうした分野は専ら組込みの分野に独占されており，汎用性を持つ PC でありながらポケットサイズになった Raspberry Pi の登場によって，Siri のような多機能の音声認識デバイスをアマチュアでも気軽に作成することができるようになった．そこで音声認識で動作するオーディオプレイヤを製作しようというのが本レポートの主題である．無論ここでいうオーディオプレイヤとは単なる音楽の再生をするだけのものではなく，iPod のように汎用的な機能を持ったデバイスを指す．

本レポートの第 2 章では構成や制作環境について簡潔にまとめている．また第 3 章では作成するオーディオの使用方法（外部仕様）について示す．製作物の概要について知りたい場合は，ここまでの章を読むだけで良い．

第 4 章では採用したアルゴリズムの説明や，ソースコードの解説を行う．https://github.com/J-Holliday/RaspiAudio 上に置かれたソースコードと併用することで，内部の仕組みについて理解することができるだろう．

最後に第 5 章において，オーディオプレイヤの現時点での性能上の問題や，その改善点について示す．

[*1] Apple 社が開発した iPhone 用音声認識システム

## 2 概要

本レポートで作成するオーディオプレイヤは以下の部品で構成されている．

- Raspberry Pi
- Raspberry Pi 用電源
- 音声入力用マイク
- 音声出力用ヘッドフォン
- 無線子機
- 音声認識サーバ

サーバは外付けの物を利用し，ネットワークを経由して Raspberry Pi と通信する．認識サーバについては Raspberry Pi のバックグラウンドで動作させることも可能であるが，サーバにはある程度の処理能力が要求される．その要求される能力は認識させたい単語の数に比例し，自然言語処理の域まで単語を詳細に分解しようと言うのであれば，Raspberry Pi のスペックでは到底追いつかない．一方数十語程度しか単語を認識させないのであれば，別途サーバを用意する必要はない．

次にプログラムの動作環境を示す．

| | | |
|---|---|---|
| PC | : | Raspberry Pi2 |
| PC の OS | : | Raspbian - Jessie |
| 音声認識エンジン | : | Julius - 4..3.1 |
| 認識用モデル | : | dictation-kit-v4.3.1-linux |
| 認識サーバ | : | 実機または VirtualBox |
| サーバの OS | : | Ubuntu 14.04 |
| フレームワーク | : | CherryPy-3.8.1 |
| 音声合成ソフト | : | aquestalkpi |
| スクリプト | : | Python2, sh |

プログラムを実行するに辺り，事前に以上の動作環境を用意しておく必要がある．冗長かつ本質から逸れるためにインストール手順等については触れないが，Web 上の情報を参考にすれば誰でも無償で用意できるものばかりである．

また一部ソフトウェアの，デフォルトの解凍先ディレクトリについて明示しておく．

/home/pi/workspace/raspi-audio/downloads/

ただし認識用モデルについては，以下のディレクトリに解凍すること．

/home/pi/workspace/raspi-audio/downloads/julius-kits/

そして以下のディレクトリに楽曲ファイルを用意する．

/home/pi/workspace/raspi-audio/music/

楽曲ファイルについては tokyo の branch にある RaspiAudio/music/を参考にするとよい．将来的に楽曲ファイルの追加とファイルサイズの肥大化が見込まれるため現行ブランチからは RaspiAudio/music/を削除しているので，注意すること．

Python のモジュールについても適宜インストールしておかなければならない．ソースコードを参考にして，未インストールのモジュールをインストールする．また使用している Python のバージョンが 3 である場合は，以下のコマンドで Python2 へと一時的にダウングレードできるので，活用すると良い．

ソースコード 1　/bin/sh

```
1 # temporary downgrade
2 $ virtualenv -p /usr/bin/python2.7 --
      distribute temp-python
3 $ source temp-python/bin/activate
4
5 # upgrade again
6 $ deactivate
```

最後にディレクトリ構成について示す．最上位のディレクトリを RaspiAudio とする．オーディオプレイヤとして機能させるためには，RaspiAudio の直下に最低でも以下の構成要素が必要である．

- ASRServer.py
- musicPlayer/
- semanticAnalysys/
- raspisan.py
- voice/
- yukkuri/

branch によっては要件を満たさないものもあるため，fetch または pull したディレクトリについて注意を払わなければならない．master の構成が要件を満たしているとも限らないので，要件を満たす branch に checkout しなければならない．

# 3　外部仕様

この章ではオーディオプレイヤの使用方法について説明する．口頭で命令を述べるだけで動作するのだが，その前準備として幾らかの手順を踏む必用があるため，主にその部分について述べる．

## 3.1　プログラムのインストール

プログラムは github に公開されている．以下のコマンドによって作業用ディレクトリに保存する．

ソースコード 2　/bin/sh

```
1 $ git clone https://github.com/J-Holliday/
    RaspiAudio
```

git コマンドが実行されない場合には，git をインストールする必要がある．Raspberry Pi に git をインストールするには

ソースコード 3　/bin/sh

```
1 $ sudo apt-get install git
```

を実行すればよい．なお，デフォルトの作業用ディレクトリは

/home/pi/workspace/raspi-audio/

である．任意のディレクトリを指定する場合は，ファイルパスの構成に気を付ける必要がある．

clone に成功したら，branch を切り替える必要がある．2016 年 2 月 16 日現在，master では前述の要件を満たしていない．この時点で要件を満たす branch は takao である．よって以下のコマンドで branch を変更する．

ソースコード 4　/bin/sh

```
1 $ git checkout -b takao origin/takao
```

これでプログラムのインストールは完了した．

## 3.2　基本的な使い方

オーディオプレイヤの基本的な使い方は以下の通りである．

1. 音声認識サーバを起動する
2. 実行プログラムを起動する
3. マイクに向かって発話する

音声認識サーバは以下のコマンドで起動する．

ソースコード 5　/bin/sh

```
1 $ python ASRServer.py
```

すると 8000 番ポートが待ち受け状態となり，ここに wav ファイルを POST で送信することで，音声認識サーバたる Julius へ wav ファイルが入力されるようになる．なお，ASRServer.py は sayonari によるスクリプト [1] を改変したものである．

次いで実行プログラムを起動する．

ソースコード 6　/bin/sh

```
1 $ python raspisan.py
```

ヘッドフォンから合成音声による挨拶が聞こえたら，プログラムが発話を待ち受ける状態へと遷移する．試しに「音楽を再生して下さい」などと喋りかけて見ると，Raspberry Pi から音楽が再生されるようになる．

他にも複数の Raspberry Pi が解釈できる命令が搭載されている．命令のうち代表的なものを抜粋し次節に示す．

## 3.3　代表的な命令

当オーディオプレイヤはただの音楽再生機器としての命令だけではなく，linux コンピュータとしての命令をも実行することができる．

- 音楽を再生する
- 音楽を停止する
- 楽曲ファイルを選択する

- シャッフル再生を行う
- ニュースを流す
- メールチェックを行う
- アラームをセットする

命令文には特定の文法規則が必要となるが，ある程度自然な会話でも解釈できるような仕組みを搭載しているため，適当に喋りかけてみれば良い．

# 4　内部仕様

## 4.1　Julius による音声認識

音声認識は Julius によって行われる．Julius についての詳細な説明は開発者である李と河原のドキュメント [2] を参照すれば良いが，動作原理の概要についてもここで触れておく．

Julius は音響モデルと言語モデルとのパターンマッチによって入力音声から言語を抽出する音声認識エンジンである．音響モデルとは音素や音節の周波数のパターンを指し，言語モデルは音素に対応した語句や，語句を文としてまとめたデータを指す．文については正規文法しか処理することが出来ない．

Julius は幾つかの入力方式を持っており，今回は wav ファイルを入力する方式を使用する．単体で使う場合はサーバ側に専用のインタープリタが立ち上がり，クライアントから参照できないため，Python のフレームワークである CherryPy を活用し，クライアントから POST でデータを送信できるようにする．

これらの設定はすべて ASRServer.py で設定している．基となったコードの詳細は製作者のサイト [1] に示されているため，ここではオーディオプレイヤ用に変更した部分について説明する．

JULIUS_HOME および JULIUS_EXEC はそれぞれ前述の解凍先ディレクトリ内を参照している．SERVER_PORT は 8000 番としているが，Python の SimpleHTTPServer などと競合しやすいポート番号であるので，必要に応じて変更すること．ASR_FILEPATH は RaspiAudio/voice を指定している．voice/がない branch で実行すると，無論エラーとなるので，気を付ける．

Julius は Julius_EXEC が指定したスクリプトを subprocess.Popen で子プロセスとして実行することによって実行される．subprocess.Popen は引数のシェルスクリプトを実行するコマンドであり，そこから Julius を実行する際のパラメータの指定方法が伺える．Julius のパラメータはバイナリの後に-C オプションで設定ファイルを指定している．設定ファイルの中身はパラメータが記載されており，ここで音声モデルと言語モデル，辞書ファイル，入出力方式を設定する必要がある．

クライアント側における入出力の遣り取りは raspisan.py 内で行われる．以下に Julius へリクエストを送信しレスポンスを受け取るまでのコードを抜粋して示す．

ソースコード 7　RaspiAudio/raspisan.py

```
1 url = ‘‘http://192.168.100.101:8000/
      asr_julius’’
2 files = {
3   ‘myFile’:␣open(‘voice/record.wav’,’rb’)
4 }
5 s = requests.Session()
6 r = s.post(url, files=files)
```

まずは requests の属性 Session() によってセッションのインスタンスを作成している．ここで名前空間 requests は python のモジュールである．Session() のインスタンス変数 s の属性 post は，第一引数に送信先 URL を，第二引数に送信先データを指定して POST を実行するメソッドである．URL のドメイン（IP）部分には Julius サーバ（CherryPy）のドメイン（IP）及び SERVER_PORT で指定したポート番号を指定する．パスで指定する asr_julius という値は，クラス ASRServer の関数 asr_julius を呼び起こす．

ソースコード 8　RaspiAudio/ASRServer.py

```
1 class ASRServer(object):
2   （省略）
3   def asr_julius(self, myFile):
4     （省略）
```

ASRServer.py のソースを見ると asr_julius の引数が呼び出し元の名前空間と myFile という仮引数であることがわかるが，この仮引数の名前が files で指定した dict 形式のデータ構造のキーに結びついている．そのためリードバイナリ形式で開かれた voice/record.wav という音声ファイルが ASRServer へと渡される．ASRServer は Julius の実行コマンド JULIUS_EXEC を引数に持つインスタンス変数 p をクラス変数として定義しており，asr_julius 内で p を呼ぶことで Julius に音声ファイルを入力している．Julius からの応答は raspisan.py 内の r へと返される．この応答が持つ属性の中に，認識された文章（単語列）が含まれているのである．

## 4.2 認識された文章の解釈

Julius からのレスポンスによって，音声認識の結果の文章を得ることができる．それでは得た文章をどのようにしてプログラムに解釈させればよいのか．

文章解釈プログラムは RaspiAudio/semanticAnalysis/semanticAnalysis.py である．このモジュールは RaspiAudio/raspisan.py から以下のように呼ばれる．

ソースコード 9　RaspiAudio/raspisan.py

```
1 from semanticAnalysis import
      semanticAnalysis as sa
2 （省略）
3 r = s.post(url, files=files)
4 ary = r.text.split("\n")
5 data = ary[0].split(":")
6 words = data[1].strip("␣").split("␣")
7 on = sa.send(words)
```

Julius からのレスポンスをパースして文字列のリストである words に格納し，それを引数に取って semanticAnalysis モジュールの関数 send() を呼ぶ．send() は引数から特徴ベクトルを検出して最適だと思われる命令を選択し，on へと命令のキーを返す．

特徴ベクトルの検出は以下の順で実行される．

1. 単語データと命令データを読み込む
2. 語彙抽出を行い，入力文章の全体の特徴ベクトルを検出する
3. 命令データから命令のモデルを作成する
4. 命令の特徴ベクトルとパターンマッチを行い，最小誤差のモデルを選ぶ
5. モデルから命令のキーを取得し，呼び出し元に返す

単語データの読み込みはソースコード 10 によって行われる．ここでは CSV 形式の単語データを二次元の配列 worditem に格納している．worditem はいずれのクラス・関数にも属さないモジュール直下の変数であり，イミュータブルでありながら参照することが出来る．同様にして命令データを orderitem に束縛している．

ソースコード 10　RaspiAudio/semanticAnalysis/semanticAnalysis.py

```
1 # get data from word.csv
2 f = open("semanticAnalysis/word.csv")
3 res = f.read()
4 f.close()
5 line = res.split("\n")
6 worditem = []
7 for i in range(1,len(line)−1):
8   worditem.append(line[i].split(","))
```

語彙抽出の方法は単純である．文字列リストを拡張 for 文に入れて，各要素に対し worditem の要素と一致するか否かを見ればよい．このとき一致した場合，特徴ベクトルの値を更新する．特徴ベクトルの扱いに関しては，モジュール semanticAnalysis 下のクラス featureVector によって行われている．更新する場合，ベクトルを引数にして featureVector.setVector() を呼ぶ．このときデフォルトで重みを 1 だと決めているのだが，オプションで重みを変更することも出来る．featureVector.setVector() の処理をソースコード 11 に示す．

ソースコード 11　RaspiAudio/semanticAnalysis/semanticAnalysis.py

```
1 def setVector(self, coordinate, weight=1,
      vectype="input"):
2   """set␣feature␣vector␣include␣init."""
3   try:
```

```
    # select vector
    if vectype == "input":
      v = featureVector.fv
    elif vectype == "model":
      v = np.arange(0)
    # init vector
    if len(v) == 0:
      v = featureVector.initVector()
    # set value
    column = coordinate % 10
    row = (coordinate - column)/10
    v[row][column] += weight
    # update
    if vectype == "input":
      featureVector.fv = v
    elif vectype == "model":
      featureVector.model.append(v)
  except:
    print "Exception in setVector."
    print "coordinate:%d, weight=%d" % (
        coordinate, reward)
```

featureVector.setVector() では 10*10 の行列をベクトル空間のモデルとし，ベクトルを引数 coordinate としてその座標上に捉えて重み付けをしている．

行列はリストのクラス変数 featureVector.fv で定義されているが，featureVector.fv は初期状態では numpy.array 型の空の行列である．featureVector.setVector() が呼ばれた際に行列が空であった場合は，featureVector.initVector() を実行して初期化処理を行う．初期化処理によって，すべての成分の値が 0 である 10*10 の行列を得られる．

なお，オプション vectype に model を指定したときは，入力文章の特徴ベクトルを指す featureVector.fv ではなく，命令モデルの特徴ベクトルを指す featureVector.model が対象の行列となって上記の処理が行われる．ただし featureVector.fv が行列であるのに対し，featureVector.model は行列のリストである．すべての命令モデルの行列が featureVector.model に格納されているのだ．

入力値と命令モデルの誤差を検出する方法は，以下の式によって示される．

$$\min \sum_{i=0}^{l} |x_i - y_i| \tag{1}$$

$\mathbf{x}$ は命令モデルの特徴ベクトル，$\mathbf{y}$ は入力文の特徴ベクトル，l は特徴ベクトルの系列長である．

式 1 を実現するのにはループ処理を使えば良いが，ここでは式 2 のようにベクトルの内積を取ることで簡潔にまとめた．

$$\min \phi^T |\vec{x} - \vec{y}| \phi \tag{2}$$

なお $\phi$ はすべての成分が 1 である 1*10 行列である．numpy.dot() を使うと，第一引数と第二引数の内積を求めることができる．内積を取ることによって，100 次元のデータ df を 1 次元のデータ summation に圧縮し，summation の大小を比較することによって最小誤差のモデルを判断している．なお，df は命令モデルと入力文の特徴ベクトルの差である．

以上の処理の実装をソースコード 12 に示す．

ソースコード 12　RaspiAudio/semanticAnalysis/semanticAnalysis.py

```
import numpy as np
（省略）
def collation(self):
  """collate model and feature vector."""
  try:
    # init model
    if len(featureVector.model) == 0:
      print "init model"
      featureVector.setModel()
    # search error between model and feature vector
    candidate = featureVector.model[0], 9999 # model, df-value
    for model in featureVector.model:
      df = model - featureVector.fv
      ones = np.repeat(np.array([1]), 10) # (1, 1, 1, ... , 1)
      summation = np.dot(np.abs(df), ones).dot(ones)
      # update candidate
      if summation < candidate[1]:
        candidate = model, summation
    print candidate
    return candidate[0]
  except:
    print("Exception in collation.")
```

最後に，得られた命令モデルから命令のキーを取得する．命令とモデルとキーはそれぞれ 1 対 1 の関係にあるため，探索を掛けることで命令のキーを得ることが出来る．

## 4.3 フィルタ処理

音声認識を行う上では，周囲の雑音の対策をする必要があるのは自明のことだ．雑音はパターンマッチの精度を下げる要因になるためである．

Julius では周波数によるパターンマッチが行われている．そのため雑音による影響も，周波数成分上で表現されると言える．故に周波数に関してフィルタ処理を施せば，雑音を抑えることが出来るかもしれない．

このことは荒木によるテキスト [4] でも言及されている．加法性雑音と呼ばれる背景雑音については，周波数空間での減算によって消去可能なのだ．

Julius では単純な減算 (supectrul subtraction) より優秀なケプストラム平均減算による雑音除去が行われているが，河原らのドキュメント [3] によるとマイク入力時における記述しかない．調査不足の可能性はあるが，wav ファイル入力時における雑音除去は為されていないと仮定し，単純な減算フィルタを用意することにした．

フィルタのプログラムは RaspiAudio/voice/lowpassFilter.py である．このプログラムは aidiary によるプログラム [5] を改変したものである．まずは改変したプログラムをソースコード 13 に示す．

ソースコード 13　RaspiAudio/voice/lowpassFilter.py

```
1  #coding:utf-8
2  import struct
3  import wave
4  import numpy as np
5  import scipy.signal
6  from pylab import *
7
8  def save(data, fs, bit, filename):
9      wf = wave.open(filename, "w")
10     wf.setnchannels(1)
11     wf.setsampwidth(bit / 8)
12     wf.setframerate(fs)
13     wf.writeframes(data)
14     wf.close()
15
16 def filtering():
17     wf = wave.open("voice/record.wav", "r")
18     fs = wf.getframerate()
19
20     x = wf.readframes(wf.getnframes())
21     x = frombuffer(x, dtype="int16") / 32768.0
22
23     nyq = fs / 2.0 # nyquist frequency
24
25     # make filter
26     # normalize nyquist frequency
27     fe = 7500.0 / nyq # cutoff frequency
28     numtaps = 255 # number of taps
29
30     b = scipy.signal.firwin(numtaps, fe) # Low-
           pass
31
32     # filtering
33     y = scipy.signal.lfilter(b, 1, x)
34
35     # output filtered data
36     y = [int(v * 32767.0) for v in y]
37     y = struct.pack("h" * len(y), *y)
38     save(y, fs, 16, "whitenoise_filtered.wav")
```

このモジュールを活用するには，lowpass-Fileter.filtering() を外部から呼び出せば良い．するとまずは wave.open() により録音された音声ファイル record.wav が開かれる．第二引数に r を指定したことにより，返り値 wf は Wave_read オブジェクトとなる．

Wave_read オブジェクトの属性 getframerate() はサンプリング周波数を返す．サンプリング周波数 fs を 2 で割った値はナイキスト周波数として nyq に代入される．

Wave_read.getnframes() はオーディオフレームの総数を返す．オーディオフレームとはサンプリングされたデータのことであり，すなわちこのメソッドは wav ファイルがサンプリングされた回数の合計値を返す．この合計値と Wave_read.readframes() を活用することで，wav ファイル全体の標本値の列を得ることが出来る．Wave_read.readframes() は数値

型の引数をひとつ取り，その値が示す個数のオーディオフレーム値をバイナリ文字列で返すメソッドである．こうして得たオーディオフレームのバイナリ x を，np.frombuffer() によって数値の array に変換している．

scipy.signal.firwin() は，窓関数を用いた FIR フィルタを作成する．その第一引数はフィルタ係数であり，第二引数はカットオフ周波数である．カットオフ周波数 fe の実装は，フィルタのカットオフ周波数をナイキスト周波数で正規化することによって得られているので，フィルタのカットオフ周波数（ここでは 7500）を変更することでフィルタの帯域を変更することが出来る．

そして scipy.signal.lfilter() によってフィルタリングが実行される．最後にその実行結果 y を音声バイナリに戻して保存している．

## 5　今後の展望

音声認識オーディオプレイヤの肝となるのは，やはり音声認識である．Julius では予め辞書ファイルに登録された正規文法しか処理できない上，登録された語句が多ければ多いほど誤検知も増えるといった問題を抱えている．この点を近年発達の著しい機械学習を活用することで改善できるのではないか．

当初の計画では N-gram モデルに立脚した選択アルゴリズムや，Web サーバからコーパスを取得するプログラムのような機械学習の仕組みをオーディオプレイヤに搭載する予定であったのだが，時間的・技術的問題により断念せざるを得なくなった．この失敗を活かし，克服することができれば一挙にオーディオプレイヤの使い勝手が向上することだろう．

また，視覚的なインタフェースに対応させることも視野にいれるべきだ．現状のオーディオプレイヤでは完全に音声による入出力に限定されている．しかしより優れた対話型デバイスを目指すというのであれば，それはより人間に近づく進化が必要だろう．単なる GUI への対応だけではなく，MMDAgent[*2]のような 3 次元 CG で作成されたキャラクターを音声認識で動かす技術を実装することが出来るなら，現在よりもずっと人間との対話に近づくに違いない．個人が趣味として行うにはまだまだ難しいとは思うが，いずれこのような人と機械との対話を実現させてみたいと思う．

## 参考文献


[1] sayonari．Julius で音声認識サーバを立てて，wav ファイルを POST 送信して認識する．http://qiita.com/sayonari/items/65a5aea83d1fadac7d5c

[2] 李 晃伸，河原 達也．”Julius を用いた音声認識インタフェースの作成”．ヒューマンインタフェース学会誌，Vol．11，No．1，pp．31–38，2009．

[3] 河原 達也， 李 晃伸．「連続音声認識ソフトウエア Julius」人工知能学会誌， Vol．20， No．1，pp．41–49， 2005．

[4] 荒木 雅弘．イラストで学ぶ音声認識．講談社，2015．

[5] aidiary．SciPy の FIR フィルタの使い方．http://aidiary.hatenablog.com/entry/20111102/1320241544


[*2] http://mmdagent.jp