HARK クックブック

Version 2.0.0. (Revision: 6357)

奥乃 博
中臺 一博
高橋 徹
武田 龍
中村 圭佑
水本 武志
吉田 尚水
大塚 琢馬
柳楽 浩平
糸原 達彦
坂東 宣昭

目 次

第1章　はじめに

本ドキュメントでは，HARK を使う際によくおこる問題とその解決策について記述する．料理本に作りたい料理とその作り方が書いている構成と類似しているので，ここでは問題と解決策の組をレシピと呼ぶことにする．

第 2 章 初めての HARK

初めて HARK を使う人のためのレシピを収録している．音声の録音，音源定位，音源分離，音声認識までを順番に解説していくので，初めて HARK を使う人はこの章のレシピを順番に読んでいこう．

第 3 章 よくある問題と解決方法

よくある問題，たとえばインストールができないとか，録音できないなどの解決方法に関するレシピを収録している．問題があればまずここを見よう．また，デバッグの方法に関するレシピもある．

第 4 章 マイクロホンアレー

マイクロホンアレーの設計に関するレシピを収録している．マイクの数や種類，ロボットやシステムにどう設置するかなどが分からないときは，この章を参照しよう．

第 5 章 入力データの作成

HARK への入力データの作成方法に関するレシピを収録している．具体的には，録音の方法とインパルス応答の測定方法などである．また，シミュレーションで録音データを作成することもできるので，その方法もかかれている．

第 6 章 音響モデルと言語モデル

HARK が音声認識器としてサポートしている Julius は，音響モデルと言語モデルが必要である．この章では，それらの作り方に関するレシピを収録している．

第 7 章 FlowDesigner

HARK によるシステム構築は，ミドルウェア FlowDesigner 上でノードを置き，それらを繋ぐことで行う．ここでは，FlowDesigner の使い方に関するレシピを収録している．バージョン 1.9.9 から新たに開発されたネットワーク作成 GUI **HARK Designer** に関しては別ドキュメント参照。

第 8 章 音源定位

音源定位に関するレシピを収録している．音源定位システムの作り方からパラメータチューニング，デバッグまでカバーしている．

第 9 章 音源分離

音源分離に関するレシピを収録している．ここでも音源定位同様，システムの作り方からチューニング，よくある問題に関するレシピがある．

第 10 章 特徴量抽出

音声認識を行うためには，分離した音から特徴量を抽出する必要がある．ここでは，音声認識でよくつかわれる特徴量やその抽出方法，信頼度に基づいて特徴量を取捨選択するミッシングフィーチャマスクに関するレシピが収録されている．

第 11 章 音声認識

音声認識はオープンソースソフトウェア Julius を使うことを前提としている．ここでは，その設定ファイルの作り方に関するレシピを収録している．

第 12 章 その他

他の章に入らないレシピを収録している．たとえば 短時間周波数解析に使う窓の選び方などである．

第 13 章 進んだ使い方

HARK に新しい機能を追加したり，HARK と別のシステムを接続するなど，HARK の枠を越えたいときにつかえるレシピを収録している．

第 14 章 サンプルネットワーク

ここでは様々なサンプルのネットワークファイルを収録している．まずはこのサンプルを見てネットワークを作成してみるとよいだろう．

第2章　はじめてのHARK

2.1　はじめての録音

Problem

HARK を使って音声を録音してみたいが，やり方が分からない．HARK を初めて使うので，なにをすればいいか分からない．

Solution

HARK をオンラインで動かそうとすると，マイクからの音声入力は避けられない．また，録音は HARK の基本でもあるので，初めて HARK を使う人は，ここから試していこう．ただし，単純に録音をするだけなら録音/再生ツール wios を使えば良い．使い方は HARK ドキュメントか，レシピ: 多チャネル録音したい を参照．

録音デバイス :

まずは，録音デバイスを用意しよう．音源定位や音源分離には多チャネルの録音デバイスを用意する必要がある．ただし，単に録音するだけなら，マイクをコンピュータのマイク端子に差せばよい．録音デバイスの詳細は HARK ドキュメントの デバイスの章を参照．

ここでは，ALSA デバイスを前提とする．ターミナルで，

```
arecord -l
```

を実行してみよう．たとえば，このような出力が得られるはずだ．`card` から始まる行に，接続したデバイス名があることを確認しよう．

```
card 0: Intel [HDA Intel], device 0: AD198x Analog [AD198x Analog]
  Subdevices: 2/2
  Subdevice #0: subdevice #0
  Subdevice #1: subdevice #1
card 1: ** [***], device 0: USB Audio [USB Audio]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
```

ALSA では，デバイス名は，card と subdevice の二つで指定する．例えば，card 0 の subdevice 1 に対応するデバイス名は `plughw:0,1` である．良く分からない場合は，`plughw:0,0` を使っておこう．

録音ネットワーク :

次に，録音ネットワークを作成しよう．図 2.1 と 図 2.2 を参考に作っていこう．パラメータのほとんどはデフォルト値でよいが，表 2.1 に示したパラメータだけは自分で設定しよう．

デバイスのチャネル数，サンプリング周波数は使用するデバイスの仕様を確認して決めよう．デバイス名は，上で決めた名前を使おう．録音フレーム数 (frames) は，次の録音時間 (duration) の計算式から逆算できる．

$$duration[sec] = (LENGTH + (frames - 1) * ADVANCE)/SAMPLING\_RATE \tag{2.1}$$

ただし，上記の変数たちは AudioStreamFromMic のパラメータ．例えば，すべてデフォルト，サンプリング周波数 16000Hz で，5 秒録音したいなら，以下の計算から，498 にすればよい．

$$5[sec] \quad = \quad (512 + (frames - 1) * 160)/16000 \tag{2.2}$$

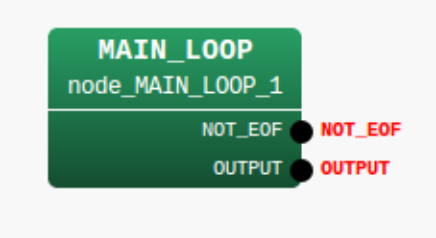


図 2.1: MAIN (subnet サブネットワーク)

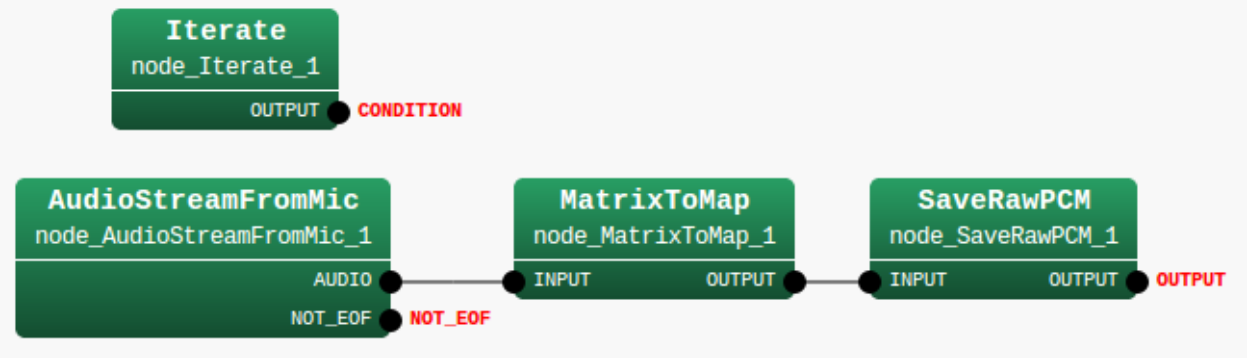


図 2.2: MAIN_LOOP (iterator サブネットワーク)

表 2.1: 録音ネットワークのパラメータ

ノード名	パラメータ名	型	意味と設定値
Iterate	MAX_ITER	`int`	録音フレーム数 (498)
AudioStreamFromMic	CHANNEL_COUNT	`int`	デバイスのチャネル数 (8)
AudioStreamFromMic	SAMPLING_RATE	`int`	デバイスのサンプリング周波数 (1600)
AudioStreamFromMic	DEVICETYPE	`string`	使用するデバイスの種類 (`ALSA`)
AudioStreamFromMic	DEVICE	`string`	使用するデバイス名 (`plughw:0,0`)

録音の実行 ：

完成したネットワークを実行してみよう．録音できれば，チャネルの数だけファイル `sep_0.wav`, `sep_1.wav`... ができているはずだ．適当なプレーヤーで再生しよう．たとえば，次のようにすればよい．

```
aplay sep_0.wav
```

トラブルシューティング ：

これでうまくいけばよいが，失敗する場合は，次のことを確認しよう．また，レシピ うまく録音できないも確認しよう．

マイク接続 ：

プラグが抜けていたり，緩んでいないかを確認し，しっかり接続する．

プラグインパワー ：

録音デバイスがプラグインパワーに対応している時問題ないが，そうでない時は，外部電源を供給する必要がある．電池の残量や電源スイッチを確認しよう．

別の録音ソフトで確認 ：

他のソフトウェアでも録音できない場合は，OS やドライバの設定，録音デバイスの接続または設定を調べよう．

デバイス名が間違い ：

デバイスを複数使用しているときは，名前を間違えている可能性も考えよう．`arecord -l` でもう一度確認しよう．あるいは，いろいろ別のデバイスも試してみよう．

Discussion

SaveWavePCM が最も簡単だが，ヘッダが不要な場合は，SaveRawPCM を使うと 16bit リトルエンディアンで書き込まれた raw ファイルとして保存もできる．その場合のネットワークを図 2.3 に示す．この場合，録音ファイルの拡張子は `.sw` になる．

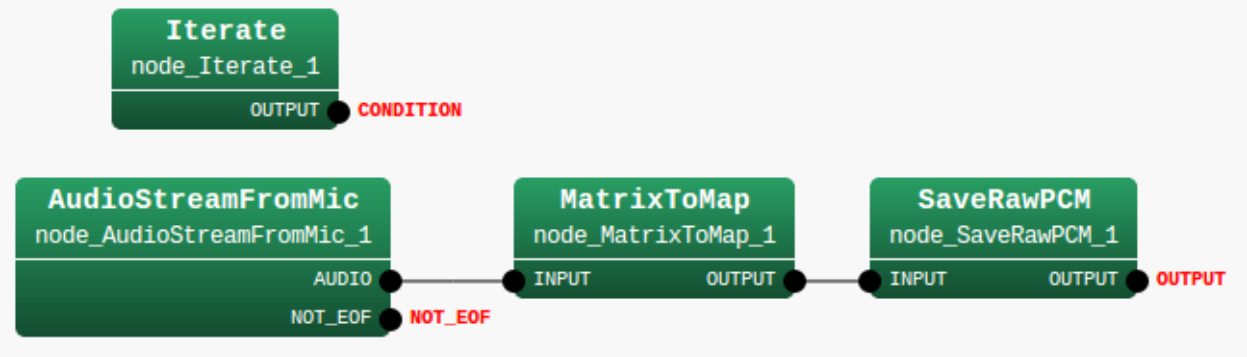


図 2.3: MAIN_LOOP (iterator サブネットワーク)

raw ファイルの読み込みと表示は，python の numpy，pylab モジュールを使って次のように実現できる．

```
import numpy, pylab
waveform = numpy.fromfile("sep_0.sw", numpy.int16)
pylab.plot(waveform)
pylab.show()
```

raw ファイルの再生は, `aplay` に，ファイル形式とサンプリング周波数を与えれば可能だ．

```
aplay -c 1  -r 16000 -f S16_LE sep_0.sw
```

raw ファイルにヘッダを付加すれば，wav ファイルへの変換もできる．sox を使えば簡単だ．例えば，16kHz で録音したファイル `sep_0.sw` を `sep_0.wav` に変換するには，次のようにすればよい．

```
sox -r 16000 -c 1 --bits 16 -s sep_0.sw sep_0.wav
```

See Also

HARK のデバイスの章，レシピ: うまく録音できない などが関連した文章である．録音ツール wios による録音は多チャネル録音したい を参照.

ネットワークファイルのノードやパラメータは，HARK ドキュメントを参照.

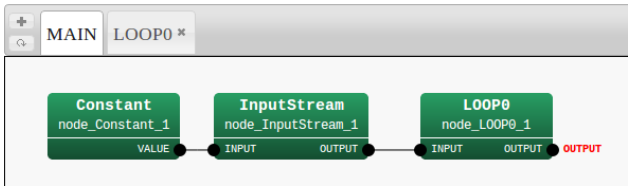
2.2　はじめての音源定位

Problem

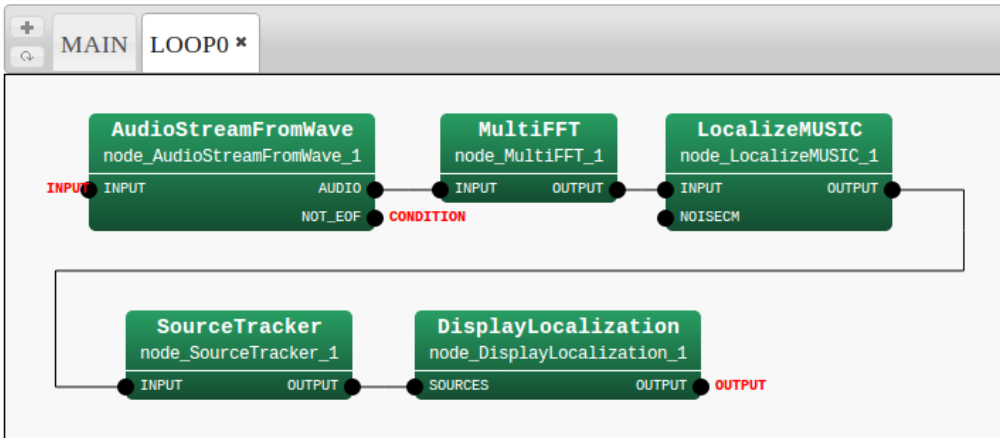
HARK で音源定位をしたいが，何からすればよいかよくわからない．

Solution

2.2.1　音声ファイルの音源定位



(a) メインシート



(b) イタレーターシート

図 2.4: Wave ファイルを入力とした音源定位のネットワークファイル

音源定位をするための最も簡単なネットワークファイルの例として，録音済みの音声データ（マルチチャネルの Wave ファイル）を使って音源定位し，その結果を表示するシステムを図 2.4 に示す．

各モジュールのプロパティの設定に関しては HARK ドキュメントの 6.2 章のモジュールリファレンスを参照されたい．

音源定位が含まれた HARK のネットワークファイルの一例を，HARK 音声認識ファイルセットの中の recog.n で提供している．HARK 音声認識ファイルセットをダウンロードして解凍し，解凍先のディレクトリの中で次のコマンドを実行する．

```
./recog.n MultiSpeech.wav loc_tf.dat sep_tf.dat
```

すると，図 2.5 のような音源定位結果を見ることができる．この音源定位結果を表示するウィンドウが出力されれば，音源定位は正しくできている．

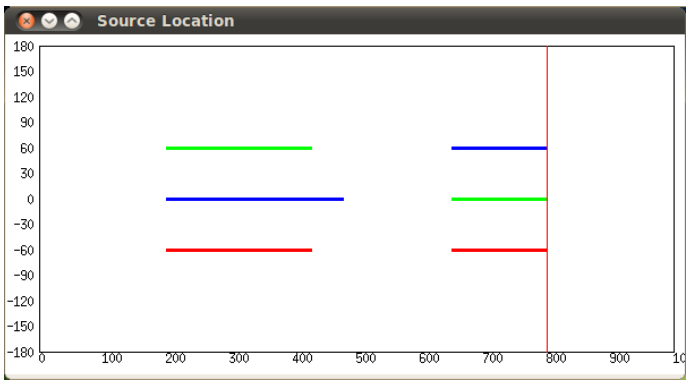
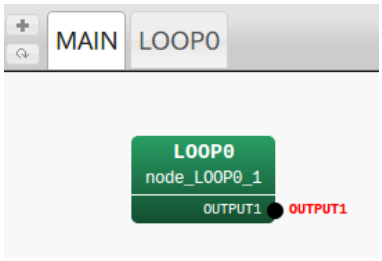


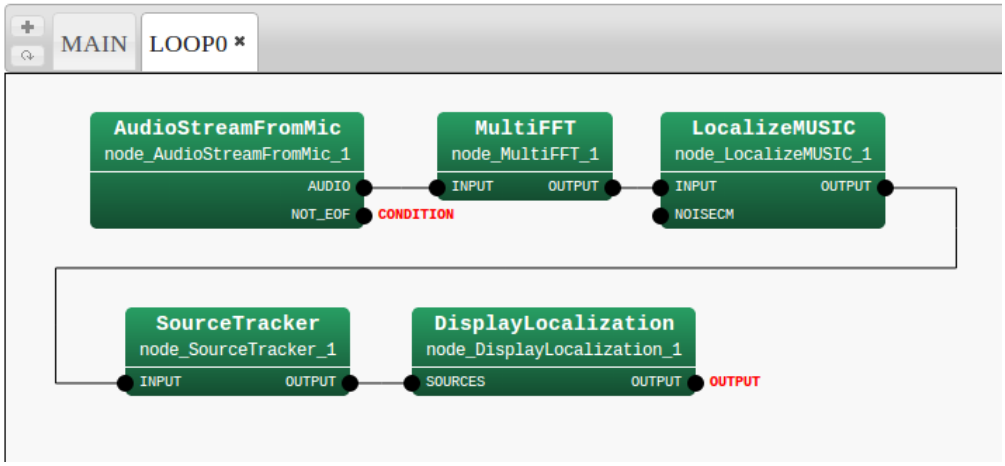
図 2.5: recog.n 実行時の音源定位結果のスナップショット

2.2.2 マイクロホンからのリアルタイム音源定位

次に，マイクロホンアレイから録音したマルチチャネルの音声入力信号を使って音源定位し，その結果を表示するシステムを図 2.6 を示す.



(a) メインシート



(b) イタレータシート

図 2.6: マイクロホンアレイの音声入力による音源定位のネットワークファイル

変更点は AudioStreamFromWave が，AudioStreamFromMic になるのみである．AudioStreamFromMic モジュールのパラメータを HARK ドキュメント 6.1 節のモジュールリファレンスに従って，適切に設定することで，マイクロホンアレイの音声入力信号をリアルタイムに定位することができる．図 2.5 のような音源定位結果が出てくるかを確認されたい.

もし動かないときは，レシピ「うまく録音できない」や「うまく定位できない」を参照されたい．

2.2.3 定常雑音を白色化する機能を用いた音源定位

図 2.4 と図 2.6 で示した音源定位では，目的音や雑音を区別することなく定位するため，目的音よりも大きなパワーを持つ雑音が存在する環境下では，雑音方向の定位結果が出力されやすくなる．結果として，音声認識に必要な音声方向の定位結果が得られなくなり，認識性能が著しく劣化してしまう．

特に，ロボットに搭載されたマイクロホンアレイによる音声認識では，ロボットのファンやサーボモータの雑音が，音声よりもマイクロホンアレイに近いところに存在するため，音源定位性能が劣化する問題がある．

この問題を解決するため，HARK では，既知の定常雑音を白色化する機能を持つ音源定位をサポートしている．本機能を使用するためには以下の手順が必要となる．

3-1) 既知の雑音情報ファイルを生成する．

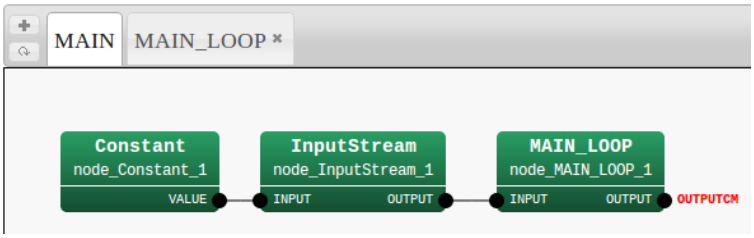
3-2) 雑音情報ファイルを使用して，雑音白色化機能を持つ音源定位を行う．

以下では，この一つ一つについて，解説する．

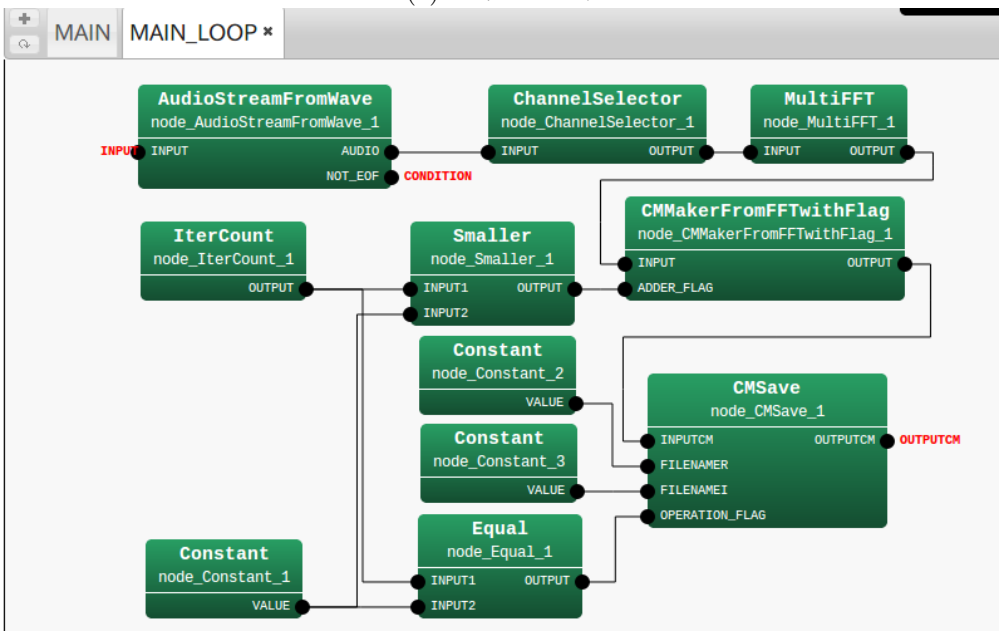
既知の音源定位用雑音情報ファイルの生成

図 2.7 に既知の雑音情報ファイルを生成するネットワークファイルの例を示す．各モジュールのパラメータの設定については，HARK ドキュメント 6.2 節のモジュールリファレンスを参照されたい．また，イタレータシート内の FlowDesigner の標準モジュールである Constant , IterCount , Smaller , Equal については，以下のようにパラメータを設定している．

- `node_Constant_1`
  - **VALUE**
    `int` 型．VALUE = 200.
    先頭フレームから数えて，雑音ファイル作成に使用する雑音のフレーム数を表す．
- `node_Constant_2`
  - **VALUE**
    `string` 型．VALUE = NOISEr.dat.
    雑音ファイルの実部のファイル名を表す．
- `node_Constant_3`
  - **VALUE**
    `string` 型．VALUE = NOISEi.dat.
    雑音ファイルの虚部のファイル名を表す．
- `node_IterCount_1`
  - パラメータ無し
    HARK の処理でのフレーム番号を `int` 型で常に出力する．
- `node_Smaller_1`



(a) メインシート



(b) イタレータシート

図 2.7: 音源定位用雑音情報ファイルを生成するネットワークファイルの例

  - パラメータ無し
    node_IterCount_1 と組み合わせることで，あるフレーム番号との大小関係を比較できる．

- node_Equal_1
  - パラメータ無し
    node_IterCount_1 と組み合わせることで，指定したフレームのみに真を出力できる．

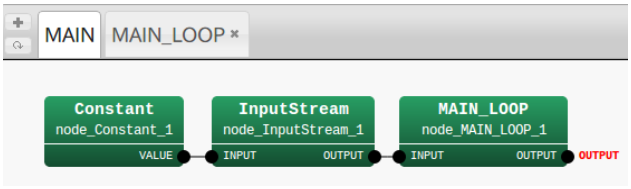
また，ここでは node_Constant_1 の VALUE を 200 と設定しているため，CMMakerFromFFTwithFlag のパラメータである，MAX_SUM_COUNT は 200 より大きな値に設定するのが良い．

このネットワークファイルは，AudioStreamFromWave に雑音情報が入った Wave ファイルを入力することを想定し，先頭フレームから node_Constant_1 で指定したフレーム数までの雑音情報を用いて音源定位用の雑音情報ファイルを生成する．

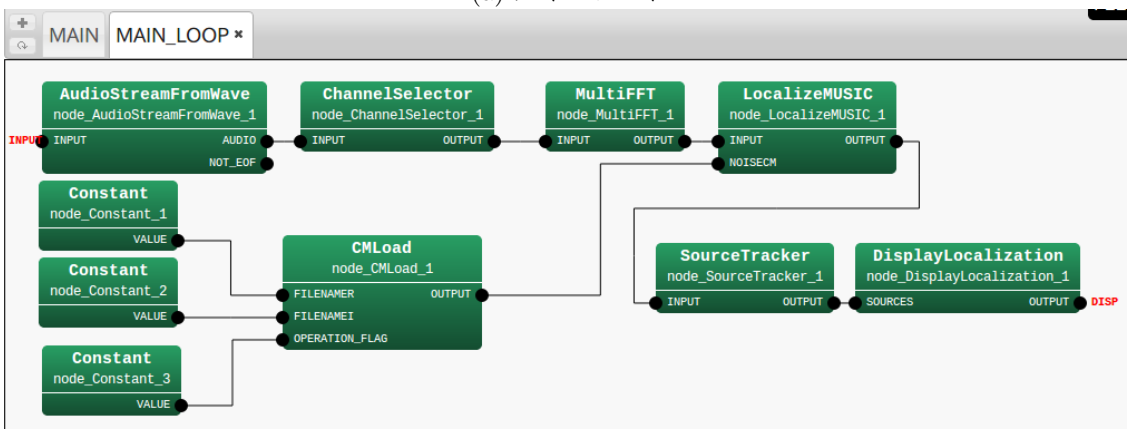
本ネットワークファイルを実行すると，カレントディレクトリに NOISEr.dat と NOISEi.dat が生成されるはずである．この 2 つのファイルを次節の音源定位で使用する．

本稿では，先頭フレームから指定したフレーム数までの雑音情報を使用したが，Smaller などの条件分岐を表すモジュールを適切に接続することで，雑音情報ファイル生成に用いるフレームを変更することも可能である．

雑音白色化機能を持つ音源定位



(a) メインシート



(b) イタレータシート

図 2.8: 雑音を白色化する機能を持つ音源定位のネットワークファイルの例

2.2.3 で生成した雑音情報ファイル (NOISEr.dat, NOISEi.dat) を用いた，雑音白色化機能を持つ音源定位のネットワークファイルの例を図 2.8 に示す．各モジュールのパラメータの設定については，HARK ドキュメント 6.2 節のモジュールリファレンスを参照されたい．また，イタレータシート内の FlowDesigner の標準モジュールである Constant については，以下のようにパラメータを設定している．

- `node_Constant_1`
  - **VALUE**
    `string` 型．VALUE = NOISEr.dat.
    読み込み対象の雑音ファイルの実部のファイル名を表す．
- `node_Constant_2`
  - **VALUE**
    `string` 型．VALUE = NOISEi.dat.
    読み込み対象の雑音ファイルの虚部のファイル名を表す．
- `node_Constant_3`
  - **VALUE**
    `int` 型．VALUE = 0.
    毎フレーム雑音情報ファイル読み込むかを表す．0 の場合は最初のフレームのみ読み込まれる．

このように雑音ファイルを CMLoad によって読み込むことで，音源定位において，そのファイルの雑音情報を白色化することができる．白色化機能を用いるには，LocalizeMUSIC のパラメータ `MUSIC_ALGORITHM` を，

GEVD または GSVD に設定すれば良い．雑音白色化の詳細については HARK ドキュメント 6.2 節のモジュールリファレンスを参照されたい．

このネットワークファイルを実行することで，図 2.5 のような音源定位結果が出てくる．大きなパワーを持つ雑音がある場合に，その雑音が白色化され，音声方向の定位がしやすくなっていることが確認できるだろう．

Discussion

音源定位のアルゴリズムの詳細や雑音情報の白色化については HARK ドキュメント 6.2 節のモジュールリファレンスに詳しく書かれているので参照されたい．より精度を上げるには，8 章のレシピも参考になるだろう．

See Also

うまく録音できない, うまく定位できない

2.3 はじめての音源分離

Problem

HARK を使って特定の方向からくる音響信号だけを分離したい．

Solution

HARK を用いて音源分離を行う場合には，ネットワークファイルと分離を行う GHDSS モジュールで使用する伝達関数 (HGTF バイナリ形式) のファイルもしくはマイク配置 (HARK テキスト形式) が必要である．

音源分離を行うためのネットワークを作成するためには，大きく分けて以下が 4 点が必要となる．

音響信号を入力する ：

音響信号を入力するためには，AudioStreamFromMic もしくは AudioStreamFromWave モジュールを使う．

分離したい方向を指定する ：

分離したい方向を指定するためには，ConstantLocalization ，LoadSourceLocation ，LocalizeMUSIC モジュールを使用する．とりあえず分離を行いたい場合は ConstantLocalization モジュールを使用するのが簡単である．オンラインで定位しながら分離を行うためには LocalizeMUSIC モジュールを使用する．

分離を行う ：

分離を行うのは，GHDSS モジュールである．上記の音響信号と方向の情報を入力とし分離音を出力する．この分離には，実測したインパルス応答から作成した伝達関数もしくはマイクロホンアレイの配置から計算した伝達関数を使用する．

分離音を保存する ：

分離音は周波数領域で出力されるため，Synthesize モジュールを用いて時間領域に戻してから SaveRawPCM もしくは SaveWavePCM モジュールを用いて保存する．

HARK では分離音に対する後処理を行うことができる．この処理は必ずしも必要というわけではないので，使用環境・目的などに応じて使用するかどうかを判断する．

後処理を行う ：

分離音に対して，HRLE モジュールなどを用いてノイズを推定し，除去を行う．これには，PowerCalcForMap ，HRLE ，CalcSpecSubGain ，EstimateLeak ，CalcSpecAddPower ，SpectralGainFilter モジュールを用いる．

図 2.9，2.10 2.11 は音源分離のためのネットワークのサンプルであり，図 2.10 は後処理を行わない例を，図 2.11 は後処理を行う例を示す.「はじめての音源定位」までで作成したネットワークファイルに対し，GHDSS モジュールを接続し，モジュールのパラメータに伝達関数もしくはマイク配置のファイルを指定することにより，音源分離が可能である．ただし，分離音は周波数領域であるため，分離音を聞いてみたい場合などは，Synthesize モジュールを用いて時間領域に変換してから録音する．このネットワークファイルを実行すると，2 話者（正面，左 45 度）を分離し sep_0.wav，sep_1.wav の様にインデックス付きのファイル名で保存していく．

Discussion

以下に目的別のネットワーク作成方法を示す．

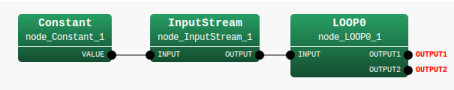


図 2.9: MAIN

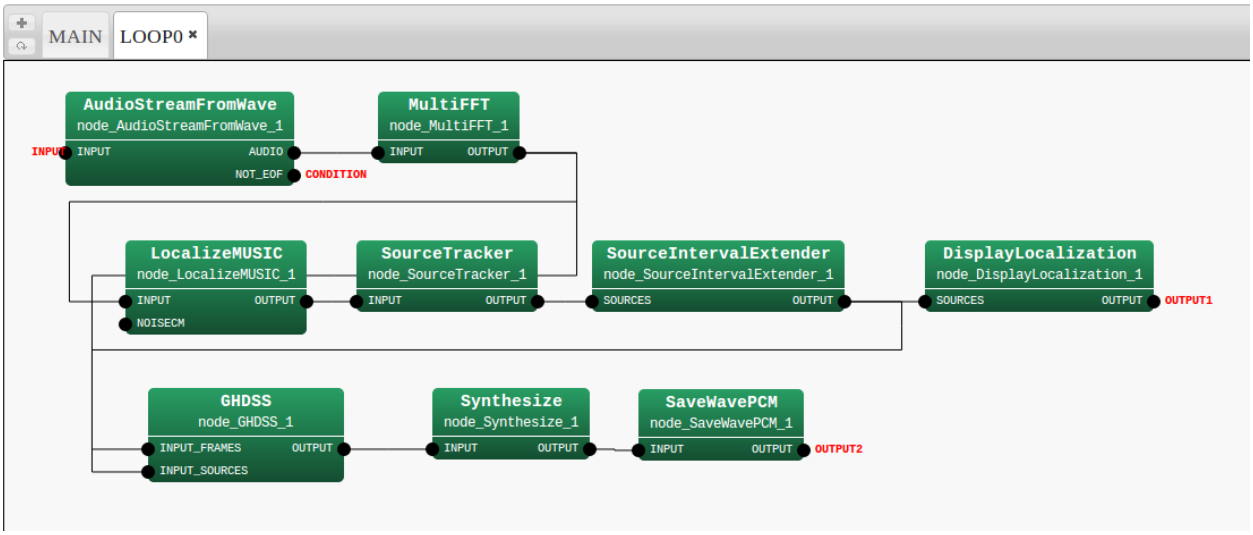


図 2.10: MAIN_LOOP (後処理なし)

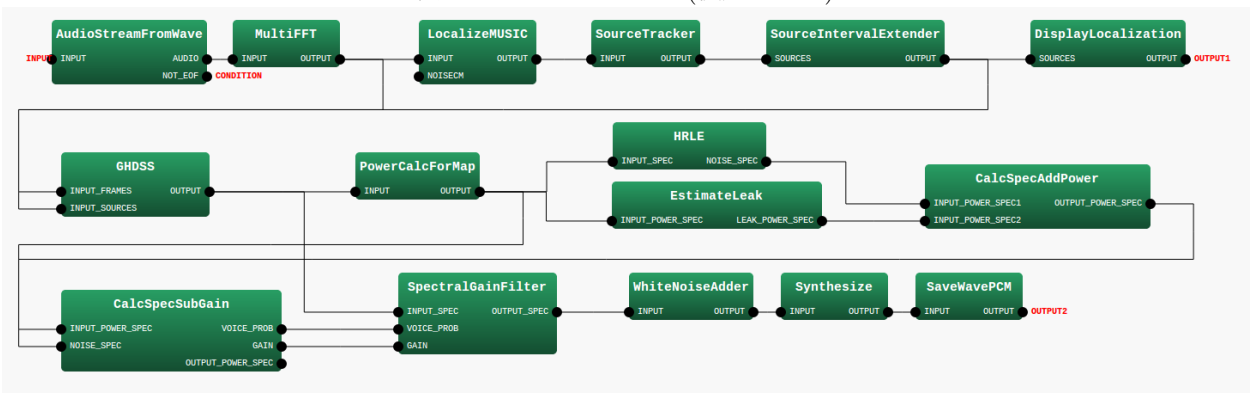


図 2.11: MAIN_LOOP (後処理あり)

オフライン・オンライン :

オンラインで分離処理を行いたい場合には，音声の入力を AudioStreamFromWave モジュールから AudioStreamFromMic モジュールへ変更すれば良い．

音源方向を推定する・指定する :

GHDSS モジュールの INPUT_SOURCES に LocalizeMUSIC モジュールの出力を接続すれば音源方向の推定結果を用いた音源分離を行う．ConstantLocalization モジュールの出力を接続すれば指定した方向にある音源の音のみを分離することができる．なお，LocalizeMUSIC の出力を SaveSourceLocation モジュールを使ってファイルに保存しておけば， LoadSourceLocation モジュールでそのファイルを読み込むこともできる．

実測の伝達関数・マイク位置から計算した伝達関数 :

実測の伝達関数を用いる場合は，TF_CONJ を DATABASE にして，TF_CONJ_FILENAME に伝達関

数のファイル名を指定する．マイク位置から計算した伝達関数を用いる場合は TF_CONJ を CALC にして，MIC_FILENAME にマイク位置を記述したファイル名を指定する．Kinect 以外のデバイスに変更するなどマイクロホンアレイのマイク配置が変更になる場合には，定位用・分離用伝達関数を別途用意する必要がある．

サンプルはあらかじめ各パラメータをチューニングしてあるため，異なる環境では音源分離性能が劣化する可能性がある．チューニングの方法については,「音源分離」を参照されたい．

See Also

伝達関数やマイク配置を記述したファイルについては，HARK ドキュメントのファイルフォーマットの章を参照されたい．うまく分離できない場合には,「うまく分離できていないけどどうすればいいの？」を参照されたい．音源分離は音源定位のあとに行われるため，録音や音源定位など前の段階がきちんと動作しているかを確認することが重要である．録音や音源定位に関しては,「はじめての録音」,「はじめての音源定位」や「うまく録音できない」,「うまく定位できない」が役に立つだろう．

2.4 はじめての音声認識

Problem

HARK を使って音声を認識してみたいが，やり方が分からない．

Solution

HARK を使って音声認識をするためには，大きく分けて二つの処理が必要となる．

1. HARK を使って波形から認識のための特徴量を抽出する
2. Julius_mft を使って特徴量の認識を行う

処理に必要な各種ファイルやパラメータ設定などが複雑であるので，全てを一から準備するより 付録 のサンプルを参考に適宜変更を加えると良い．

特徴量抽出 ：
HARK でサポートする特徴量 MSLS，MFCC を音声から抽出するネットワークの作成方法について説明する．ここでは，一般的に用いられる MSLS，Δ MSLS，Δ パワーもしくは MFCC，Δ MFCC，Δ パワーを抽出するネットワークについて説明する．図 2.12，2.13 は周波数領域の音声から MSLS，MFCC をそれぞれ抽出するネッ

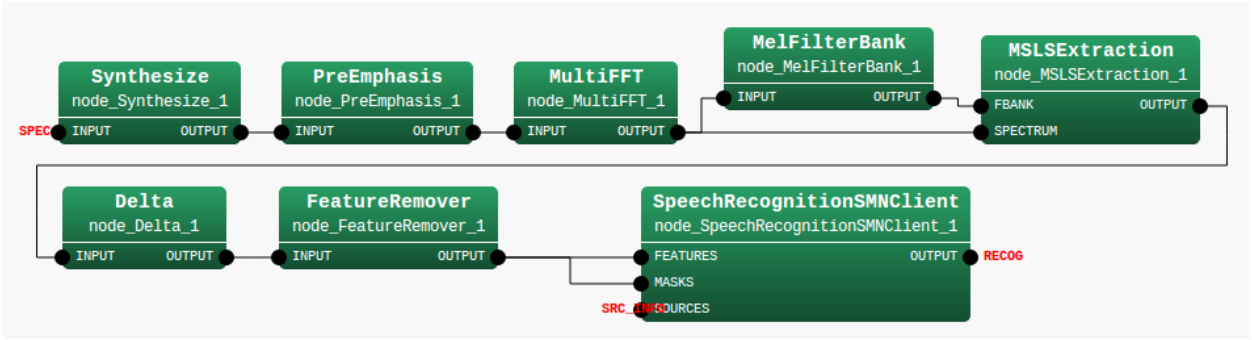


図 2.12: MSLS

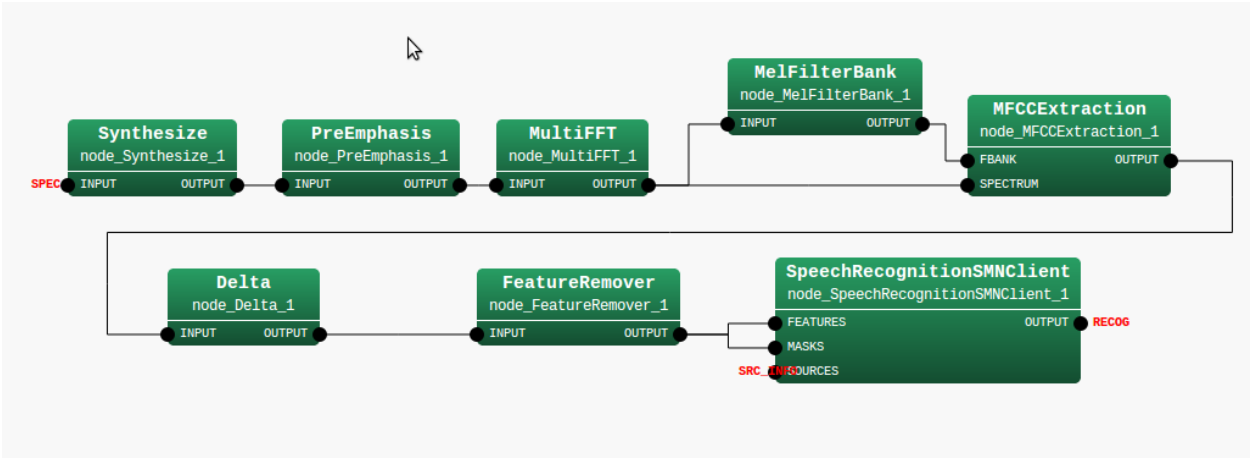


図 2.13: MFCC

トワークの例である．PreEmphasis，MelFilterBank，Delta，FeatureRemover モジュールと MSLSExtraction もしくは MFCCExtraction モジュールを用いて特徴量を抽出している．最後の SpeechRecognitionClient モジュールは，抽出した特徴量を Julius_mft にソケット通信により送信するためのモジュールである．音源方向ごとの認識を行うため，入力には特徴量の他に音源定位結果も必要である．

特徴量抽出が正しくできているかを確認するためなど，ファイルに特徴量を保存したい場合には，SaveFeatures，SaveHTKFeatures モジュールを使用すると良い．

認識 ：

認識は音声認識エンジン Julius を基に拡張された Julius_mft を用いて行う．基本的な使い方は Julius と同様であるため，Julius を使ったことのないユーザーは，Julius のウェブページを参考に基本的な Julius の使い方を習得するとよい．

HARK で抽出した特徴量をソケット通信で受け取り認識を行うためには設定ファイルの入力形式を “mfcnet” にする必要がある．以下は設定の例である．

```
-input mfcnet
-plugindir /usr/lib/julius_plugin
-notypecheck
-h hmmdefs
-hlist triphones
-gram sample
-v sample.dict
```

このうち，はじめの 3 行は HARK から特徴量を送信して認識を行うために必要となる．1 行目は特徴量をソケット通信で受けとるため，2 行目はソケット通信を可能とするプラグインのインストールされているフォルダへのパスを指定するため，3 行目は Julius_mft の型チェックで HARK で抽出した MSLS 特徴量を使用するための設定である．このうち，2 つ目のパスは各自の環境に応じて適切に設定する必要がある．

Discussion

最もシンプルな方法の一つは，AudioStreamFromWave モジュールを使用してモノラルの音声を読み込み，図 2.12 の PreEmphasis モジュールにつなぐ (PreEmphasis モジュールに時間領域の音声波形を渡す) 方法である．分離音の認識を行いたい場合は，GHDSS モジュールの出力 (周波数領域の波形) を図 2.12 もしくは 2.12 の Synthesize モジュールにつなげば良い．

See Also

Julius_mft の設定に関しては，すでに述べた様に Julius とほとんど同じであるため，Julius のウェブページを参考にするとよい．特徴量については，特徴量抽出を，認識に用いる音響モデルや言語モデルの作成については，音響モデルと言語モデルを参考にすると良い．認識の前に定位や分離などを行っている場合は，それぞれの段階で正常に処理できているか一つ一つ確認する必要がある (「うまく録音できない」「うまく定位できない」「うまく分離できない」「うまく認識できない」)

第3章　よくある問題と解決方法

3.1　うまくインストールできない

Problem

HARK のインストールは，HARK インストールガイドに従って HARK リポジトリを登録し、

```
sudo apt-get install harkfd
```

を実行すればインストールできる．
それでうまくいかない．

Solution

HARK がサポートしていない OS にインストールしたいときは，ソースからコンパイルする必要がある．HARK インストールガイドの Installation from Source Compilation を読んで，ソースをダウンロード，コンパイル，インストールすればうまくいくことが多い．
これで失敗する場合は，古いバージョンの HARK が入っていたり，以前にインストールを失敗した HARK が残っている可能性がある．そこで，システムから HARK を 完全に除去し，もう一度インストールを試みよう．

1. HARK 関係のソフトウェアをすべてアンインストールする．ソースからコンパイルしようとしていたなら次を実行して削除する．

   ```
   make uninstall # remove copied files
   make clean     # remove compiled files
   make distclean # remove Makefile
   ```

2. きちんと削除できているかを確かめる．
   コマンドラインで `flowdesigner` を実行し，コマンドが見つからないというエラーが出れば OK である．`ls /usr/local/lib/flowdesigner` を実行し，ディレクトリがなくなっていることを確認．あれば削除する．以前のコンパイル時に別のディレクトリを指定していたら，それも確認し，あれば削除する．

Discussion

インストールで困る大きな問題は以下の 2 点に原因があることが多い:

1. ソフトウェア同士の依存関係が解決できていない
2. インストールするパスが異なる

本レシピは，それを 1 から確認する方法を示している．

ただし，何よりも先に，エラーメッセージをよく読むのが重要である．エラーメッセージが解読できれば問題は半分解決したと言っても過言ではない．

See Also

インストールができたらはじめての HARK を読んでいこう．

3.2 うまく録音できない

Problem

システムがうまく動作しない理由が，そもそも録音できていないということは多い．ここでは，(1) 録音ができているかの確認方法と，(2) 典型的な録音ができない理由と対処法を述べる．

Solution

最初に レシピ はじめての録音 を読みながら，録音してみる．録音中に何か声を出しておけば，あとで確認がしやすい．次に，録音したファイルを波形を表示できるソフトウェアで確認する．ソフトウェアには例えば，Audacity や wavsurfer などを使えば良い．3 個以上のマイクロホンを使って録音しているときは，これらの多チャネルに対応するソフトウェアを使うのが便利である．表示した結果，マイクロホンを接続しているすべてのチャネルで自分の波形が録音されていれば成功である．少なくとも接続は正しい．

もし振幅がほとんどなかったり，クリッピングを起こしていたりするチャネルがあったら，次の事を確認しよう．

それぞれのマイクロホンは故障していないか？ 身近な PC 等につないで，一つずつ録音できているかを確認しよう．

マイクロホンのコネクタはきちんと接続されているか？ コネクタが固定されているか確認しよう．抜き差ししてみよう．

プラグインパワーは使っているか？ ある種のマイクロホンは，外部電源 (プラグインパワー) が無いと十分な音量を確保出来ない．録音デバイスが供給してくれる場合もあるが，もしマイクロホンがプラグインパワーが必要で，かつデバイスの供給が無い場合は電源が必要である．マイクロホンとデバイスの両方のマニュアルを確認しよう．

Discussion

家電が動かない原因がコンセントが差さっていないことが多いのと同様に，音声認識システムが動作しない原因がそもそも録音ができていなかったという事は多い．特に多数のマイクロホンを使うシステムでは，ソフトウェアを動かす前に着実にそれぞれのマイクロホンで録音できるかを確認することは重要である．

See Also

詳しい録音の方法は はじめての録音を参照．HARK でサポートしている録音デバイスの説明は HARK ドキュメントのデバイスが詳しい．

3.3 うまく定位できない

Problem

音源定位システムを作ったけれどもうまく定位しない．

Solution

ここでは，定位システムが動作しない場合の典型的な確認方法について述べる．

録音の確認

まずは，うまく録音できないを見て，録音ができているか確認しよう．

オフラインで音源定位

録音ができていれば，次は波形ファイルを使って定位を試してみよう．波形ファイルは，作成したマイクロホンアレイで録音すればよい．例えば，10 秒間録音して，その間一人で声を出しながらゆっくりマイクロホンアレイのまわりを歩いてみるとよい．

ネットワーク中の AudioStreamFromMic を AudioStreamFromWave に入れ替えれば，音声波形から定位ができる．定位結果は，SourceTracker の出力に DisplayLocalization を接続すれば視覚的に確認できる．

この時点でうまく動いていなければ，まずは LocalizeMUSIC の DEBUG を ON にして，MUSIC スペクトルを表示してみよう．音声がある時間・方向で値が大きくなっているはずである．もしなっていなければ，そもそも伝達関数の計算が怪しい．うまくいっていれば， SourceTracker のパラメータ THRESH を，音声部分よりは小さく，無音部分よりは大きく設定しよう．これで，音源が定位できる．

その他にも，次のような場合は LocalizeMUSIC のパラメータを調節してみると良い．

1. NUM_SOURCE を話者数に合わせる．
   例えば話者が 2 名までしかいないと仮定するなら，NUM_SOURCE は 2 にすると良い．
2. MIN_DEG と MAX_DEG を定める．
   音源が無いのに定位結果が出る状況では，壁からの反射やノイズ源 (PC のファンなど) が原因である可能性がある．そのとき，その方向に目的音源が無いと仮定できるなら，この二つのパラメータを使ってその方向からは定位結果が出力されないように設定できる．例えばロボットの後ろから話しかけることはないなら，MIN_DEG を -90，MAX_DEG を 90 とすれば，正面側のみを定位対象にできる．

オンラインで音源定位

ここまでできれば，あとはネットワークファイルを AudioStreamFromMic に戻して，実際のシステムに使ってみよう．MUSIC スペクトルの値は，話者との距離や音量によって異なるので，SourceTracker のパラメータ THRESH の微調整が必要かもしれない．0.1 - 0.2 ぐらいずつ上げ下げしてみよう．誤りの定位結果が多いなら THRESH をやや増やし，声を出しても定位結果が出ないなら THRESH をやや減らそう．

周囲の雑音の計測

LocalizeMUSIC の伝達関数を測定した環境と音源定位を行うときの環境の周囲の雑音レベルが異なれば，性能は劣化してしまう。例えば、音源定位時に近くにエアコンがある場合にそのエアコンの音源定位結果をずっと表示してしまう。そのような場合は，2 種類の対策がある。

1. 方向のフィルタリング:
   SourceSelectorByDirection を SourceTracker の後段に接続すればその方向の音源定位結果を捨てることができる。あるいは、LocalizeMUSIC の MIN_DEG, MAX_DEG を変更することでも特定方向の音源定位結果を出さないことは可能だ。目的音源の無い方向がわかっている場合は、これが有効だろう。
2. 雑音相関関数の作成:
   事前に雑音を録音しておき、それを LocalizeMUSIC に与えることで、その雑音による影響を排除した定位が可能となる。特に目的音より大きなパワーの雑音がある場合などに有効だ。詳しい方法は定常雑音を白色化する機能を用いた音源定位を参照。

Discussion

部屋の残響や話者の声の大きさなどで，適切なパラメータは変わる．現在の HARK で定位性能をあげるには，その場でチューニングを行うのが最も確実である．最も重要なのは THRESH なので，基本的にはそれのみを調節すればよい．

See Also

定位システムを初めて作るときは はじめての音源定位 を参照するとよい．HARK ドキュメントの の LocalizeMUSIC と SourceTracker を参照すると，手法の詳細が分かる．他にも，音源定位の章には音源定位に関するノウハウがある．

3.4 うまく分離できない

Problem

マイクロホン入力からの混合音を分離できない．出力を聞いたところ，音声とは思えない音が出力されていたり，歪みが酷いと感じる．

Solution

GHDSS ノードを用いて分離を行なっている場合は，次のことを確認する．

音源定位ができているかどうか GHDSS は入力に音源定位結果を用いているため，性能は音限定位に依存する．したがって，まずは音源定位結果を表示するようにネットワークファイルを修正し，音源定位の確認を行う．詳しくは，うまく定位できないを参照．

GHDSS が正しく音源分離しているかどうか GHDSS が音源分離できているかを確認するため，まず，分離音を SaveWavePCM などで保存し，分離音を確認する．GHDSS で分離が失敗している場合は，音源分離のパラメータをチューニングしたいを参照．

分離の後処理が上手く動作しているかどうか 音源分離の後にに PostFilter や HRLE といった後処理ノードを用いている場合は，そのパラメータ設定が原因かもしれない．後処理後の分離音を保存して確認する．この場合は，分離音に入っている雑音を後処理で減らしたいを参照．

Discussion

なし．

See Also

うまく定位できない, 音源分離のパラメータをチューニングしたい, 分離音に入っている雑音を後処理で減らしたい

3.5 うまく認識できない

Problem

HARK を使った音声認識システムを作ろうとしているが，何を話しても認識してくれない．

Solution

もし，あなたがまだチュートリアルの章 (2 章) を一通り試していないなら，ここから始めるのが近道になるだろう．このチュートリアルで，HARK による音声認識システムを録音，音源定位，音源分離，音声認識の順に学ぶことができる．

次に，独自のシステムを開発して動かない場合の検証方法について解説する．音声認識システムは多数の要素が含まれているので，原因を一つずつ検証していくのが重要である．まず，HARK が正しくインストールされているか (うまくインストールできない)，録音ができているか (うまく録音できない)，音源定位ができているか (うまく定位できない)，音源分離できているか (うまく分離できない) を，それぞれのレシピを見ながらで確認しよう．

ここまで検証して正しく動いているなら，音声認識の検証に入ろう．ここでは， 大語彙連続音声認識エンジン Julius (http://julius.sourceforge.jp/) を HARK 用に改造したもの (julius_mft) を用いると仮定する．Julius を使う上で重要なのは次の 3 つのファイルである

1. 音響モデル: 音響信号の特徴量と音素の関係を表すモデル
2. 言語モデル: システムの使用者が話す言葉のモデル
3. 設定ファイル: 上記 2 種類のファイルのファイル名など

まったく動作しない，あるいは Julius が起動しない場合は，ファイルのパスが間違っているということが多いので，まず設定ファイルを見直すのが容易である．詳しくはレシピ 設定ファイル (.jconf ファイル) を作りたい を参照するとよい．あるいは，モデル自体の検証を行うときは音響モデルは 音響モデルを作りたい を，言語モデルは 言語モデルを作りたい を参照しよう．

Discussion

以上はまったく認識しない場合の対処方法で,「正しくは動くが，成功率 (=認識率) が悪い」という場合はチューニングを行う必要がある．

音声認識システムのチューニングは，定位・分離・認識の多岐に渡る複雑な問題である．例えば，分離なら音源分離ノード GHDSS のパラメータチューニング (音源分離のパラメータをチューニングしたい) や後処理 (PostFilter や HRLE) のチューニング (分離音入っている雑音を後処理で減らしたい) を参照するとよい．他にも，認識に使用する特徴量にかんするレシピ 特徴量抽出 の章や，パフォーマンスの向上に関するレシピ (音源定位のパラメータをチューニングしたい，音源分離のパラメータをチューニングしたい) が参考になるだろう．

See Also

ここでは，そもそも認識しないときに参照すべきレシピのみを列挙する．認識率が悪い場合は Discussion を参照．

1. よくある問題と解決方法の章の各レシピ
2. 音響モデルのレシピ，言語モデルのレシピ

3. Julius の設定ファイル

4. Julius のドキュメント

3.6 デバッグモジュールを作りたい

Problem

自分で作ったシステムのデバッグができるようなモジュールを作成するにはどうしたらいいか.

Solution

基本的には

- print 文で中の情報を表示して確かめる.

という普通の方法で自分のモジュール内のデバッグができる．自分で作ったモジュールのソース中に

- `cout << message1 << endl;`
- `cerr << message2 << endl;`

などと記述しておくことで，コンソール上でのメッセージを確認する．さらに，自分で作ったネットワークファイル (`nfile.n` とする) を，flowdesigner の GUI からではなく，コマンドライン上で

- `./nfile.n > log.txt`
  と実行すると，`cout` で吐き出されたメッセージ（上記の `message1` ）が `log.txt` に保存される
- `./nfile.n 2> log.txt`
  と実行すると，`cerr` で吐き出されたメッセージ（上記の `message2` ）が `log.txt` に保存される

LocalizeMUSIC などのノードでは `DEBUG` というパラメータがあり，これを `true` にした場合のみメッセージを出力する．このようなスイッチを付けておくことで，確認したいメッセージのみを抽出できて便利である.

3.7 デバッグツールを使いたい

Problem

自分でモジュールを作ってデバッグする以外に何か方法はあるか.

Solution

以下の 3 項目が主に考えられる.

1. コマンドライン上から gdb を利用したデバッグ
2. harktool による伝達関数の誤差確認

harktool についてはプログラムのデバッグというよりは，定位や分離のパフォーマンスを上げる方に近いかもしれない．以下，項目毎に簡単に説明する.

1. **コマンドライン上から gdb を利用したデバッグ**
   gdb とは GNU のフリーのソースレベルデバッガであり，ブレイクポイントなどを指定して途中で実行を止めたり，一行ごとに実行したりと，色々と便利なツールである．HARK のデバッグに関しては以下のように使用する.

   - `-g -ggdb` オプションつきでコンパイル
     （`Makefile.am` に `AM_CXXFLAGS = -g -ggdb` を追加）
   - `gdb /path/flowdesigner`
     これで gdb コンソールに移行する.
     `(gdb)`
   - ブレイクポインタの設定
     `(gdb) b MODULE::calculate`（関数名で指定する場合）
     `(gdb) b module.cc:62`（行数で指定する場合．`module.c` の 62 行目）
   - ブレイクポインタの確認
     `(gdb) i b`
   - 条件付きブレイクポイント（一例）
     `(gdb) cond 3 i==500`（ループ処理の中で 3 番目のブレイクポイントを 500 回実行した後に中断する）
   - 普通の実行
     `(gdb) r nfile.n`（`nfile.n` を実行する）
   - ステップ実行（一行ずつ実行）
     `(gdb) n`
   - 実行再開（ブレイクポイントなどで中断している時）
     `c nfile.n`

2. **harktool による伝達関数の誤差確認**
   HARK ドキュメントの harktool の節を参照.

See Also

gdb によるブレイクポイントを利用したデバッグモジュールの作り方については，HARK 講習会資料の「実装 ： Channel Selector の改造」を参照.

3.8　マイクロホンの接続をチェックしたい

Problem

作成したシステムがうまく動作しない．どうもマイクロホンの接続が怪しいと思う．

Solution

マイクロホンの接続を確認するには，マイクロホン側と録音機側の両方を調べる必要がある．マイクロホンを別の PC に接続して録音できれば，録音機に原因がある．録音機に別のマイクロホンを接続して録音できれば，マイクロホンに原因がある．そこから地道にチェックしていこう．詳しくは，うまく録音できない を見て探していこう．

Discussion

うまくいかないときは，(1) どこまではうまくいって，(2) どこからうまくいかなくなるかの境界を特定するのが重要である．根気よく探していこう．

See Also

うまく録音できない

第4章　マイクロホンアレー

4.1　マイクロホン数はいくつがいい？

Problem

自分のロボットに，マイクロホンを搭載する時に読む．

Solution

原理的には，音源数＋1個のマイクロホンを用意すれば， 任意の個数の音源を分離可能．実際には個々のマイクロホンの配置・使用環境に依存する．音源数が既知であることも稀であり，ベストなマイクロホン数は，試行錯誤して決定される．

音源定位・音源分離・分離音認識それぞれのパフォーマンスが自分の要求する水準に達する数で，最小の数にするのがよい．

全方位の音源を扱う場合には，同心円の円弧上に等間隔に配置するのが理論的にはもっともよい配置である．ただし，ロボットの頭は完全な球体であるとする．もし完全な球体でない場合，頭部形状が不連続な反射が起こる方向において，音源定位，音源分離のパフォーマンスが低下しやすい．同心円は，連続的な反射が起る頭部位にとるのがよい．

Discussion

【参考】現在我々の 3 話者同時発話認識デモは，8 つのマイクロホンで 3 音源分離している．理論的に分離可能な音源数 7 の半分以下ということになる．多数の音源を分離するために， マイクロホン数を増やし，パフォーマンスの改善を狙うことも可能である (例: 16 マイクロホン)．しかし，マイクロホン間隔が密になりすぎると，パフォーマンスの改善が得られ難くなり，計算コストだけが増大する．

4.2 マイクロホン配置はどうしたらよい？

Problem

自分のロボットに，マイクロホンを搭載する時に読む.

Solution

配置の前提条件は個々のマイクロホン間の相対的な位置が変化しないことである．定位・分離精度に配慮した配置が必要で，適切な配置は，処理対象となる音源の方向に依存する．特定方向の音源を仮定できる場合には，そちらに密に配置する．特定方向の法線ベクトルに垂直に広く分散配置するのが良い．全方位の音源を扱う場合には，円状に配置する．音源分離には，マイクロホン間隔が広い方が有利である．可能な限り広い間隔で配置する．

少なくとも音源定位できない配置では，音源分離が困難であるため，音源定位ができるかどうかをチェックし，配置を決める．

定位精度が出ない場合は，マイクロホン位置の周囲に音の反射が不連続になる形状になっていないかチェックし，そのような場所を避けて配置すると良い.

Discussion

マイクロホン間の相対的な位置が変化するとチャネル間の相関が変化する．HARK では，マイクロホン間の相対位置が固定であることが前提である.

4.3 どんなマイクロホンを使えば良い？

Problem

HARK を使うためにマイクロホンを使用したいが、何を使えばいいかわからない。

Solution

我々は数千円の無指向性エレクトレットコンデンサマイクロホンを用いている．必ずしも高価なマイクロホンを使用する必要はない．

信号対ノイズ比の大きいマイクロホンがよい．配線の被服がしっかりした物を選ぶと良い．

Discussion

信号対ノイズ比をできるだけ大きく録音するためには，配線の被服がしっかりした物，プラグのしっかりした物を選ぶとよい．こういったマイクロホンは高価であることが多いが，数千円程度のマイクロホンで動作実績がある．ロボット内部に配線する場合には，高価なマイクロホンを使用するよりも配線の被服や，他の信号線による干渉を抑制することに気を付ける方がよい．

4.4 自分のロボットにマイクロホンを搭載したい

Problem

自分のロボットにマイクロホンを搭載するときの配線のポイントが知りたい。

Solution

マイクロホンの配線は，配線長を短くし，ロボット内部での配線では，サーボ用信号線や，電源用の線などの他の配線と並行に配線しない．信号の伝送には，差動方式やディジタル方式を用いると良い．

マイクロホンは，ロボット筐体の表面から浮かせないように設置する．つまり，マイクロホンは筐体に埋め込まれた状態で，マイクロホン先端だけが外部に開放されている状態に設置する．

Discussion

マイクロホンの配線は，配線長を短くし，ロボット内部での配線では，サーボ用信号線や，電源用の線などの他の配線と並行に配線しない．他の信号線からノイズを拾わないようにするためである．これらの配慮は，シングルエンドで特に重要である．

マイクロホンは，ロボット筐体の表面から浮かせないように設置する．つまり，マイクロホンは筐体に埋め込まれた状態で，マイクロホン先端だけが外部に開放されている状態に設置する．表面から浮いていると筐体からの反射の影響を受け，音源定位，音源分離性能が低下する．ロボットの動作によって筐体が振動する場合には，マイクロホンが振動を拾わないように配慮する．ブッシュを入れるなど，振動を抑制する素材を台にする．

4.5　サンプリングレートはどう設定したらいいい？

Problem

デバイスから音響信号を入力する際に，どのようにサンプリングレートを決めたらいいか分からないときに読む．

Solution

特別な理由のない場合は，サンプリングレートは 16kHz で良い．そうでない場合は，エイリアスが生じない範囲で最も低いサンプリングレートを用いる．

サンプリングレートを設定する際には，エイリアスが発生しないように考慮する必要がある．サンプリングレートを半分にした周波数はナイキスト周波数と呼ばれ，このナイキスト周波数を越える周波数成分をもつ信号はサンプリングを行う際にエイリアスが生じる．そのため，入力信号にエイリアスが発生するのを防ぐためには，可能な限り高いサンプリングレートを用いることが好ましく，最も高いサンプリングレートを選んだとき最もエイリアスが起きにくい．

一方でサンプリングレートを高くすると，処理するデータの量も増加し計算コストが増大する．そのため，サンプリングレートを設定する際には必要以上にサンプリングレートを高くしない方が良い．

Discussion

音声のエネルギーは帯域幅にして 10kHz 以上に達するが，エネルギーは低周波成分に多く存在する．そのため，たいていの目的に対しては低い周波数帯域だけを考えれば充分な場合が多い．例えば，電話の場合はおよそ 500Hz から 3500Hz 程度であり，5kHz 程度までの帯域幅があれば，容易に理解できる音声信号を伝送することが可能である [1]．16kHz サンプリングの場合は 8kHz 以下の周波数成分はエイリアスを生じることなくサンプリング可能であり，音声認識などでは用いられることが多い．

[1] 音声の音響分析，レイ・D・ケント，チャールズ・リード著，荒井　隆行，菅原　勉監訳，海文堂，2004.

4.6 別の A/D 変換器を使いたい。

Problem

このレシピは HARK が標準でサポートしている ALSA(Advanced Linux Sound Architecture) 準拠のデバイス，System In Frontier, Inc. RASP シリーズ，東京エレクトロンデバイス製 TD-BD-16ADUSB 以外のデバイスを使って音響信号をキャプチャしたいときに読む.

Solution

上記の 3 種以外のデバイスを使ってキャプチャするためには，デバイスに対応したモジュールを自作する必要がある．もちろん，AudioStreamFromMic と全く異なる方法でも実装は可能である．しかし，ここでは AudioStreamFromMic を拡張する方法を述べる．

作成手順は大きく以下のようになる．

1. デバイスに対応したクラス “NewDeviceRecorder” を作成し，そのソースファイルとヘッダーファイルを HARK ディレクトリ内の librecorder ディレクトリに置く．
2. 作成したクラスを使えるよう AudioStreamFromMic を書き変える．
3. コンパイルするため，Makefile.am, configure.in を書き変える．

(a) では，デバイスからデータを取り込みバッファに送るクラスを作成する．このクラスは Recorder クラスを継承して実装する．デバイスを使用するための準備等を行う Initialize とデバイスからデータを取り出す operator() メソッドを実装する．

(b) では，オプションに “NewDevice” が指定された時の処理を記述する．具体的には，コンストラクタ内に NewDevice の宣言，初期化，およびシグナルの設定を行う．

(c) では，新しく追加したファイルに対応し，Makefile.am, configure.in などを変更する．

以下に示すのは新しいデバイス（NewDevice）をサポートする AudioStreamFromMic 2 のサンプルである．

```
#include "BufferedNode.h"
#include "Buffer.h"
#include "Vector.h"
#include <climits>
#include <csignal>


#include <NewDeviceRecorder.hpp> // Point1: 必要なヘッダファイルを読み込む

using namespace std;
using namespace FD;

class AudioStreamFromMic2;

DECLARE_NODE( AudioStreamFromMic2);
/*Node
 *
 * @name AudioStreamFromMic2
 * @category MyHARK
 * @description This node captures an audio stream using microphones and outputs frames.
 *
 * @output_name AUDIO
 * @output_type Matrix<float>
 * @output_description Windowed wave form. A row index is a channel, and a column index is time.
```

```
 *
 * @output_name NOT_EOF
 * @output_type bool
 * @output_description True if we haven't reach the end of file yet.
 *
 * @parameter_name LENGTH
 * @parameter_type int
 * @parameter_value 512
 * @parameter_description The length of a frame in one channel (in samples).
 *
 * @parameter_name ADVANCE
 * @parameter_type int
 * @parameter_value 160
 * @parameter_description The shift length beween adjacent frames (in samples).
 *
 * @parameter_name CHANNEL_COUNT
 * @parameter_type int
 * @parameter_value 16
 * @parameter_description The number of channels.
 *
 * @parameter_name SAMPLING_RATE
 * @parameter_type int
 * @parameter_value 16000
 * @parameter_description Sampling rate (Hz).
 *
 * @parameter_name DEVICETYPE // Point2-1: 使うデバイスのタイプを加える
 * @parameter_type string
 * @parameter_value NewDevice
 * @parameter_description Device type.
 *
 * @parameter_name DEVICE // Point2-2: 使うデバイスの名前を加える
 * @parameter_type string
 * @parameter_value /dev/newdevice
 * @parameter_description The name of device.
 END*/

// Point3: 録音を途中で停止させるための処理を記述する
void sigint_handler_newdevice(int s)
{
  Recorder* recorder = NewDeviceRecorder::GetInstance();
  recorder->Stop();
  exit(0);
}

class AudioStreamFromMic2: public BufferedNode {
  int audioID;
  int eofID;
  int length;
  int advance;
  int channel_count;
  int sampling_rate;
  string device_type;
  string device;
  Recorder* recorder;
  vector<short> buffer;

public:
  AudioStreamFromMic2(string nodeName, ParameterSet params) :
    BufferedNode(nodeName, params), recorder(0) {
    audioID = addOutput("AUDIO");
    eofID = addOutput("NOT_EOF");

    length = dereference_cast<int> (parameters.get("LENGTH"));
    advance = dereference_cast<int> (parameters.get("ADVANCE"));
    channel_count = dereference_cast<int> (parameters.get("CHANNEL_COUNT"));
    sampling_rate = dereference_cast<int> (parameters.get("SAMPLING_RATE"));
    device_type = object_cast<String> (parameters.get("DEVICETYPE"));
    device = object_cast<String> (parameters.get("DEVICE"));
```

```
// Point4: デバイスタイプに対応したレコーダクラスを作成する
    if (device_type == "NewDevice") {
      recorder = NewDeviceRecorder::GetInstance();
      recorder->Initialize(device.c_str(), channel_count, sampling_rate, length * 1024);
    } else {
      throw new NodeException(NULL, string("Device type " + device_type + " is not supported."),
                              __FILE__, __LINE__);
    }
    inOrder = true;
  }

  virtual void initialize() {
    outputs[audioID].lookAhead
      = outputs[eofID].lookAhead
      = 1 + max(outputs[audioID].lookAhead, outputs[eofID].lookAhead);
    this->BufferedNode::initialize();
  }

  virtual void stop() {
    recorder->Stop();
  }

  void calculate(int output_id, int count, Buffer &out) {
    Buffer &audioBuffer = *(outputs[audioID].buffer);
    Buffer &eofBuffer = *(outputs[eofID].buffer);
    eofBuffer[count] = TrueObject;
    RCPtr<Matrix<float> > outputp(new Matrix<float> (channel_count, length));
    audioBuffer[count] = outputp;
    Matrix<float>& output = *outputp;

    if (count == 0) { //Done only the first time
      recorder->Start();
      buffer.resize(length * channel_count);

      Recorder::BUFFER_STATE state;
      do {
        usleep(5000);
        state = recorder->ReadBuffer(0, length, buffer.begin());
      } while (state != Recorder::OK); // original

      convertVectorToMatrix(buffer, output, 0);

    } else { // Normal case (not at start of file)

      if (advance < length) {

        Matrix<float>& previous = object_cast<Matrix<float> > (

          for (int c = 0; c < length - advance; c++) {
            for (int r = 0; r < output.nrows(); r++) {
              output(r, c) = previous(r, c + advance);
            }
          }
      } else {

        for (int c = 0; c < length - advance; c++) {
          for (int r = 0; r < output.nrows(); r++) {
            output(r, c) = 0;
          }
        }
      }

      buffer.resize(advance * channel_count);
      Recorder::BUFFER_STATE state;

      for (;;) {

        state = recorder->ReadBuffer((count - 1) * advance + length,
     advance, buffer.begin());
```

```
        if (state == Recorder::OK) {
          break;
        } else {
          usleep(5000);
        }
      }

      int first_output = length - advance;
      convertVectorToMatrix(buffer, output, first_output);
    }

    bool is_clipping = false;
    for (int i = 0; i < buffer.size(); i++) {
      if (!is_clipping && checkClipping(buffer[i])) {
        is_clipping = true;
      }
    }
    if (is_clipping) {
      cerr << "[" << count << "][" << getName() << "] clipping" << endl;
    }
  }

protected:

  void convertVectorToMatrix(const vector<short>& in, Matrix<float>& out,
     int first_col) {
    for (int i = 0; i < out.nrows(); i++) {
      for (int j = first_col; j < out.ncols(); j++) {
        out(i, j) = (float) in[i + (j - first_col) * out.nrows()];
      }
    }
  }

  bool checkClipping(short x) {
    if (x >= SHRT_MAX || x <= SHRT_MIN) {
      return true;
    } else {
      return false;
    }
  }

};
```

次に示すのは，Recorder クラスを NewDevice が扱えるように派生させたクラスのソースコードの概略である．initialize 関数でデバイスとの接続を行い，() オペレータにデバイスからデータを読み込む処理を記述する．このソースコード（NewDeviceRecorder.cpp）は `librecorder` フォルダ内に作成する．

```
#include "NewDeviceRecorder.hpp"

using namespace boost;

NewDeviceRecorder* NewDeviceRecorder::instance = 0;

// This function is executed in another thread, and
// records acoustic signals into circular buffer.
void NewDeviceRecorder::operator()()
{

  for(;;){

    // wait during less than (read_buf_size/sampling_rate) [ms]
    usleep(sleep_time);

    mutex::scoped_lock lk(mutex_buffer);
```

```
    if (state == PAUSE) {
      continue;
    }
    else if (state == STOP) {
      break;
    }
    else if (state == RECORDING) {
      lk.unlock();

      // Point5: ここにデバイスからデータを読み込む処理を記述する
      read_buf = receive_data;
      // Point6: これまでに読み込んだデータに対応する時間だけ cur_time を進める
      cur_time += timelength_of_read_data;


      mutex::scoped_lock lk(mutex_buffer);
      buffer.insert(buffer.end(), read_buf.begin(), read_buf.begin() + buff_len);

      lk.unlock();
    }
  }

}

int NewDeviceRecorder::Initialize(const string& device_name, int chan_count, int samp_rate, size_t buf_size)
{

// Point7: 使用する変数やデバイスを初期化する処理を記述する
  new_device_open_function

}
```

Discussion

AudioStreamFromMic モジュールは Recorder クラスが持つバッファの内容を読み込む処理だけをしており，デバイスとのデータのやり取りは Recorder クラスから派生した各クラス（ALSARecorder, ASIORecorder, WSRecorder）が行っている．そのため，これらの派生クラスに変わるクラス（NewDeviceRecorder）を実装することで，新たなデバイスをサポートすることができる．

See Also

ノードの作り方の詳細は 13.1 節「ノードを作りたい」に記述されている．NewDeviceRecorder クラスの作り方は，例えば ALSARecorder クラス等のソースコードを参考にすると良い．

第5章　入力データの作成

5.1　多チャネル録音したい

Problem

マイクロホンアレイなどからの多チャネル音響信号を録音したい.

Solution

対応デバイス

多チャネル録音を行うためには，多チャネルの入力を同期して入力できるデバイスを用いる必要がある. 現在, HARK を用いて録音に用いることができるデバイスは以下のとおりである. 各デバイスの詳しい説明や接続方法は HARK ドキュメントの第 8 章を参照。

ALSA Advanced Linux Sound Architecture (ALSA) を通してアクセスできるオーディオデバイス,

RASP System In Frontier, Inc. RASP シリーズ,

TDBD 東京エレクトロンデバイス製 TD-BD-16ADUSB.

2 つの録音方法

HARK では 2 種類の方法で録音ができる。ひとつは、録音を行うためのネットワーク (HARK のプログラム) を作成して録音する方法。もうひとつは、wios というサポートツールを用いて録音する方法である。以下ではそれぞれについて説明する。

HARK のプログラムによる録音

HARK には、録音デバイス (A/D 変換器) から音声を取得するノード AudioStreamFromMic と、波形を記録する SaveRawPCM がある (ノードの詳細は HARK ドキュメント参照). これらを直接接続すれば，マルチチャネルの録音が可能となる. 詳細は はじめての録音を参照。

wios による録音

HARK ではサポートツールとして、wios という ALSA, RASP, TDBD に対応した録音・再生用ツールを提供している. 最大の特徴は、再生と録音を同時に行えることである. 現状の HARK の定位、分離ノードは、音源から各マイクへ音が伝わる過程 (伝達関数 or インパルス応答) を事前に計測することを前提としている. そのためには、計測用信号を再生し、同時に録音することが必須である。ここでは録音の方法だけを解説するので, インパルス応答の計測の詳細はレシピ: インパルス応答を録音したいを参照。

HARK のリポジトリを登録していれば、wios は次のコマンドでインストールできる. (登録方法は HARK installation instructions を参照)

```
sudo apt-get install wios
```

wios のオプションは 4 つに分類できる詳しくはオプションなしの実行で表示されるヘルプを参照。

動作モードの指定 :

再生 (-p), 録音 (-r), 同期再生録音 (-s) の 3 種類. -t で動作時間を指定.

ファイルの指定 :

再生 (D/A) に用いるファイル名 (-i), 録音に用いるファイル名 (-o)
量子化ビット数 (-e), サンプリング周波数 (-f), チャネル数 (-c)

デバイスの指定 :

使用するデバイスの種類の指定 (-x) (ALSA:0, TDBD:1, RASP:2)
使用するデバイスの指定 (-d) (ALSA: デバイス名。デフォルト値は plughw:0,0. TDBD: デバイスファイル。デフォルト値は/dev/sinchusb0. RASP: IP アドレス。デフォルト値は 192.168.33.24.)

デバイス毎のオプションの指定 :

ALSA: バッファのサイズなど。 TDBD: ゲインなど。 RASP: ゲインなど。

以下に使用例を示す。

- IP アドレスが 192.168.1.1 の RASP から...
  1. 8 チャネルの音声を 3 秒録音する。出力ファイル名は output.wav
     ```
     wios -r -x 2 -t 3 -c 8 -d 192.168.1.1 -o output.wav
     ```
  2. input.wav を再生する。
     ```
     wios -p -x 2 -d 192.168.1.1 -i input.wav
     ```
  3. tsp.wav を同期再生録音する。出力ファイル名は response.wav
     ```
     wios -s -x 2 -d 192.168.1.1 -i tsp.wav -o response.wav
     ```
- コンピュータのサウンドカード (ALSA 対応) から...
  1. モノラルで 10 秒間録音
     ```
     wios -r -x 0 -t 10 -c 1 -o output.wav
     ```
  2. output.wav を再生 wios -p -x 0 -i output.wav

See Also

HARK のネットワークファイルを使った録音の方法ははじめての録音を参照。また、録音に失敗する典型的な理由はうまく録音できないを参照。

デバイスの使い方は、HARK ドキュメントのデバイスの章を参照. 新しいデバイスを使って録音をしたいなら、次のレシピ: 別の A/D 変換器を使いたい を読むとよい.

インパルス応答の測定方法はインパルス応答を計測したいを参照.

5.2　インパルス応答を計測したい

Problem

音源定位や音源分離で使用するためにインパルス応答を測定したい．

Solution

用意するもの

インパルス応答の測定には，スピーカ，マイクロフォンアレー，同期再生録音可能なオーディオデバイス，TSP 信号が必要である．1 回分の TSP 信号は，harktool をインストールしたときに，`/usr/bin/harktool3_utils` へインストールされる．インストールの方法は HARK Installation Instructions を参照．TSP 信号のファイル名は `16384.little_endian.wav` である．このままだと TSP 信号が 1 回分しか含まれないので，必要な回数 (例えば 8 回分や 16 回分) だけ波形編集ソフトウェアや matlab, python などでカット＆ペーストする．このとき，TSP 信号同士に空白を入れてはいけない．

TSP 信号の再生回数は、録音する環境に応じて決めるとよい．たとえば、雑音が小さい環境で測定するなら 8 回で十分だが、雑音が大きい場所、たとえばエアコンがずっと運転している場所など、なら 16 回再生するとよい．

これと等価な, `16384.little_endian.tsp` という 32bit float の波形データがリトルエンディアンで並んだ raw ファイルもある．このファイルを読み込みたいときは，`wavesurfer` うとよい．(インストールは `sudo apt-get install wavesurfer`) 次に，`wavesurfer` でファイルを開き，次のコマンドで開く `wavesurfer /usr/bin/harktool3_utils/16384.little_endian.tsp` 図 5.1 のように設定すれば波形として読み込める．

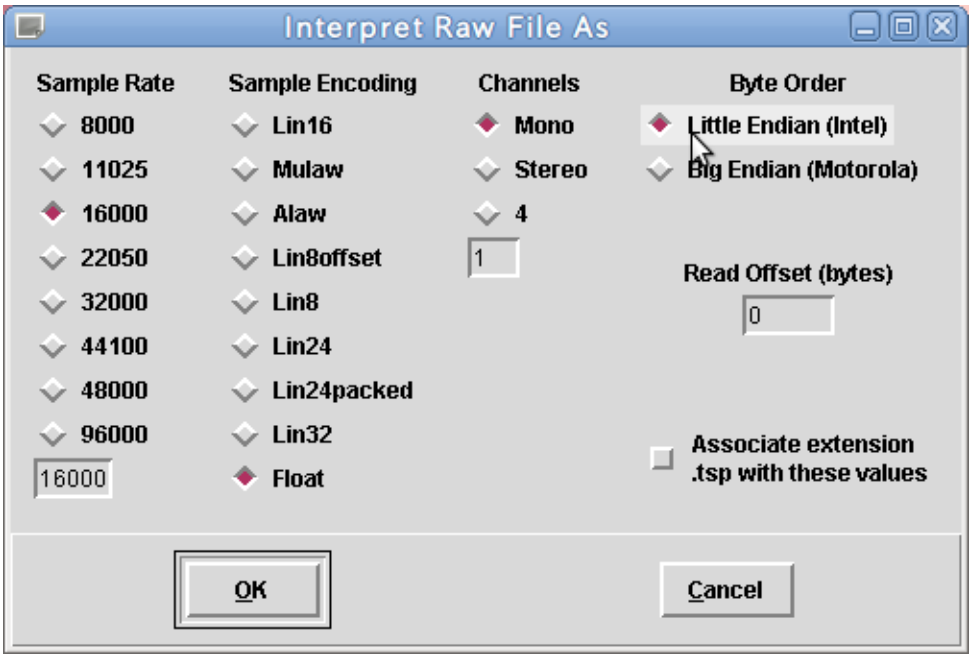


図 5.1: raw ファイルの読み込み時の設定

インパルス応答測定手順

インパルス応答の測定は 2 つのステップから成る．(1) TSP 信号の録音・再生と，(2) TSP 信号からのインパルス応答の計算である．

1. TSP 信号の録音・再生
   TSP 信号を複数回再生し，それぞれの信号を加算することで暗騒音や残響の影響を軽減できる． 8 回の TSP を含むファイルを `tsp8.wav` とする．

   あとは，HARK のサポートツールのひとつ wios を使えば測定ができる．たとえば 8 チャネルの入力をもつ ALSA 対応デバイスで同期録音再生を行うには，以下のコマンドを実行すれば良い．

   ```
   wios -s -x 0 -i tsp8.wav -c 8 -o r100_d000.wav
   ```

   詳しくはレシピ: 多チャネル録音したいを参照.

   インパルス応答は，各方向からの音をすべて測定する必要がある．マイクロホンアレーの中心から 1-2 メートルぐらいの同心円上に，等角度で順番にスピーカを動かし，録音していく．したがって，マイクアレイから距離 1m で 10 度間隔で計測した場合のファイル名は例えば次のようになる．

   ```
   r100_d000.wav
   r100_d010.wav
   r100_d020.wav
   ```

2. インパルス応答の計算
   こうして録音した信号に，逆 TSP 信号を畳み込み，複数回の再生を同期加算して平均値を求めることでインパルス応答を求める．インパルス応答の計算には harktool を用いればよい．詳しくは HARK ドキュメントの harktool の章を参照.

Discussion

インパルス応答とは，システムにインパルス関数を入力に与えたときの出力のことである．音響処理の場合は，たとえば広い部屋で手を叩いた (=インパルス) ときの残響がインパルス応答だと思えば良い．しかし，本当にインパルスを与えて計測しようとすると，相当に大きな音でなければ十分な信号対雑音比 (S/N 比) を得られない．そこで，かわりにエネルギーを時間で分散させた Time Stretched Pulse (TSP) 信号を用いて計測する．こうして録音した TSP 信号に，元の TSP 信号を逆にした逆 TSP 信号を畳み込むことでインパルス応答を求めることができる [1, 2].

角度の間隔は，マイクロホンアレーの配置にもよるが，5-10 度間隔ぐらいで録音すれば充分だろう．

See Also

wios による同期再生録音についてはレシピ: 多チャネル録音したいを参照.
harktool の使い方は，hark-document の サポートツールの章を参照.

(1) Y. Suzuki, F. Asano, H.-Y. Kim, and Toshio Sone, ”An optimum computer-generated pulse signal suitable for the measurement of very long impulse responses”, J. Acoust. Soc. Am. Vol.97(2), pp.-1119-1123, 1995

(2) TSP を用いたインパルス応答の測定 http://tosa.mri.co.jp/sounddb/tsp/index.htm

5.3 インパルス応答から音声データを合成したい

Problem

シミュレーションによりデータを作成し，オフラインで動作テストを行いたい．

Solution

手元に音源の波形データと，インパルス応答ファイルがある場合には，マルチチャネルデータを合成できる．

合成は, 音源から各マイクロホンまでのインパルス応答を音源データに畳み込むことによって, 行われる．畳み込みは, FFT を用いた循環畳み込み法などで高速に実現できる．Matlab などを利用することも可能である．以下に Matlab の擬似コードを示す．

```
x=wavread('SampleData1ch.wav');
y1=conv(x,imp1);
y2=conv(x,imp2);
y3=conv(x,imp3);
y4=conv(x,imp4);
```

SampleData1ch.wav に Microsoft RIFF format で 1ch 音声が収録されているものとする．また，imp1,...,imp4 は，音源からマイクロホン 1,...,4 までのインパルス応答の時間表現とする．y1,y2,y3,y4 が，シミュレーション合成したマルチチャネルデータである．

合成する前に, インパルス応答と音源データのサンプリング周波数を確認しておくこと．同一サンプリング周波数のデータを畳み込む必要ある．

混合音を合成する場合は，畳み込んだデータ同士を加算合成すれば良い．

Discussion

ここで付加されるのは、乗法性ノイズである。乗法性ノイズとは、クリーン音声に畳み込まれる歪みのことで、音声の伝達特性や録音機器の特性などが含まれる。従って、ここでは実際に使用するシステムの乗法性ノイズをシミュレートしていると言い換えることができる。

See Also

インパルスを計測したいなら、次のレシピを参照: インパルス応答を計測したい．もう一つの雑音である加法性雑音を付加したいなら、次のレシピを参照: ノイズデータを加えたい．

5.4 ノイズデータを加えたい

Problem

シミュレーションで実験するために, ロボット動作ノイズやファンノイズを加えたい.

Solution

音源定位・音源分離などを実際のシステムに適用するときは，そのシステムの周囲や、システム自体、録音機器などから加法性ノイズが入る。例えばロボットなら、ロボットの周囲の騒音、ロボットのファンノイズなどである。そこで、あらかじめそれらのノイズを実際に使うマイクロホンで録音しておくことで、より実際の使用に近いシミュレーションができる。

8ch のデータを合成したいで生成したインパルス応答畳み込み音声に、あらかじめ録音したノイズを加算することで、加法性ノイズをシミュレートできる。

Matlab 擬似コードを以下に示す.

```
convolved=wavread('signal.wav');
noise=wavread('noise.wav');
data=convolved+noise;
```

data がノイズを加えたシミュレーション音声データである.

Discussion

ここでいうノイズとは、ノイズの無い音声波形に加算されるノイズ (加法性ノイズ) のことである。例えば、ロボットのファンノイズは加法性ノイズのひとつである。

See Also

もう一つの種類のノイズである乗法性ノイズを加えたいときは、次のレシピを参照: 8ch のデータを合成したい

第6章　音響モデルと言語モデル

6.1　音響モデルを作りたい

Problem

このレシピでは音声認識で用いる音響モデルの構築法を解説する．ロボットに HARK を導入した後に音声認識性能を向上させるために有用である．

Solution

音響モデルは，音素と音響特徴量の関係を統計的なモデルで表現したもので，音声認識の性能を大きく左右するデータである．通常，隠れマルコフモデル (Hidden Malkov Model, HMM) を用いることが多い．ロボット搭載マイクロホンのレイアウトを変更したり，分離や音声強調のアルゴリズムやパラメータを変更する場合には，音声認識に入力される音響特徴量の性質が全体的に変わることが多いため，音響モデルをその条件に適応させたり，新規にその条件に合わせて作成したりすることによって音声認識性能を向上することができる．

ここでは，HARK で用いている音声認識エンジン Julius の音響モデルを作成する際に用いられる Hidden Malkov Model ToolKit (HTK) を使って，次の３通りの音響モデル (HMM) 構築方法を解説する．

1. マルチコンディション学習
2. 追加学習
3. MLLR/MAP 適応

なお，実際には音響モデルには様々なパラメータがあるが，ここでは 3 状態，16 混合の triphone HMM の学習を例に挙げる．各パラメータの詳細については,『HTK Book』,『IT Text 音声認識システム』他，多数の教科書が出版されているので，そちらを参考にされたい．

6.1.1　マルチコンディション学習

典型的なトライフォンベースの音響モデルの作成の基本的な流れを下記に示す．

1. 学習データの生成
2. 音響特徴量抽出
3. モノフォンモデルの学習
4. コンテキスト非依存トライフォンモデルの学習
5. 状態クラスタリング

6. コンテキスト依存トライフォンモデルの学習

マルチコンディション学習では，クリーンな音声信号に加えて，ロボットで収録した音声を用いて学習を行う．HARK で音声認識に用いる音声データは，マイクロホンアレイを用いて収録した音声データに対して，音源分離や音声強調を行った後のデータである．このため，学習データも同様に音源分離や音声強調を行った後のデータを用いる．しかし，音響モデルの学習には大量の音声データが必要なため，実際にこのようなデータを収録することは現実的ではない．このため，音源とマイクロホンアレイの伝達系のインパルス応答を計測しておき，このインパルス応答をクリーンな音声に畳み込むことよって，マイクロホンアレイを用いて収録した音声データを擬似的に生成して用いる．具体的なデータの作成は，5.2 – 5.4 節を参照されたい．

音響特徴量抽出

音響特徴量は メル周波数ケプストラム係数 (MFCC) が用いられることが多い．HARK では，MFCC を用いることもできるが，メルスケール対数スペクトラル係数 (MSLS) を用いることを推奨している．MSLS は HARK のネットワークを用いて，wav ファイルから簡単に作成することができる．MFCC も同様に HARK のネットワークを用いて作成できるが，HTK では MFCC を抽出するため，同様のツール HCopy が提供されており，MFCC を抽出するという点では，HARK よりもパラメータの設定自由度が高い．HCopy の使い方は，HTKBook など他の文献を参考にされたい．いずれにしても，特徴量抽出後は，HTK を用いて音響モデルを作成する．詳しくは，10 節を参照されたい．

辞書，MLF(Mater Label File) の準備

1. **データの修正:**

   一般的に，配布されているコーパスを用いる場合でも，完全に表記の揺れや，記述エラーを取り除くことは難しい．事前に気が付くことは難しいが，こうしたエラーが性能の低下につながるため，発見し次第修正することが望ましい．

2. **words.mlf の作成:**

   特徴量に対応する（仮想的な）ラベル用のファイル名とそのファイルに含まれる発話を単語単位で書き下した words.mlf を作成する．words.mlf ファイルは，ファイルの先頭行が #!MLF!# となっている必要がある．各エントリは，“ ” で囲んだラベルファイル名を先頭行に記述した後，そのラベルファイル名に含まれる発話を単語ごとに行を分けて記述する．また各エントリの最後の行には 半角ピリオド “. ” を記述する．

```
#!MLF!#
"/data/JNAS/pb/bm001a02.lab"
IQSHU:KANBAKARI
SP
NYU:YO:KUO
SP
SHUZAISHITA
.
"/data/JNAS/pb/bm001a03.lab"
TEREBIGE:MUYA
SP
```

```
PASOKONDE
SP
GE:MUO
SP
SHITE
SP
ASOBU
.
```

3. **単語辞書の作成:**

   音素列と単語を関係付ける単語辞書を作成する．単に，音素列とそれに対応する単語を以下のように記述する．

```
AME a m e
TOQTE t o q t e
  :
  :
SILENCE sil
SP      sp
```

4. **音素 MLF の作成:**

   辞書と単語 MLF を用いて，音素 MLF(phones1.mlf) を作成する．具体的には，HLEd を用いる．

```
  % HLEd -d dic -i phones1.mlf phones1.led words.mlf
```

   phones1.led には，例えば以下のようなルールを記述する．

```
EX
IS silB silE
```

   音素 MLF のフォーマットは，行の単位が単語から音素に変わったことを除けば，単語 MLF とほぼ同じである．phones1.mlf の例を以下に示す．

```
------------------------------
#!MLF!#
"/data/JNAS/pb/bm001a02.lab"
silB
i
q
sh
u:
k
a
N
```

```
 :
(中略)
 :
sh
u
z
a
i
sh
i
t
a
silE
.
"/data/JNAS/pb/bm001a03.lab"
 :
 :
```

特徴量ファイルのリスト train.scp の準備

基本的には，特徴量ファイル名を絶対パスで羅列（一行に 1 つのファイル名）したファイルを作成すればよい．ただし，特徴量ファイルの中身に異常な値が含まれる場合があるので，HList などを用いて値をチェックした上で正常なファイルのみをリストアップすることが望ましい．

triphone の準備

この作業は，monphone の学習をしてからでもよいが，チェックの結果次第では phones1.mlf を作り直す可能性があるので，時間を節約する意味でここで行う．

1. **tri.mlf の作成:**

   まず，単純に音素の三つ組みを作成する．

   ```
   % HLEd -i tmptri.mlf mktri.led phones1.mlf
   ```

   mktri.led の例を以下に示す．mktri.led に記述している音素は音素コンテクストから除かれる．

   ```
   ------------------
   WB sp
   WB silB
   WB silE
   TC
   ------------------
   ```

   さらに，前後の長母音コンテクストは短母音コンテクストと同一視するなどの処理を行ってパラメータ数を減らす．作成された tri.mlf の例を示す．

```
------------------------------
#!MLF!#
"/data/JNAS/pb/bm001a02.lab"
silB
i+q
i-q+sh
q-sh+u: q-sh+u
sh-u:+k y-u:+k
u:-k+a u-k+a
k-a+N
a-N+b
N-b+a
b-a+k
a-k+a
k-a+r
a-r+i
r-i
sp
ny+u: ny+u
ny-u:+y y-u:+y
u:-y+o: u-y+o
y-o:+k
o:-k+u o-k+u
 :
 :
------------------------------
```

2. **triphones の作成:**

   triphones は tri.mlf に含まれる音素の三つ組みのリストに相当する.

   ```
   grep -v lab tri.mlf | grep -v MLF | grep -v "\."| sort | uniq > triphones
   ```

3. **physicalTri:**

   学習時 (tri.mlf) に出現しない音素コンテクストまで含めた triphone リスト.

4. **整合性のチェック:**

   triphones と physicaiTri のチェックを行う．このチェックは重要.

monophone の準備

1. **HMM のプロトタイプ (proto-ini) を作成:**

   proto は HTK のツール MakeProtoHMMSet を使って作成することができる．MSLS 用の proto-ini の例は以下の通り.

```
~o
<STREAMINFO> 1 27
<VECSIZE> 27<NULLD><USER>
~h "proto"
<BEGINHMM>
<NUMSTATES> 5
<STATE> 2
<MEAN> 27
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0
<VARIANCE> 27
<VARIANCE> 27
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.
<STATE> 3
<MEAN> 27
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.
<VARIANCE> 27
1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.
<STATE> 4
<MEAN> 27
0.0 0.0 0.0 0.0 0.0 0.0 0.0
<VARIANCE> 27
1.0 1.0 1.0 1.0 1.0 1.0 1.0
<TRANSP> 5
0.0 1.0 0.0 0.0 0.0
0.0 0.6 0.4 0.0 0.0
0.0 0.0 0.6 0.4 0.0
0.0 0.0 0.0 0.7 0.3
0.0 0.0 0.0 0.0 0.0
<ENDHMM>
```

初期モデルの作成

HCompV を用いて初期 HMM を作成する.

```
%   mkdir hmm0
%   HCompV -C config.train -f 0.01 -m -S train.scp -M hmm0 proto-ini
```

MSLS 用の config.train は以下の通り.

```
SOURCEKIND=USER
TARGETKIND=USER
RAWENERGY=F
```

この結果, hmm0/ の下に, train.scp に含まれる学習データ全てから分散と平均を学習した proto と vFloor（初期モデル）が作成される．データ量にもよるが，時間がかかるので注意.

1. **初期 monophone の作成:**

   - hmm0/hmmdefs

     hmm0/proto の h ”proto” の proto を各音素に置き換える．これを monophone1.list に含まれるすべての音素について行う．monophone1.list は sp を含む音素のリストである．

   - hmm0/macros

     macro というファイルを vFloor の中身を少し書き換えることで作成する．これは，データが足りないときなどのフロアリングに用いるファイルである．vFloor を macro という名前でコピーした後，MSLS では，下記を macro のヘッダとして追加する．一般には，ヘッダの記述は，hmmdefs の記述と同じになるようすること.（つまり proto の内容に依存する）

```
~o
<STREAMINFO> 1 27
<VECSIZE> 27<NULLD><USER>
```

monophone の学習

```
% cd ../
% mkdir hmm1 hmm2 hmm3
```

最低 3 回は繰り返し学習する．(hmm1 hmm2 hmm3)
* hmm1

```
% HERest -C config.train -I phones1.mlf -t 250.0 150.0 1000.0 -T 1 \
    -S train.scp -H hmm0/macros -H hmm0/hmmdefs -M hmm1
```

* hmm2

```
% HERest -C config.train -I phones1.mlf -t 250.0 150.0 1000.0 -T 1 \
    -S train.scp -H hmm1/macros -H hmm1/hmmdefs -M hmm2
```

* hmm3

```
% HERest -C config.train -I phones1.mlf -t 250.0 150.0 1000.0 -T 1 \
    -S train.scp -H hmm2/macros -H hmm2/hmmdefs -M hmm3
```

再度アライメントを取り直す場合は，この後行うこと．ここでは省略.

triphone の作成

1. **monophone から triphone の生成:**

```
% mkdir tri0
% HHEd -H hmm3/macro -H hmm3/hmmdefs -M tri0 mktri.hed monophones1.list
```

2. **triphone の初期学習:**

```
% mkdir tri1
% HERest -C config.train -I tri.mlf -t 250.0 150.0 1000.0 -T 1 -s stats \
          -S train.scp -H tri0/macro -H tri0/hmmdefs -M tri1 triphones
```

10 回程度繰り返し学習する.

クラスタリング

1. **2000 状態にクラスタリング:**

```
% mkdir s2000
% mkdir s2000/tri-01-00
% HHEd -H tri10/macro -H tri10/hmmdefs -M s2000/tri-01-00 2000.hed \
  triphones > log.s2000
```

ここで，2000.hed は下記の通り．1 行目の stats は 9.2 で得られる出力ファイル，また，thres はとりあえず，1000 前後の値に置き換えて，その後，実行 log を見て, 試行錯誤で状態数が 2000 になるように設定すること.

```
-------------------------------------------------------------
    RO 100.0 stats

    TR 0

    QS "L_Nasal"   { N-*,n-*,m-* }
    QS "R_Nasal"   { *+N,*+n,*+m }

    QS "L_Bilabial"       { p-*,b-*,f-*,m-*,w-* }
    QS "R_Bilabial"       { *+p,*+b,*+f,*+m,*+w }
           ...

    TR 2

    TB thres "TC_N2_" {("N","*-N+*","N+*","*-N").state[2]}
    TB thres "TC_a2_" {("a","*-a+*","a+*","*-a").state[2]}
           ...

    TR 1
```

```
  AU "physicalTri"
  ST "Tree,thres"
-----------------------------------------------------------------
```

QS 質問

TB ここに書いたものがクラスタリングの対象となる．この例では，中心音素が同じもので同じ状態にあるもの以外はまとめない．

thres 分割しきい値適当に変化させて (1000,1200 など) 最終状態数を制御（log をみて確認）

2. **学習:** クラスタリング後，学習を行う．

```
% mkdir s2000/tri-01-01
% HERest -C config.train -I tri.mlf -t 250.0 150.0 1000.0 -T 1 \
       -S train.scp -H s2000/tri-01-00/macro -H s2000/tri-01-00/hmmdefs \
       -M s2000/tri-01-01 physicalTri
```

3 回以上繰り返す．

混合数の増加

1. **混合数の増加（1mix → 2mix の例）:**

```
%  cd s2000
%  mkdir tri-02-00
%  HHEd -H tri-01-03/macro -H tri-01-03/hmmdefs -M tri-02-00 \
   tiedmix2.hed physicalTri
```

2. **学習:**

混合数増加後，学習を行う．

```
%  mkdir tri-02-01
%  HERest -C config.train -I tri.mlf -t 250.0 150.0 1000.0 -T 1 \
       -S train.scp -H s2000/tri-02-00/macro -H s2000/tri-02-00/hmmdefs \
       -M tri-02-01 physicalTri
```

3 回以上繰り返す．

これらの手順を繰り返し，混合数を 16 程度まで順次増加させる．なお混合数は，倍々で増加させるのがよい．(2mix → 4mix → 8mix → 16mix)

6.1.2 追加学習

追加学習では，マルチコンディション学習を行って作成した音響モデル，もしくはクリーン音響モデルを種の音響モデルとして用いる．その他については，マルチコンディション学習と基本的に同じである．

例えば，16 混合のトライホンモデルを種の音響モデルとして用いるものとする．この場合，まず追加学習用音声データをマルチコンディション学習と同様の手法で用意する．その後，マルチコンディション学習の混合数増加後の学習と同様に（HERest を用いる），追加学習用音声データを用いて追加的に学習を行う．

追加学習によって，もともと別の学習データを用いて学習した音響モデルであっても追加学習データを使った適応学習によって，初めから追加学習データを用いてマルチコンディション学習を行った場合に近い性能を得ることができる．基本的にマルチコンディション学習と同様の手法を用いて適応を行うため，追加学習用データは大量にあることが望ましい．大量の学習データがえられない場合，もしくは適応学習にかかる計算時間を短縮したい場合は，以下の MLLR/MAP 適応を用いる．

6.1.3 MLLR/MAP 適応

この手法は，種の音響モデルを用いる点では，追加学習と同じであるが，種の音響モデルから目的の音響モデルへの線形変換行列を推定することによって，適応を行う手法である．HTK 3.3 から適応学習の方法が HAdapt を用いる方法から HERest を用いる方法に変更になった．具体的な手法については，HTK Book を参考にされたい．

See Also

HTK Speech Recognition Toolkit
Julius の音響モデルに関する web ページ
このドキュメントのもとになった web ページ
CMU の Sphinx の音響モデル生成チュートリアル
Sphinx の音響モデル生成レシピ
HTK の音響モデル生成レシピ

6.2 言語モデルを作りたい

Problem

音声認識で用いる言語モデルの構築法について説明する．

Solution

言語モデルには，大きく統計言語モデル，ネットワーク文法モデルの二種類があり，Julius も両方を用いることができる（以前は前者が Julius, 後者が Julian という別の名前のバイナリであったが，ver 4 系から両者が Julius として統合されている）．ネットワーク文法の作成については，Julius のホームページに詳しい説明が載っているので，そちらを参考にされたい．以降では，統計言語モデルの作成を中心に述べる．

Julis 用の辞書の作成

辞書は正解テキストを形態素解析したものをベースに HTK 形式で作成する．形態素解析には chasen（茶筅）を使用する．

chasen をインストール後，Home ディレクトリに .chasenrc を作成して，その中の，grammar.cha があるディレクトリを「文法ファイル」に指定し，出力フォーマットを下記のように記述する．

```
(文法ファイル /usr/local/chasen-2.02/dic)
(出力フォーマット "%m+%y+%h/%t/%f\n")
```

正解テキストファイルを用意し，このファイルを seikai.txt とする．言語モデル作成時に使うので，文の最初と最後にそれぞれ <s>，</s> を挿入しておくこと．

```
seikai.txt の例 (単語間は区切られていなくても構いません)
<s> あらゆる 現実 を すべて 自分 の ほうへ ねじ 曲げ た の だ 。 </s>
<s> 一週間 ばかり ニューヨーク を 取材 し た 。 </s>
 :

% chasen seikai.txt > seikai.keitaiso
```

text.keitaiso の内容を見て，形態素解析が間違っている部分があれば修正する．また,「へ」,「は」は表記と読みが異なるので，読みをそれぞれ「エ」,「ワ」に直す．実際には，これ以外にも形態素の正規化，不要部分の除去などの作業をする必要があるが，ここでは省略する．

```
seikai.keitaiso の例

<s>+<s>+17/0/0
あらゆる+アラユル+54/0/0
現実+ゲンジツ+2/0/0
を+ヲ+59/0/0
すべて+スベテ+16/0/0
自分+ジブン+2/0/0
の+ノ+68/0/0
ほう+ホウ+21/0/0
```

へ+エ+59/0/0
ねじ曲げ+ネジマゲ+45/6/4
た+タ+71/55/1
の+ノ+21/0/0
だ+ダ+71/56/1
。+。+75/0/0
</s>+</s>+17/0/0
EOS++
<s>+<s>+17/0/0

つぎに,

```
% w2s.pl seikai.keitaiso > seikai-k.txt
```

によって, 各行の改行を半角スペースにすると共に, <s>+<s>+17/0/0 の部分を, <s> に </s>+</s>+17/0/0 は </s> に変換する. また EOS を改行に変換する. これによって形態素解析済の正解テキストが作成できる. このテキストは以降の言語モデルの作成に使用する.

seikai-k.txt の例

<s> あらゆる+アラユル+54/0/0 現実+ゲンジツ+2/0/0 を+ヲ+59/0/0 すべて+スベテ+16/0/0 自分+ジブン+2/0/0 の+ノ+68/0/0 ほう+ホウ+21/0/0 へ+エ+59/0/0 ねじ曲げ+ネジマゲ+45/6/4 た+タ+71/55/1 の+ノ+21/0/0 だ+ダ+71/56/1 。+。+75/0/0 </s>
<s> 一週間 ...

最後に seikai.keitaiso を利用して辞書を作成する.

```
% dic.pl seikai.keitaiso kana2phone_rule.ipa | sort | uniq > HTKDIC
% gzip HTKDIC
```

HTKDIC.gz が Julius で使用する辞書となる. Julius で使用する際には, これを -v オプションで指定すること.

HTKDIC の例

```
</s>    []      silE
<s>     []      silB
。+。+75/0/0    [。]    sp
あらゆる+アラユル+54/0/0        [あらゆる]      a r a y u r u
```

用語: 形態素解析, chasen, HTK 形式, w2s.pl, dic.pl, kana2phone_rule.ipa - vocab2htkdic に含まれているもの

Julis 用の言語モデルの作成

言語モデルの作成は, 基本的には,「音声認識システム」(オーム社) を参照のこと. ただし, サンプルの jconf にあるような 2-gram と逆向き 3-gram を作成するためには, CMU-Cambridge Toolkit だけでは不十分であり, CMU-Cambridge Toolkit 互換の palmkit を使うとよい. また最近では, julius は逆向き 3-gram は不要となったようであるので, palmkit を使う必要は必ずしもないかもしれない.

palmkit の使い方は以下の通り．正解テキストを用意し，seikai-k.txt とする．このファイルは形態素解析済であることが必要である．(つまり，句読点も一単語とし単語間をスペースで区切る．) 文の最初と最後にそれぞれ <s>, </s> を挿入してあるのは前述の通り，<s> や </s> をまたがるような遷移を取り除くためである．この場合，learn.css ファイルには，<s>, </s> の記述が必要である．

```
% text2wfreq < learn.txt > learn.wfreq
% wfreq2vocab < learn.wfreq > learn.vocab
% text2idngram -n 2 -vocab learn.vocab < learn.txt > learn.id2gram
% text2idngram -vocab learn.vocab < learn.txt > learn.id3gram
% reverseidngram learn.id3gram learn.revid3gram
% idngram2lm -idngram learn.revid3gram -vocab learn.vocab -context learn.ccs -arpa learn.rev3gram.arpa
% idngram2lm -n 2 -idngram learn.id2gram -vocab learn.vocab -context learn.ccs -arpa learn.2gram.arpa
```

これで，2-gram と逆向き 3-gram ができる．最後にこれらを一つにまとめる．julius のツール mkbingram を用いて，以下のようにすると Julius 用の言語モデルが作成できる．

```
% mkbingram learn.2gram.arpa learn.rev3gram.arpa julius.bingram
```

See Also

このドキュメントのもとになった web ページ

第7章　FlowDesigner

7.1　コマンドラインから引数を与えて起動したい。

Problem

FlowDesigner は GUI 上でネットワークを構築し, その場で実行できるが, 処理の度に GUI を起動し実行ボタンを押すのは面倒くさい. そこで，ネットワークファイルをコマンドラインから実行したい．その時，ネットワークのパラメータをコマンドライン引数から変更したい.

Solution

コマンドラインから引数を与えて処理を行うためには，対象となるパラメータとコマンドライン引数の対応をネットワークファイル内のノードに設定する必要がある．

まず FlowDesigner を開き，ネットワークファイルを次のように編集する

1. MAIN サブネットワークで，引数を渡したい変数を含むノードのプロパティを開く．
2. 引数が代入される変数の Type を `subnet_param` に， Value を “ARG?” と入力する．なお，?には引数のインデックスが入る．この際，引数が整数もしくは浮動小数点の場合には，型も含めて記述する．

例えば，一つ目のコマンドライン引数をあるパラメータに文字列として渡す場合は “ARG1” とし，二つ目のコマンドライン引数を別のパラメータに浮動小数点として渡す場合は “float:ARG2” とする．

ただし，コマンドライン引数を与えられるのは MAIN サブネットワークのノードのみである．他のサブネットワークのパラメータを指定するときは，まずそのネットワークのパラメータを `subnet_param` に指定して MAIN サブネットワークから見えるようにし，MAIN で指定する必要がある．

ネットワークファイルの内容を読み込んで実行するのは `batchflow` であるので，ネットワークファイル名の前に実行するアプリケーションの名前である `batchflow` を指定しなければならない．

```
$ batchflow foo.n 0.5 0.9
```

こうすると，`foo.n` の ARG1 に 0.5, ARG2 に 0.9 が渡されてネットワークが実行される．この方法は Windows であっても Ubuntu であっても同様に行うことができる．

For Ubuntu

Ubuntu でコマンドラインから実行する場合のみ，以下の様に `batchflow` を省略しても実行することができる．

```
$ ./foo.n 0.5 0.9
```

こうすると、`foo.n` の ARG1 に 0.5, ARG2 に 0.9 が渡されてネットワークが実行される．

For Windows

Windows でコマンドラインから実行する場合のみ，以下の様に `batchflow` を `qtflow` としても実行することができる．

```
$ qtflow foo.n 0.5 0.9
```

こうすると、`foo.n` の ARG1 に 0.5, ARG2 に 0.9 が渡されてネットワークが実行される．

Discussion

FlowDesigner はネットワークファイルを編集する GUI であり，実際にネットワークファイルを実行するのは Ubuntu では `batchflow` であり，Windows では `qtflow` である．ただし，Windows で `batchflow` とコマンドを打っても実際に内部では `qtflow` が実行される．そのため，どちらの OS でも同じ方法で実行することができる．

コマンドライン引数は，すべて `string` 型と解釈される．`int` 型や `float` 型で渡したい時には，Value に “`int` :ARG1”， ”`float` :ARG1” のように指定する．現在のバージョン (Version 1.2) では，オブジェクト，たとえば `Vector<int>` などの `Object` 型にはこの方法では渡すことはできない．

See Also

なし．

7.2 他のネットワークファイルからノードをコピーしたい

Problem

他のファイルからノードをコピー・ペーストできない.

Solution

他のファイルから設定値を保持したノードをコピー・ペーストしたい場合はよくある.

FlowDesigner を 2 つ以上起動し，異なるウィンドウ間では，コピー・ペーストができない．同一ウィンドウ内でのコピー・ペーストは以下の手順で行うことが可能である.

1) `"ファイル" -> "開く"`
   でコピーしたいブロックの入ったネットワークファイルを起動

2) コピーしたいブロックをドラッグで選択し，
   `"編集" -> "コピー"`

3) コピー先で `"編集" -> "貼り付け"`

また，以下のショートカットも有効である.

- `Ctrl-c`：コピー
- `Ctrl-x`：カット
- `Ctrl-v`：ペースト

Windows 版ではコピー・ペースト機能に対応していない.

7.3 for ループのように指定した回数だけ繰り返し処理をしたい

Problem

for ループのように、決めた回数だけ処理を繰り返すネットワークを作成したい。

Solution

Iterator サブネットワークと、Iterator ノードを使えば実現できる。例えば, ConstantLocalization を 500 回だけ DisplayLocalization に渡したいときは、図 7.1 のようにネットワークを作成すればよい。Iterator ノードは New Node → Flow → Iterator にある。

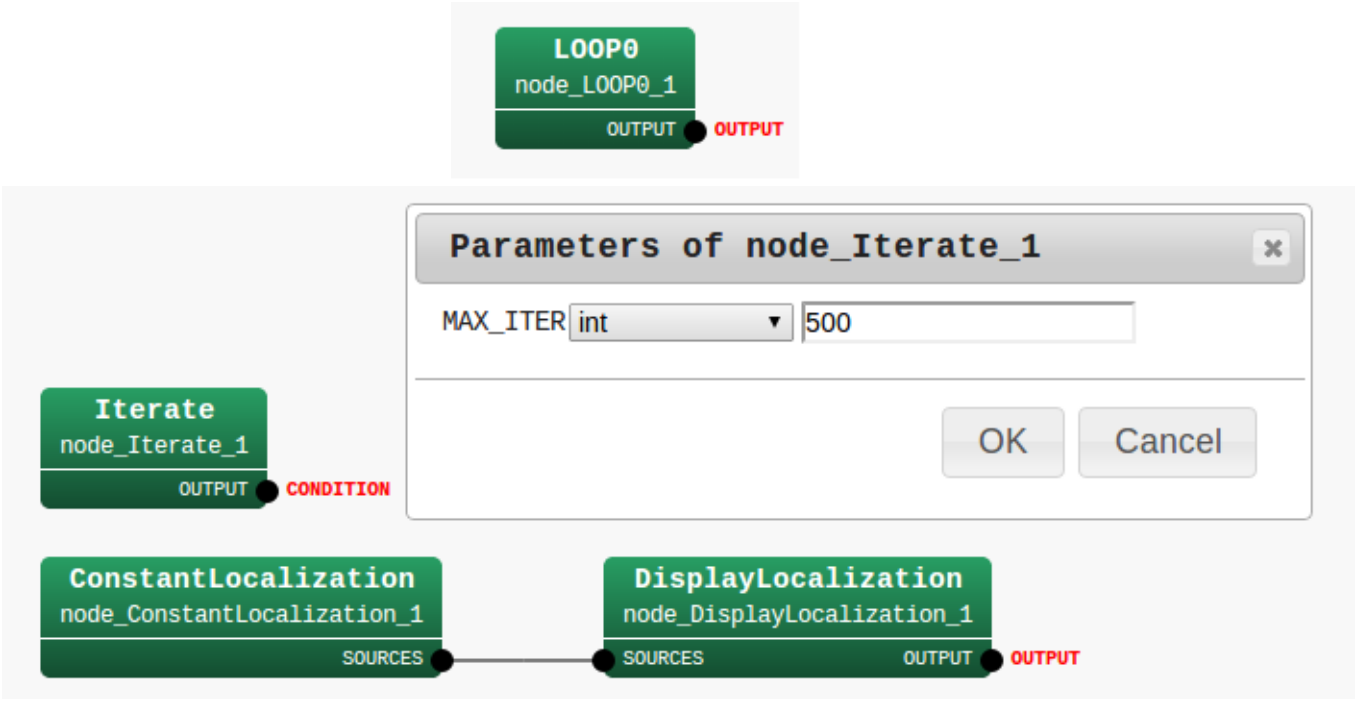


図 7.1: サンプルネットワーク: 左が MAIN サブネットワーク, 右が Iterator サブネットワーク

このネットワークを実行して、一定の音源定位結果が、Iterator で指定したフレーム数だけ表示されれば、このネットワークは正しく動いている。

Discussion

Iterate ノードは、各繰り返しごとに MAX_ITER パラメータに与えた値を 1 ずつ減らしていく。値が 0 より大きければ `true` を出力し、0 になれば `false` を出力する。この出力を 終了条件端子に指定することで、指定回数だけ繰り返す処理が実現できる。

See Also

FlowDesigner のヘルプ。FlowDesigner を起動し、ヘルプメニューから User Guide を選ぶと閲覧できる。

第8章　音源定位

8.1　はじめに

o

Problem

マイクロホンアレイを使って，音源の方向を推定したい．

Solution

HARK では音源定位機能に加えて，音源の追従，定位結果の可視化，保存，読み込み機能も提供している．本章では，それらに関するレシピをまとめている．

あなたが初めて音源定位を行うのなら，まずははじめての音源定位を見ながら定位システムを作成してみよう．それができたら，次はデバッグをしよう．レシピ: 定位できているかどうかを確認したいを見て動作確認をしよう．もし問題があれば，音源定位がまったくでない / 出すぎる，音源定位結果が細かく切れてしまう / 全部繋がってしまうなどを見よう．音源分離に接続するときは，音源分離した音の先頭が切れてしまうという問題が起こるかもしれない．定位結果の解析などのために保存したいならレシピ: 定位結果をファイルに保存したいを見よう．

音源定位の性能をあげるには，レシピ: 音源定位のパラメータをチューニングしたいを見てチューニングしよう．音源数が複数ある場合は，同時に複数の音を定位したいを使ってパラメータを変えればよい．

音源定位システムは，拡張することもできる．例えば 3 次元の音源定位をしたいなら，音源の高さや距離も推定したいを見よう．また，マイクアレイのうちいくつかのマイクのみを使いたい時は，レシピ: マイクロホンアレイの一部だけを使いたい を見よう．

Discussion

音源定位に関わる主なノードは以下の 7 つである．

音源定位 LocalizeMUSIC
　入力された音声から音源定位結果を出力

固定の音源定位 ConstantLocalization
　固定した音源定位結果を出力．分離のデバッグなどに使える．

同一音源追従 SourceTracker
　音源定位結果に対して，音源位置を基準に同一音源か否かを判定し，音源 ID を付与

定位結果の表示 DisplayLocalization
　音源定位結果を表示

定位結果の保存と読み込み SaveSourceLocation と LoadSourceLocation

音源定位結果を保存・読み込み

音源定位の引き延ばし SourceIntervalExtender

音源定位結果の引き延ばし．分離用．

See also

各ノードの使い方については，HARK ドキュメントの当該箇所を参照．MUSIC による音源定位の原理は HARK ドキュメントの LocalizeMUSIC を参照．

8.2 音源定位のパラメータをチューニングしたい

Problem

音源定位の性能をパラメータの調節で改善したい。

Solution

音源定位で起こりうる問題ごとに，解決方法を示す.

Q.1) 定位方向がぐちゃぐちゃに表示される，又は全く表示されない．

DisplayLocalization で定位結果を表示すると，定位方向がぐちゃぐちゃに表示されることがある．これは，パワーの低いものまで音源として定位してしまっているためである．全く表示されない場合はその逆である.

A.1-1) **SourceTracker の** `THRESH` **を変更する**

音源方向の期待値の閾値を直接変更するパラメータであるので，うまく音源のみのピークをとらえるように調節を行う．調節には，LocalizeMUSIC の DEBUG オプションを `true` にすれば MUSIC スペクトルが表示されるので，それを使うとよい.

A.1-2) **LocalizeMUSIC の** `NUM_SOURCE` **を目的音の数に合わせる**

MUSIC は NULL スペースを利用して目的音方向のピークをより大きくするため，`NUM_SOURCE`（音源数）の設定次第で引き立たせるピークの個数の設定が変わることになり，これを誤って設定すると，ノイズ方向にピークが立ったり，目的音方向にうまくピークが立たなかったりと性能が劣化してしまう.（実際にはピークがなまるのみで，使用できる場合が多い.）話者一人の場合は 1 と設定するのが，より性能が向上する.

Q.2) 複数音あるのに一つしかピークが立たない

たとえば 2 つの音源があるのにもかかわらず音源定位結果が 1 つしか結果が出ないことがある．この原因は，音源同士が近すぎるか，音源数がネットワークファイルのパラメータと合っていないからである.

A.2-1) **SourceTracker のを** `MIN_SRC_INTERVAL` **変更する**

近いところに音源がある場合，例えば到来方向 10 度しか離れていない二音源に対しては，パラメータ `MIN_SRC_INTERVAL` を 10 度以下に指定する必要がある.

A.2-2) **LocalizeMUSIC の** `NUM_SOURCE` **を目的音の数に合わせる**

A.1-2 と同様．ただし，音源の音量が十分で，40 度程度離れていれば，経験的にはネットワークの音源数設定と異なっていてもうまく動くことが多い.

Q.3) 音声以外の音を使っている

LocalizeMUSIC の `LOWER_BOUND_FREQUENCY`，`UPPER_BOUND_FREQUENCY`

音源定位はこの二つで指定される周波数の中で，相関行列を足し合わせて平均を取る方法を用いているため，定位したい音源と全く異なる周波数を設定していると，平均した時にピークがなまってしまう．音源に合わせた周波数を用いる．音声であればデフォルトの $500[Hz] \leq f \leq 2800[Hz]$ にしておけばよい.

Q.4) 特定の範囲は音源が来ないと仮定できるので無視したい

LocalizeMUSIC の `MIN_DEG`，`MAX_DEG`

音源定位はこの二つの値で指定された方向の範囲内での定位のみしか行わない．360 度の定位を行いたい場合は 180 度と-180 度に指定する.

Discussion

上記の Solution はノードのパラメータの調整による音源定位のチューニングである．しかし，音源定位を行う部屋の残響が伝達関数測定時と大きく異なる場合は，その部屋で伝達関数の再測定を行うとよい．伝達関数の測定方法は動画を容易しているので HARK Web ページを参照.

See Also

音源定位の具体的なアルゴリズムや, パラメータの具体的な説明については HARK ドキュメントの LocalizeMUSIC を参照.

8.3 マイクロホンアレイの一部だけを使いたい

Problem

信号処理に使うマイクロホンを指定して音源定位を実行したい.

Solution

例えば、マイクアレイに使うマイクを間引いて性能を比較したい時がある．このときは，ChannelSelector の SELECTOR パラメータで使うチャネルを選択することで実現する．パラメータの型を `Object` に指定し，音源定位に使用したいマイクロホンのチャネル番号のみを `Vector<int>` で指定する (例: 4ch のうち 1, 3 番目のマイクロホンを指定する場合は, `<Vector<int> 0 2>` とする.)

Discussion

なし.

See Also

ノードの詳しい挙動は，HARK ドキュメント: ChannelSelector を参照

8.4 同時に複数の音を定位したい

Problem

- 同時に複数の音が鳴ったときにうまく定位できない
- LocalizeMUSIC モジュールで音源数のプロパティにどのような値を入れたらよいのか知りたい

ときに読む.

Solution

手法の制約により，同時に定位できる音源数は利用するマイクロホン数以下である．もし，定位したい目的音源数が N の場合，この値は N に設定する．さらに，M ノイズが特定の方向から常に聞こえてくると仮定出来るならば，$M + N$ に設定する．

音源数の指定は，LocalizeMUSIC モジュールの NUM_SOURCE パラメータを通じて行う．

Discussion

理論上，MUSIC 法はマイクロホン数以下の音源を同時に定位することが出来る．つまり，8 つマイクロホンを用いた場合，最大音源数は 8 である．しかし，実験的には 3 ～ 4 個の音源までなら安定して同時に定位することが確認されている．

8.5 定位できているかどうかを確認したい

Problem

- LocalizeMUSIC モジュールによる音源定位が正しく行われているかを確認したい
- GHDSS モジュールによる音源分離など定位結果を利用する処理のパフォーマンスが悪い

ときに読む.

Solution

特定の方向から声やスピーカからの音を定位し，定位結果と照合することで確認する．定位結果の確認には，DisplayLocalization モジュールや， SaveSourceLocation を使用する．

定位精度を確認する際は， LocalizeMUSIC の MIN_DEG を -180 ，MAX_DEG を 180 に設定し，すべての方向から音を定位するように設定する．特定の方向から音は到来しないと仮定出来るならば，MIN_DEG, MAX_DEG を適宜設定することで音源方向に制約を与え，定位結果が安定する場合がある．

Discussion

定位精度を向上させるポイントは

1. マイクロホンアレーの伝達関数を実際に測定する
2. SourceTracker モジュールに適切なチューニングを行う
3. 音源が存在しないと分かっている角度からは定位しないように設定する

などが挙げられる．

See Also

- 定位結果をファイルに保存するにはどうしたらいい？
- SourceTracker の閾値はどうやって決める？
- HARK ドキュメント：DisplayLocalization モジュール
- HARK ドキュメント：SaveSourceLocation モジュール

8.6 音源定位がまったくでない / 出すぎる

Problem

- LocalizeMUSIC モジュールを使った定位がうまくいかない
- 音がなっていないのに特定の方向から音源が定位され続ける

Solution

SourceTracker モジュールの THRESH プロパティの値を適切に設定する．

1. LocalizeMUSIC モジュールの DEBUG プロパティを `true` に設定したネットワークファイルを実行する．
2. 実行中，まわりが無音の場合と，手を叩くなどで音をならした場合の MUSIC スペクトルのパワー値を見る．
3. 2 つの場合のパワー値の中間になる値を設定する．

ポイントは無音の場合の定常パワーより少し大きめの値に設定することである．例：定常パワーが 25.5 – −25.8 ぐらいなら THRESH は 26 にする，など．この際，手順 1. で出力された各時刻・方向の MUSIC スペクトルの値をスペクトログラムのように表示すると，適切な閾値を比較的容易に選ぶことが出来る．

Discussion

LocalizeMUSIC モジュールが出力する値はマイクロホンのゲインや周囲の環境に依存して変化するため，上記のように試行錯誤によって適切に設定する．

THRESH の設定には以下のトレードオフ関係がある：THRESH を小さい値にすると，パワーの小さな発話でも定位可能な反面，予期せぬノイズ (足音など) を定位することがある．THRESH を大きな値にすると，周りで突発音などがあっても定位結果として報告しない反面，発話音の定位にもより大きなパワーを要することになる．

See Also

- 定位できているかどうかを確認したい
- SourceTracker の PAUSE LENGTH はどうやって決めるのか？
- HARK ドキュメント：LocalizeMUSIC モジュール
- HARK ドキュメント：SourceTracker モジュール

8.7 音源定位結果が細かく切れてしまう / 全部繋がってしまう

Problem

- ひとつながりの音なのに定位結果がぶつぎれになる
- 音の定位結果がすべてつながってしまう

ときに読む.

Solution

SourceTracker モジュールの PAUSE_LENGTH プロパティの値を適切に設定すればよい.

1. SourceTracker モジュールを DisplayLocalization モジュールに接続し，定位結果を表示する.
2. 適当な文章を読み上げて定位結果を見る.
3. **定位結果が途切れる：** PAUSE_LENGTH の値を大きくする.
4. **定位結果がつながりすぎる：** PAUSE_LENGTH の値を小さくする.

Discussion

SourceTracker モジュールの PAUSE_LENGTH プロパティの目的は，発話の息継ぎなどで LocalizeMUSIC モジュールの MUSIC スペクトルのパワーが下がっても，連続した音声として定位し，適切に音声認識させることである．人の発話に限らず，実際は途切れている音でも連続した音として定位するために使用することが出来る．人の発話の定位が目的ならばデフォルトの値を使用すればよい.

PAUSE_LENGTH の単位　このパラメータの単位はミリ秒である．従って，PAUSE_LENGTH の値が実際に何 msec に対応するかは AudioStreamFromMic や AudioStreamFromWave モジュールで指定されるサンプリング周波数 (SAMPLING_RATE)，FFT のステップ幅 (ADVANCE) に依存する．すべてデフォルト (サンプリング周波数：16000 Hz，ステップ幅：160 pt) の場合，PAUSE_LENGTH の値を 1 変えることは，1 (msec) 変更することに対応する.

See Also

- SourceIntervalExtender の使い方は？
- HARK ドキュメント：SourceTracker モジュール

8.8 音源分離した音の先頭が切れてしまう

Problem

- 分離音の先頭部分が途切れてしまっているとき
- 分離音の先頭の無音区間が長すぎるとき
- SourceIntervalExtender モジュールをどのように使用すれば良いか分からないとき

に読む.

Solution

以下のように調節する.

1. 分離音をファイルに保存したい を参考に，分離結果を保存できるネットワークファイルを作成する．この際，LocalizeMUSIC モジュールなどの定位モジュール分離モジュール GHDSS の間に SourceTracker と SourceIntervalExtender モジュールをはさむ.
2. 音を分離して，その結果を表示する or 聞く
3. **分離音の先頭部分が途切れている：** PREROLL_LENGTH プロパティの値を上げる
4. **分離音の先頭の無音区間が長い：** PREROLL_LENGTH プロパティの値を下げる

Discussion

SourceIntervalExtender モジュールの役割は，音源定位が報告された時点で既に発話開始から 500 (msec) ほど時間が経過しているので，分離音の先頭部分を失ってしまい，音声認識に失敗するという問題を解消することである．音源定位の開始時点からどれだけ遡って分離するかを指定するために PREROLL_LENGTH の値を決定する.

PREROLL_LENGTH の値を減らしすぎると，分離音の先頭部分が切れて音声認識に悪影響を及ぼすが，PREROLL_LENGTH の値を増やしすぎると，一つ前の発話と繋がってしまい，音声認識で使用する言語モデルによっては，誤認識に繋がることがある.

PREROLL_LENGTH の単位　この値の単位は短時間フーリエ変換したときの 1 時間フレーム分に相当する．そのため，実際の時間 (秒やミリ秒) との対応関係は，AudioStreamFromMic や AudioStreamFromWave モジュールで指定されるサンプリング周波数 (SAMPLING_RATE)，FFT のステップ幅 (ADVANCE) に依存する．すべてデフォルト (サンプリング周波数：16000 Hz，ステップ幅：160 pt) の場合，PREROLL_LENGTH の値を 1 変えることは，10 (msec) 変更することに対応する.

See Also

- SourceTracker の PAUSE_LENGTH はどうやって決める？
- HARK ドキュメント：SourceIntervalExtender モジュール
- HARK ドキュメント：SaveRawPCM ，SaveWavePCM モジュール

8.9 音源の高さや距離も推定したい

Problem

- 音の到来について水平面での方向だけでなく，高さなどの情報も知りたい
- 音の方向に加えて，音源までの距離も知りたい

Solution

現時点での LocalizeMUSIC モジュールは水平面の到来方向のみを推定する．これに加えて，音源の高さや音源までの距離を推定するには，モジュールのプログラム自体を改造する必要がある．MUSIC アルゴリズム自体は水平面の角度などを仮定しないため，拡張は可能である．

ただし，定位したい情報に応じて伝達関数を用意しなければならない．たとえば，音源の到来する高さを推定する場合には，水平方向と高さを変化させた場合の音源からの伝達関数が必要となる．

また，マイクロホンアレーの形状にも配慮する必要がある．水平面の到来角推定であれば，水平面上に配置されたマイクロホンアレイがあればよいが，音源の高さの推定を考える場合は，高さが変化した時の音源到達時間差が取りやすいよう，三次元的にマイクロホンを配置する方が良い．

Discussion

MUSIC アルゴリズムは事前に与えられた伝達関数を元に音源位置などの推定を行うアルゴリズムなので，知りたい情報に応じて伝達関数を測定することで，高さの推定などにも応用可能である．ただし，HARK における実装は水平面の音源方向推定に限定されているので，適宜改造が必要となる．

See Also

- HARK ドキュメント：LocalizeMUSIC

8.10 定位結果をファイルに保存したい

Problem

- 定位結果をファイルに保存する方法がわからない

ときに読む.

Solution

SaveSourceLocation モジュールを, LocalizeMUSIC , ConstantLocalization , LoadSourceLocation などの定位結果を出力するモジュールの後に接続する.

FILENAME パラメータに, 定位結果保存先のファイル名を指定する.

Discussion

なし.

See Also

- HARK ドキュメント: SaveSourceLocation

第9章　音源分離

9.1　はじめに

Problem

マイクロホンで複数の音源を収録して，個々の音源に分離したい。

Solution

音源分離ノード GHDSS を使うことで，マイクロホンから入力された混合音を分離できる．なお，GHDSS は，音声波形と音源定位結果を入力にとるので，音源分離の前に音源定位を行う必要がある．

GHDSS による音源分離には、音源の位置から各マイクロホンまでの伝達関数が必要である。伝達関数の計算方法には、(1) TSP (time stretched pulse) 信号で実際に測定する方法と、(2) マイクロホンの位置情報からシミュレーションで計算する方法がある。

1. TSP 信号で測定する方法
   マイクロホンからある程度離れた円 (たとえば 1-2m ぐらい) を描き、その円周上からある程度の角度 (たとえば 5 度や 10 度) ごとにスピーカから TSP 信号を流し、収録する。それらと harktool を使って、伝達関数を計算する。 詳しくはインパルス応答を計測したい を参照。

2. シミュレーションで計算する方法
   インパルス応答を計測するかわりにマイクロホン座標 (MICARY-LocationFile) を作成する。マイクロホン座標ファイルの作成方法は、HARK ドキュメント の harktool の使い方と， マイクロホン配置だけから音源分離したいを参照.

伝達関数が準備できたら、GHDSS のパラメータを適切に設定する。詳しくは はじめての音源分離を見よう．音源分離した音を保存して聞きたいなら，分離音をファイルに保存したいを見よう．

音源分離の性能を向上させるには，音源分離のパラメータをチューニングしたい を見よう．また，音源分離を雑音がある環境下で使うときは，ファンノイズなどの定常ノイズのせいで音源分離がうまくいかないや， 分離音に入っている雑音を後処理で減らしたい．を見ておくとよい．移動する音源を分離したいときは，音源やロボットが移動する状況で分離したい が参考になるだろう．

Discussion

GHDSS は高次無相関化と音源定位結果に基づく幾何的制約を用いて音源分離を実現する。詳細は HARK document の GHDSS の項目を参照

See Also

HARK ドキュメント の GHDSS に手法の詳細が解説されている。また、HARK チュートリアル資料や音源分離のサンプルも参考になるだろう。

9.2　分離音をファイルに保存したい

Problem

分離した音を聞きたい、あるいは別の目的で使用したいので、ファイルに保存したい.

Solution

基本的に SaveRawPCM もしくは SaveWavePCM モジュールを使う. SaveRawPCM モジュールの出力形式はヘッダなし Integer の Raw フォーマットである. SaveWavePCM モジュールの出力形式はヘッダ有り Integer の Wave フォーマットである. 保存したい信号の値によって，保存するための方法が異なる.

1. 実数信号 (時間波形) の保存

時間波形を保存する場合は，図 9.1 の様に接続する. SaveRawPCM , SaveWavePCM の入力に直接接続すれば良い. MultiFFT などのスペクトル変換を行なうモジュールを接続していなければ，ADVANCE パラメータは入力オブジェクト (INPUT) の次元数と同じ値を設定する.

2. 複素信号 (スペクトル) の保存

スペクトルを保存する場合は，図 9.2 の様に接続する. まず，時間波形に再合成する必要があるため，Synthesize モジュールを接続し，スペクトルから時間波形に変換する. その後は，実数値信号の保存方法と同様に，SaveRawPCM , SaveWavePCM を接続すれば良い. この時，ADVANCE パラメータは，Synthesize モジュールの ADVANCE パラメータと同じ値に設定する.

保存が成功したかどうかは，実行時に分離音ファイルが生成されたかどうかで，判定できる.

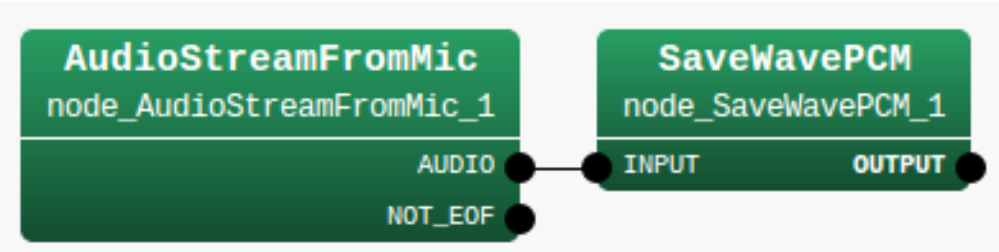


図 9.1: 接続例 1

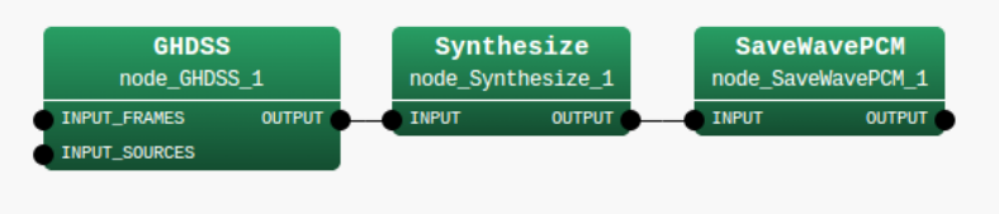


図 9.2: 接続例 2

Discussion

SaveRawPCM モジュールと SaveWavePCM モジュールの違いは，ヘッダが付いているかいないかだけである. 一般的には，ヘッダの付いた wave 形式のファイルの方が他のソフトウェアなどでの扱いが簡単であるため，特に理由が無い場合は，SaveWavePCM モジュールを使用すると良い.

See Also

HARK ドキュメントの Synthesize , SaveRawPCM , SaveWavePCM モジュール.

9.3 音源分離のパラメータをチューニングしたい

Problem

音源分離の性能が悪い場合，その場でどうパラメータ調節できるか

Solution

音源分離のメインモジュールである GHDSS のパラメータ設定について記述する．

1) **既知の空間伝達関数に合わせた設定**

GHDSS は事前に計測をした（またはマイクロホン位置を使って計算した）空間伝達関数の情報を用いて分離を行うため，持っている伝達関数に応じた設定が必要となる．具体的には以下の設定値に相当する．

- LC_CONST = DIAG
  もし伝達関数が正確に測定できているときは FULL だが，ほとんどの場合は DIAG に設定する．
- TF_CONJ **の設定**
  TF_CONJ = DATABASE : 実測伝達関数モード
  TF_CONJ_FILENAME に伝達関数ファイルを指定
  TF_CONJ = CALC : 従来の配置情報から計算するモード
  MIC_FILENAME にマイクロホン配置を指定

2) **非線形拘束の曲率の指定**

GHDSS ではシグモイド関数の曲率（原点での傾き）に相当する係数 SS_SCAL が性能を決める．この曲率を大きくすることで，線形拘束により近い形となり，小さくすることで非線形性を上げることができる．あまりに小さくすることは，適応をなまらせてしまうことになるため，この値を上下して，状況に合わせた設定を行う．

3) **定常ノイズの対処**

- FIXED_NOISE = true
  で定常ノイズの仮定のもとでの分離

4) **分離行列の初期値**

INITW_FILENAME で指定したファイルを分離行列の初期値とすることができる．あらかじめ同じような状況で録音したファイルなどから初期値を作っておくことで，分離の収束の高速化が見込める．これを指定しない場合は，初期値は定位結果と分離用伝達関数ファイルから決められる．

5) **ステップサイズ**

ステップサイズには SS_METHOD と LC_METHOD の 2 種類がある（コスト誤差に対するものと，線形拘束のステップサイズ）．両方とも ADAPTIVE にすることで，よほど最適化された環境でなければ手動で設定するより性能はよくなる．また，SS_METHOD = LC_METHOD = 0 に設定し，INITW_FILENAME に適当な分離行列ファイルを指定することで，固定ビームフォーマーによる分離が可能となる．以下，各パラメータの値の詳細を述べる．

5-1) **ステップサイズ :** SS_METHOD

1. FIX

SS_METHOD パラメータの Value を FIX に設定する．すると, SS_MYU パラメータが出現するので, その Value に例えば, 0.001 といった値を代入する． この値を大きく設定すると, 分離行列の収束は早くなるが入力データに対する収束安定性や分離精度は低くなる． 逆に小さく設定すると, 分離行列の収束は遅いが, データに対する収束安定性や分離精度は高い.

2. ADAPTIVE

SS_METHOD パラメータの Value を ADAPTIVE に設定する．入力データに対して, 自動的に適切なステップサイズを計算するため, 収束も早く, その安定性や分離精度も高い.

3. LC_FIX

LC_METHOD に連動したステップサイズに設定される．この場合, SS_METHOD で設定するパラメータはない.

5-2) **ステップサイズ：** `LC_METHOD`

1. FIX

LC_METHOD パラメータの Value を FIX に設定すると, LC_MYU パラメータが出現する．説明は SS_METHOD と同じである.

2. ADAPTIVE

LC_METHOD パラメータの Value を ADAPTIVE に設定すれば, 自動的に最適な値を計算して設定する.

Discussion

上記 Solution の説明を参照

See Also

音源分離の具体的なアルゴリズムについては HARK 講習会資料の「 HARK の技術説明（音源分離）」参照.

9.4 マイクロホン配置だけから音源分離したい

Problem

マイクロホンアレーで GHDSS を用いた音源分離を行いたいが、インパルス応答の測定はしたくない．ただし，マイク同士の 3 次元座標は測定済.

Solution

次の 2 つのステップで, マイク座標のみから音源分離ができる.

1. マイクロホン座標を記したファイルの用意: まず，マイクロホン座標を記したファイルを用意する必要がある．これは，HARK ドキュメント の「8.6 マイクロホン位置・ノイズ源」を参考にして作成すればよい.

2. GHDSS でマイクロホン座標ファイルの指定: マイクロホン座標を記したファイル (ここでは，ファイル名を micpos としておく) を GHDSS モジュールのプロパティーで指定する．まず，GHDSS プロパティーにおいて，TF_CONJ パラメータの Value を CALC に設定する．その後，MIC_FILENAME パラメータの Value に micpos を記述する.

micpos に記述されたマイクロホン座標系の原点の設定を変更したい場合は，MIC_POS_SHIFT パラメータを変更する．原点をマイクロホン座標の重心に設定したい場合は，Value を SHIFT にする．ファイルに記述された座標をそのまま用いる場合は，FIX にしておけば良い.

詳しい説明は GHDSS モジュールの説明を参考にされたい.

Discussion

マイク同士の位置関係から伝達関数をシミュレートすることで、インパルス応答を測定せずに分離が可能になる．ただし、マイクは自由空間に置かれていると仮定して計算するので、マイクの位置以外の影響 (ロボットの頭に着けるなら頭部の形状) は無視される．インパルス応答を測定すればその影響も含めた伝達関数が求まるので，測定できるならしたほうがよい.

See Also

HARK ドキュメントの GHDSS の節

9.5　ファンノイズなどの定常ノイズのせいで音源分離がうまくいかない

Problem

ロボットのファンノイズなどの固定ノイズの影響を考慮して, 音源分離を行ないたい場合に対応する. ロボットのノイズ源座標を記したファイルが必要.

Solution

以下の 2 ステップ必要である.

1. ノイズ位置ファイルの作成： 基本的には, マイクロホン座標の指定方法 (マイクロホン位置の指定方法) と同様に記述すればよい. 詳しくは, hark-document の「8.6 マイクロホン位置・ノイズ源」を参考にされたい.

2. GHDSS でのノイズファイルの指定： まず, GHDSS のプロパティーウィンドウを開く. FIXED_NOISE パラメータを `false` から `true` に変更する. すると, FIXED_NOISE_FILENAME というパラメータが出現するので, そこに作成してあるノイズ源座標ファイル名を記入する. これで, 既知のノイズに反応することなく分離を行なうことができる.

成功したかどうかは, SaveRawPCM などの結果出力モジュールを接続し, 分離音を調べると良い. ノイズ源に関する分離音は出力されなくなっているはずである.

Discussion

なし.

See Also

GHDSS , マイクロホン位置の指定方法

9.6 分離音に入っている雑音を後処理で減らしたい

Problem

分離音に歪みが含まれている時, 音声強調によって認識性能を改善したい.

Solution

音声強調に関する以下のノード PostFilter , HRLE , WhiteNoiseAdder , MFMGeneration とその周辺の設定について記述する.

1) **PostFilter**

この処理に関しては，状況により，PostFilter が無いときのほうが認識性能が良いときがある．PostFilter のパラメータを環境に対して適切に設定する必要がある．デフォルトのパラメータは HARK 開発チームの環境で合わせたものであるから, ユーザの環境で上手く動作する保証はない.

PostFilter のパラメータ数は非常に多く, 相互依存しているものも多い. そのため, 手動でのチューニングは非常に困難である.

解決方法の一つは, 組み合わせ最適化手法を用いることである．データセットを準備できるなら, 評価値として認識率や SNR を用いることで, Generic Algorithm や Evolutional Strategy などの最適化手法を適用すればよい. 用いるデータによっては, 環境に特化しすぎたパラメータを学習する可能性があるので, 注意が必要である.

PostFilter では，基本的に入力信号のパワーの大小関係を利用して，定常ノイズ，残響，漏れのノイズを動的に推定し，それを減じることで，高精度な分離音を引き出すことを行う．状況により性能が下がるのは，その減算をした時の音声の歪みが認識に悪影響を及ぼすためである．よって，定常ノイズ，漏れ，残響の 3 種類の推定からの減算の影響を小さくしたり大きくしたりすることで，PostFilter の影響の大小を変化させることができるため，状況次第でパフォーマンスがあがることがある．PostFilter の影響を極力小さくするには以下のパラメータを 0 に指定すれば実現できる.

- 漏れ ： `LEAK_FACTOR = 0`
- 残響 ： `REVERB_LEVEL = 0`
- 定常ノイズ ： `LEAK_FACTOR = 0`

逆に PostFilter の影響を大きくしたい場合はこのパラメータを 1 に近づければ良い.

2) **HRLE**

PostFilter に比べ，HRLE のパラメータ数は非常に少ない．HRLE は入力分離音のパワーからヒストグラムを算出し，音声と雑音のパワー差を利用して音声強調をするモジュールである．従って，ヒストグラムの設計が性能に大きく影響する．HRLE には `LX`, `TIME_CONSTANT`, `NUM_BIN`, `MIN_LEVEL`, `STEP_LEVEL`, の 5 つのパラメータがあるが，`LX` 以外のパラメータはヒストグラムの細かさや考慮する時間幅を考えており，前もって十分な値に設定されている．ただし，`LX` のみは，雑音と音声を切り分けるレベルを指定するもので，環境に依存して大きく変化する．`LX` を大きくすると，雑音とみなすレベルが大きくなり，雑音は抑圧されやすくなるものの，パワースペクトルを引きすぎてしまい，歪みが大きくなる可能性がある．一方，小さな `LX` は，雑音を抑圧できない可能性がある．環境によって適切な `LX` を設定する必要がある.

3) **WhiteNoiseAdder**

WN_LEVEL の値を調節する．小さすぎると分離音で生成された歪みが十分に緩和できない．大きすぎると歪み部分だけでなく，分離音自体が影響を受けてしまう．

4) **MFMGeneration**

THRESHOLD の値 0 から 1 の範囲で値を変化させることで，特徴量をマスクする閾値を変化させることができる．1 に近づくほど全ての特徴をマスクせずに 0 として扱ってしまうし，0 に近づくほと全ての特徴をマスクして 1 として扱ってしまうので認識率が下がる．

Discussion

なし．

See Also

「うまく分離できていないけどどうすればいいの？」

9.7 音源やロボットが移動する状況で分離したい

Problem

音源やロボットが移動すると、分離に追従性が必要になる。そのまま使用すると、移動速度によっては分離性能が低下してしまう。どのように分離性能を保持できるか.

Solution

GHDSS プロパティにある UPDATE_METHOD_TF_CONJ と UPDATE_METHOD_W を適切に設定することで解決できる.

よくわからない場合は，デフォルト値を使用しておけば良い.

変更する場合は，音源自体 (ID) に着目するか，方向 (POS) のみに着目するか，の方針を定める必要がある.特に，音源が移動したり，ロボット本体が移動した時など，分離の追従性が必要とされる場面で，分離結果が若干変ってくる.

ID に着目する設定にした場合，同じ ID が付与されたときに，幾何的拘束や分離行列の値として前ステップのものを再利用するので，音源が高速に移動するとそれらの値が条件として不適切で，分離性能が低下してしまう．逆に，あまり音源が移動しない場合は再利用性により，分離精度が向上し続ける.

POS に着目する設定にした場合は ID の場合と逆になり，音源が高速で移動する場合に有利といえる．これは，移動間隔が広いと，拘束条件や分離行列の値をデータベースのものを用いたり，初期化したりするためである.

理想的はこれらを動的に音源毎に適用することである．詳しくは，GHDSS モジュールの説明を参照されたい.

Discussion

なし.

See Also

GHDSS

第10章　特徴量抽出

10.1　はじめに

Problem

音声認識に用いられる特徴量にはどのようなものがあるのかを知りたい．そのうちで，HARK がサポートしている特徴量には何があるのかも知りたい．

Solution

一般的な音声認識に用いられる特徴量には，以下のようなものがある．

1. LPC(Linear Predictive Coding: 線形予測) 係数
2. PARCOR(PARcial CORelated: 偏自己相関) 係数
3. MFCC(Mel-Frequency Cepstrum Coefficient)
4. MSLS(Mel-Scale Log Spectrum)

このうち、HARK は 3, 4 番目の 2 種類をサポートしている。web 上で公開されている音響モデルを用いて音声認識を行いたい場合は MFCC を使うと良い．ミッシングフィーチャー理論と組み合わせて性能向上を図りたい場合は MSLS を使う方が良い．

Discussion

LPC 係数はスペクトル包絡のモデルのパラメータであり，定常過程 x_t の時刻 t における値が近い過去のサンプルと相関をもつことに基づいている．図 10.1 に LPC 係数の求め方を示す．過去の M 個の入力信号の値から予測した値（$\hat{x}_t$）と実際の入力信号の値 x_t の二乗平均誤差が最小となるように求めた予測係数（a_m）が LPC 係数である．この LPC では比較的正確な音声のモデルが得られるため，音声の分析合成に広く利用されてきた．しかし，LPC に基づくモデルは係数感度が高く，わずかな係数の誤差によって不安定になることがある．そこでこの問題に対処するため，音声の分析合成は PARCOR の形で行われる．

PARCOR 係数は，$x_{t-(m-1)}, \ldots, x_{t-1}$ から予測した x_t（前方予測）と x_{t-m}（後方予測）の予測誤差の相関係数である．図 10.2 は PARCOR 係数を導出する様子を表す．この PARCOR に基づくモデルは原則的に安定である [1].

MFCC はケプストラムパラメータの 1 つであり，メル周波数軸上で等間隔に配置されるたフィルタバンクを用いて導出される特徴量であり [1]，図 10.3 はその導出処理を示す．

MSLS は MFCC 同様にフィルタバンク分析を用いて導出されるが，MFCC が時間領域の特徴量であるのに対し， MSLS は周波数領域の特徴量である．特定の周波数を持つノイズが音響信号に混入した場合，MSLS はその周波数を含む特定の特徴量が影響を受ける．一方 MFCC の場合はノイズの影響が広がってしまい，複

数の特徴量が影響をうける．そのため，一般にミッシングフィーチャー理論と組み合わせて音声認識を行う場合は MSLS の方が良い性能を示す．

1 今井　聖，音声信号処理，森北出版株式会社，1996.
2 鹿野　清宏他，IT Text 音声認識システム，株式会社オーム社，2001.

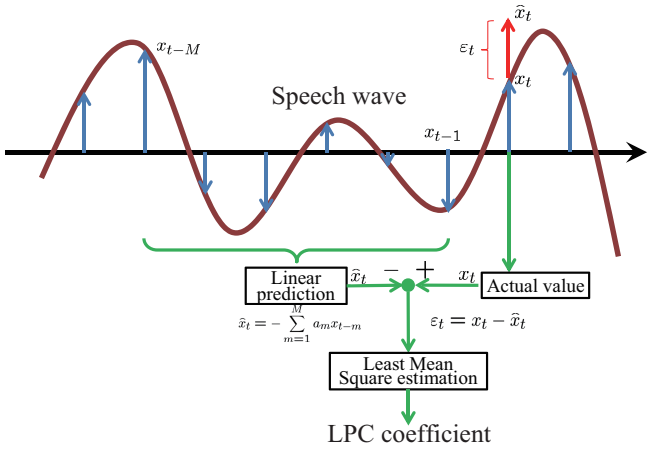


図 10.1: LPC coefficients

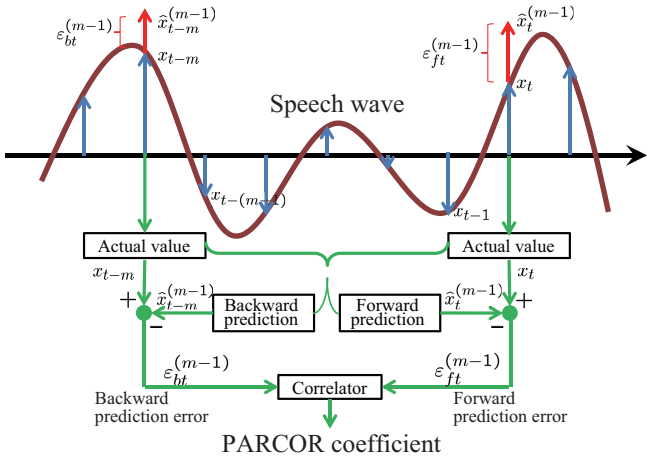


図 10.2: PARCOR coefficients

See Also

HARK ドキュメントの MFCCExtraction，MSLSExtraction の節

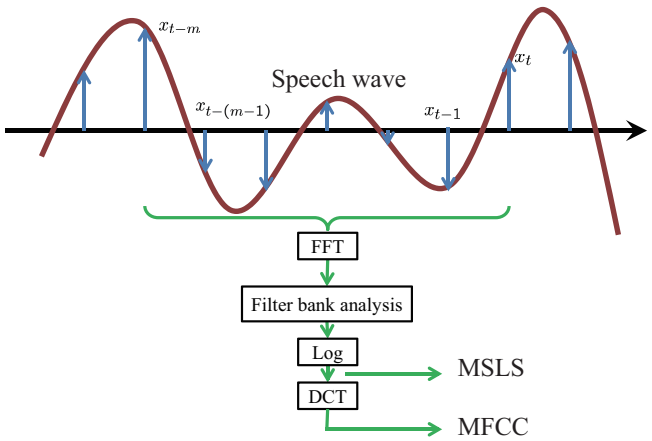


図 10.3: MFCC and MSLS

10.2 ミッシングフィーチャマスク (MFM) の閾値の設定の仕方がわからない

Problem

MFMGeneration モジュールのパラメータをどのように設定したら良いか分からないときに読む.

Solution

MFMGeneration には THRESHOLD というパラメータがあり，このパラメータが音声認識の性能を左右する．閾値を 0.0 に設定すると，ミッシングフィーチャー理論を使わない音声認識を行う．一方で閾値を 1.0 に設定すると，すべての特徴量にマスクをかけるため，まったく特徴量を使わないで認識を行う．

適した値を求めるには，例えば 0.1 刻みで変化させるなどしながら実際に音声認識を行い，実験的に求めると良い.

Discussion

MFMGeneration は次の式で表され，信頼度を THRESHOLD でしきい値処理し，0.0(信頼しない) または 1.0(信頼する) の 2 値をとるマスク (ハードマスク) を生成する.

$$m(f,p) = \begin{cases} 1.0, & r(p) > THRESHOLD \\ 0.0, & r(p) \leq THRESHOLD \end{cases}$$

ただし，$m(f,p)$ は f フレームの p 次元目の特徴量に対するマスクの値を表し，$r(p)$ は p 次元目の特徴量の信頼度を表す.

See Also

HARK ドキュメントの MFMGeneration の節

10.3　特徴量をファイルに保存したい

Problem

HARK を使って抽出した特徴量を保存したいときに読む.

Solution

特徴量を保存するためには，SaveFeatures モジュールもしくは SaveHTKFeatures を使う．SaveFeatures モジュールは特徴量を `float` バイナリ形式で保存するためのモジュールであり，SaveHTKFeatures モジュールは特徴量を HTK 形式で保存するためのモジュールである．

図 10.4 に特徴量を保存するためのネットワーク例を示す．ここでは，AudioStreamFromWave モジュールから読み込まれた 1 ch 音声信号から特徴量抽出を行い保存する．この図 10.4 の様に抽出した特徴量を SaveFeatures モジュールの入力とすることで保存することができる．

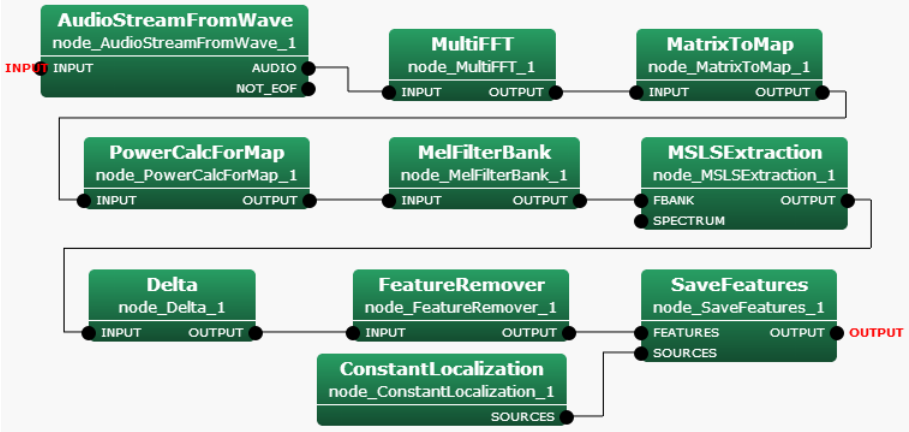


図 10.4: Sample netwok to save feature

Discussion

SaveFeatures モジュールは特徴量を 32 bit 浮動小数点型，リトルエンディアンで保存する．SaveHTKFeatures モジュールは特徴量を HTK 形式で保存する．どちらを使用するかは，ファイルの利用目的に依存する．例えば HTK を用いて音響モデルの学習を行いたい場合は，HTK 形式の方が便利である．

See Also

保存するファイル形式については，HARK ドキュメントの SaveFeatures，SaveHTKFeatures モジュールの説明を読むと良い．ファイル形式については，同じく HARK ドキュメントの `float` バイナリ節を見ると良い．

第11章　音声認識

11.1　設定ファイル (.jconf ファイル) を作りたい

Problem

HARK を用いて音声認識をする時の Julius の設定ファイル（.jconf ファイル）で指定するべきオプションが知りたい.

Solution

Julius には，多くの設定可能なオプションパラメータがある．Julius にコマンドラインで毎回オプション指定するのは，繁雑である．一連のオプションパラメータをテキストファイルに保存し，そのファイルを Julius に与えることでオプションの入力を簡略化可能である．このテキストファイルは .jconf ファイルと呼ばれる．

Julius で用いるオプションは，http://julius.sourceforge.jp/juliusbook/ja/desc_option.html にまとめられている．全てのオプションは， .jconf ファイルにテキスト記述可能である．

SpeechRecognitionClient（または　SpeechRecognitionSMNClient ）と接続する場合の注意点をまとめる．最低限必要な設定項目は,

- -notypecheck
- -plugindir /usr/local/lib/julius_plugin
- -input mfcnet
- -gprune add_mask_to_safe
- -gram grammar
- -h hmmdefs
- -hlist allTriphones

である.

-notypecheck は，必須オプションである．HARK では拡張した音響パラメータ構造を取ることから Julius デフォルトのタイプチェックでサポートされない．従って -notypecheck を付けることが必須である．このオプションを省略すると，Julius は，特徴量のタイプチェックでタイプエラーを検出して，認識しない.

-plugindir は，mfcnet などの拡張機能プラグインファイルが保存されているパスを指定する． このパスは, -input mfcnet や，-gprune add_mask_to_safe よりも先に指定されなければならない．プラグインのパス下にある拡張プラグインファイルは，全て読み込まれる．尚、Windows 版においては、プラグインを使用しないためプラグインディレクトリ名は任意となるが、本オプションは必須となる.

-input mfcnet は，SpeechRecognitionClient (または，SpeechRecognitionSMNClient) から受信する特徴量を認識するためのオプションである．ミッシング・フィーチャ・マスクをサポートするためには，このオプションを指定する．

-gprune サポートした時の，Gaussianpruning アルゴリズムを選択する．ミッシング・フィーチャ・マスクをサポートした計算になるため，オプション名が通常の Julius と異なる．
次の選択肢 $\{add\_mask\_to\_safe \| add\_mask\_to\_heu \| add\_mask\_to\_beam \| add\_mask\_to\_none\}$ から選ぶ．それぞれ通常の Julius の $\{safe \| heuristic \| beam \| none\}$ に対応する．

その他，通常の音声認識と同様に言語モデルの指定，音響モデルの指定が必要である．-gram で文法ファイルを，-h で HMM 定義ファイルを， -hlist で HMMList ファイルを指定する．

第12章　その他

12.1　窓長とかシフト長の適切な値が知りたい

Problem

最適な分析窓長とシフト長が知りたい．

Solution

Length は音声の分析窓長である．一般に，20-40 [ms] に相当する長さを指定すればよい．サンプリング周波数 fs Hz ならば，length = fs/1000 * x で求まる．x は 20-40[ms] である．Advance は，分析フレームシフト長である．一般に，後続フレームとフレーム全体の 1/3 - 1/2 重なる量シフトを指定する．

音声認識する場合，音響モデル作成にも同一の Length, Advance を使う必要がある．

Discussion

音声を扱う場合，信号が弱定常状態と仮定できる範囲が 20-40 [ms] であるからこの方針で設定する．シフト長は，窓の実行幅で決まる．具体的には，窓関数の持つエネルギーと等価なエネルギーを持つ，矩形窓の窓長を求める．この窓長は，連続するフレームを分析した時に，同一サンプルを重複してフレーム処理せず，かつサンプルの取り零しのないフレーム処理が可能である．一般に知られる音声分析用の窓関数のエネルギーは，矩形窓長で約 1/3 - 1/2 のエネルギーと同一になることから，フレームのシフト量を，この範囲で使う．1/3 が保守的な設定で，同一サンプルを重複してフレーム処理する可能性はあるが，サンプルの取り零しがない．1/2 では，使用する窓関数によっては，サンプルの取り零しが起る可能性があるが，重複してフレーム処理することはない．ただし，分析に矩形窓を使用するならば，シフト長は分析フレーム長にする．三角窓の時，フレームシフト量は 1/2 である．

12.2 MultiFFT に使う窓関数はどれを使えばよいか知りたい

Problem

MultiFFT の窓の選び方が知りたい.

Solution

（HUMMING，CONJ，RECTANGLE の 3 種類）
音声分析ならば HUMMING を選べばよい．他の信号の場合，その信号をスペクトル分析するときに用いる窓を適切に選択する.

12.3 PreEmphasis の使い方は？

Problem

PreEmphasis を時間領域と周波数領域のどちらで使えば良いか分からないときにこのレシピを読む.

Solution

プリエンファシスの一般的な音声認識における必要性や効果に関しては，さまざまな書籍や論文で述べられているので，それらを参考にして欲しい.

PreEmphasis は時間領域でも周波数領域でもかけることができる．ただし，音響モデル学習で用いるデータに合わせた方が良い.

第13章　進んだ使い方

13.1　ノードを作りたい

Problem

HARK において自分でノードを作成したい.

Solution

自分で新しいノードを作成する場合，パッケージではなく，ソースコンパイルする必要がある．ソースコンパイルからインストールする方法については，HARK 講習会資料の「 HARK のインストール」の項を参照.

準備ができたら，作成したいノードのソースを記述する．基本的なノードの作り方については HARK 講習会資料の「ノードの作成」に以下の事項が記載されている.

- cc ファイル（ソースファイル）の基本的な書式
- 例 (ChannelSelector) を用いた説明
- パラメータの追加
- `Makefile.am` の書き換え方法

本節では，さらに以下の項目

- 入力の追加
- 出力の追加
- バッファ（ `Lookback Lookforward` ）
- 各変数型の入出力
- 入力本数をダイナミックに切り替える

について `PublisherInt.cc`，`SubscriberInt.cc` などの具体的なノードを作ることで，説明を行う.

PublisherInt.cc の作成

まずは，整数をパラメータとして読み込んで，そのまま吐き出す `PublisherInt.cc` を作ってみる.（注：`hark_test` ディレクトリをパッケージとしている場合を想定している.）

`{$PACKAGE}/hark_test/src/PublisherInt.cc`

に次のソースコードをカット＆ペースト.

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class PublisherInt;

DECLARE_NODE(PublisherInt);
/*Node
 *
 * @name PublisherInt
 * @category HARK_TEST
 * @description This block outputs the same integer as PARAM1.
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description This output the same integer as PARAM1.
 *
 * @parameter_name PARAM1
 * @parameter_type int
 * @parameter_value 123
 * @parameter_description Setting for OUTPUT1
 *
END*/

class PublisherInt : public BufferedNode {
  int output1ID;
  int output1;
  int param1;

public:
  PublisherInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    output1 = 0;
    param1 = dereference_cast<int>(parameters.get("PARAM1"));
    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    output1 = param1;
    cout << "Published : [" << count << " , " << output1 << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(output1));

    // Main loop routine ends here.

  }
};
```

以下，部分ごとにソースコードを説明する．

```
#include <iostream>
```

```
#include <BufferedNode.h>
```

```
#include <Buffer.h>
```

標準出力のライブラリと，FlowDesigner 用のライブラリ．ノードを作る時は必ず include する．

```
using namespace std;
```

```
using namespace FD;
```

名前空間の宣言．HARK の大元である FlowDesigner のクラスは全て FD の名前空間の中で定義されているので，略記したい時は必ず宣言しておく．

```
class PublisherInt;
```

このノードのクラス名．下記で設定するノード名と一緒にする．

```
DECLARE_NODE(PublisherInt);
/*Node
 *
 * @name PublisherInt
 * @category HARK_TEST
 * @description This block outputs the same integer as PARAM1.
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description This output the same integer as PARAM1.
 *
 * @parameter_name PARAM1
 * @parameter_type int
 * @parameter_value 123
 * @parameter_description Setting for OUTPUT1
 *
END*/
```

DECLARE_NODE において，PublisherInt クラスを一つのノードとして定義されるよう宣言する（よって，クラス名と同じにしないとエラー）．その下のコメントアウトに見える @name などが，FlowDesigner の GUI 上での，宣言されたノードの設定になるので，コメントとは思わず，必ず設定する．この設定値だが,「ノード本体の設定，ノード入力の設定，ノード出力の設定，ノード内部パラメータの設定」の 4 つがそれぞれ設定可能．ノード本体の設定以外は複数個設けることができる（複数設ける場合の設定方法については後述）．以下が具体的な設定値である．

- ノード本体の設定
  - @name : FlowDesigner 上で表示されるノード名（クラス名と同じに）
  - @category : FlowDesigner の GUI で右クリックした時に，そのノードが属するカテゴリの設定．
  - @description : ノードの説明 (FlowDesigner でそのノードをマウスオン時に表示．無くても可)
- ノードの入力の設定
  - @input_name : ノードに表示される入力の名前
  - @input_type : 入力変数の型
  - @input_description : 入力変数の説明（省略可）
- ノードの出力の設定
  - @output_name : ノードに表示される出力の名前
  - @output_type : 出力変数の型
  - @output_description : 出力変数の説明（省略可）
- ノードの内部パラメータの設定
  - @parameter_name : ノードに表示される変数名（マウスオン時に黄色い窓で表示）
  - @parameter_type : パラメータの変数型

- @parameter_value : パラメータの初期値（ソース内で変更可）
- @parameter_description : （パラメータの説明．省略可）

今回のソースでは，一つの出力と，一つの内部パラメータを持っているので，FlowDesigner で表示すると次のようになる．

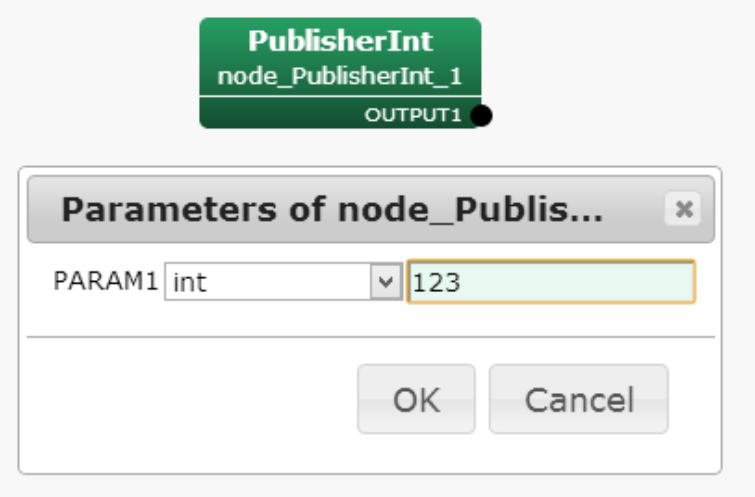


図 13.1: PublisherInt ノード

```
class PublisherInt : public BufferedNode {
  int output1ID;
  int output1;
  int param1;
```

BufferedNode クラスを継承した PublisherInt クラス定義する．(BufferedNode クラスは /flow_designer_hri/data-flow/ の中に定義されている．) outputID は出力ポートの ID を格納する整数であり，この ID をもとに，出力ポートに渡すべきポインタの対応を取っている．

```
public:
  PublisherInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    output1 = 0;
    param1 = dereference_cast<int>(parameters.get("PARAM1"));
    inOrder = true;
  }
```

BufferedNode クラスを継承したコンストラクタ．引数として，nodeName （ FlowDesigner の N ファイルにおけるクラスオブジェクト名），params （ Node クラスのメンバ変数 parameters の初期化子で，内部パラメータが個数分定義された集合）をとる．

output1ID   = addOutput("OUTPUT1"); が，FlowDesigner の GUI 側で設定した OUTPUT1 の ID を，クラス側で定義した output1ID に格納する行になる．

param1 = dereference_cast<int>(parameters.get("PARAM1")); が， FlowDesigner の GUI 側で設定した内部パラメータを， int 型にキャストして呼び出しているところである．@parameter_type の型だが，

int 型，float 型，bool 型，string 型があり，それ以外は Object として呼び出す.（string 型は Object として呼び出し， string にキャスト.）例を示す.

- int 型 (int param;)
  param = dereference_cast<int>(parameters.get("PARAM"))
- float 型 (float param;)
  param = dereference_cast<float>(parameters.get("PARAM"))
- bool 型 (bool param;)
  param = dereference_cast<bool>(parameters.get("PARAM"))
- string 型 (string param;)
  param = object_cast<String>(parameters.get("PARAM"));
- Vector 型 (Vector<int> param;)
  param = object_cast<Vector<int> >(parameters.get("PARAM"));

String が std::string，Vector が std::vector でないのは，この型が FlowDesigner の入出力用の特殊な型のため．この型にして，情報をやりとりしないとエラーとなる．

inOrder = true; を設定すると，calculate が呼び出される度に count 値が 1 ずつ増加する（詳細は後述）．

ソースの解説に戻る．

```
void calculate(int output_id, int count, Buffer &out)
{

  // Main loop routine starts here.

  output1 = param1;
  cout << "Published : [" << count << " , " << output1 << "]" << endl;
  (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(output1));

  // Main loop routine ends here.

}
```

ここがノードのメインルーチンであり，この中身が count 毎に繰り返し演算が行われる．引数である count は，ループ回数.

今回のノードでは PARAM1 の値をそのまま次へ渡すだけなので，現在のループの情報しか必要ないが，複数のループにまたがって，平均値を取ったり，速度を計算したりしたい時は，その分だけバッファを取る方法がある．これについては，後ろの章で解説する．

(*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(output1)); がノードから出力される値.（count 回目の ”output1ID” という ID で指定されるポートの出力が規定される.）このノードは出力が一つなので，(out[count] = ObjectRef(Int::alloc(output1)); と出力を書くこともできるが，一般型では上の書き方となる.（一出力の場合， *(outputs[output1ID].buffer) と out は等価となる.）

また， int 型であった output1 が，Int 型にキャストされていることに着目する．このように，全ての入出力に関する変数型は，Int 型，Float 型，String 型，Bool 型，Vector 型，Matrix 型，Map 型という FlowDesigner 独自の型にする必要がある．例を示す.

- int 型
  (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(output1));
- float 型
  (*(outputs[output1ID].buffer))[count] = ObjectRef(Float::alloc(output1));
- bool 型
  (*(outputs[output1ID].buffer))[count] = TrueObject;
- string 型
  (*(outputs[output1ID].buffer))[count] = ObjectRef(new String(output1));
- Vector 型
  RCPtr<Vector<float> > output1(new Vector<float>(rows));
  (*(outputs[output1ID].buffer))[count] = output1;
  (rows はベクトルの要素数．Vector<int> 等も定義可．Vector.h の include が必要)
- Matrix 型
  RCPtr<Matrix<float> > output1(new Matrix<float>(rows, cols));
  (*(outputs[output1ID].buffer))[count] = output1;
  (rows, cols は行列数．Matrix<int> 等も定義可．Matrix.h の include が必要)

ここで，RCPtr とは FlowDesigner 用のオブジェクトスマートポインタのこと．Matrix や Vector などの配列の入出力では，このポインタを渡す．

PublisherInt.cc をインストール

PublisherInt.cc を FlowDesigner で使えるようにソースコンパイルしてインストールする．まずは，
{$PACKAGE}/hark_test/src/Makefile.am
の中の lib****_la_SOURCES 変数（****は任意のパッケージ．今回は hark_test ）の適当な位置に

```
PublisherInt.cc \
```

を追加する．“ \ ” を忘れないようにする．

```
> cd {$PACKAGE}/hark_test/
```

で

```
> autoreconf; ./configure --prefix=${install dir}; make; make install;
```

とし，インストールする．(${install dir} は自分の設定にしたがう．/usr などが相当する．)
FlowDesigner を起動．

```
> flowdesigner
```

GUI が起動したら，

```
右クリック > HARK_TEST > PublishInt
```

で作ったノードがあることを確認．これでインストールは終了．
ここで表示されない場合のいくつかのトラブルシュートを挙げておく．

- ./configure --prefix=/*** で，指定したディレクトリと，flow_designer_hri をインストールしたディレクトリが同じか確認する．FlowDesigner は，自分のインストールされているディレクトリの def ファイルを読みにいくようになっているので，違うディレクトリにインストールした場合は無視される．

- {$PACKAGE}/hark_test/src/Makefile の中に，自分の作ったノードをコンパイルするスクリプトが正しく作られているか確認する．autoreconf が正しく行われているか．Makefile.am は正しく書き換えたか．
- cc ファイルのソース中のクラス名とノード名が同じか確認．同じでない場合，コンパイルはできても GUI で表示されない場合がある．
- パスの設定は正しいか確認 (which flowdesiner)．過去に root 権限で /usr/bin などに FlowDesigner と HARK をインストールをしたことがあり，今回はユーザー権限で自分のローカルにインストールした場合に特にこの問題に遭遇する．

SubscriberInt.cc の作成

PublisherInt.cc から出力された整数を入力し，そのまま吐き出す SubscriberInt.cc を作る．
{$PACKAGE}/hark_test/src/SubscriberInt.cc
に次のソースコードをカット＆ペースト．

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class SubscriberInt;

DECLARE_NODE(SubscriberInt);
/*Node
 *
 * @name SubscriberInt
 * @category HARK_TEST
 * @description This block inputs an integer and outputs the same number with print.
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description Same as input
 *
END*/

class SubscriberInt : public BufferedNode {
  int input1ID;
  int output1ID;
  int input1;

public:
  SubscriberInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    input1ID     = addInput("INPUT1");
    output1ID    = addOutput("OUTPUT1");
    input1 = 0;
    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    ObjectRef inputtmp = getInput(input1ID, count);
    input1 = dereference_cast<int> (inputtmp);

    cout << "Subscribed : [" << count << " , " << input1 << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(input1));

    // Main loop routine ends here.

  }
};
```

PublisherInt.cc と似ているが，違う点を見ていく．

```
  * @input_name INPUT1
  * @input_type int
  * @input_description input for an integer
```

PublisherInt.cc には入力ポートは存在しなかったが，入力を設ける場合は，まず FlowDesigner の GUI の設定をここで行う．@output と基本的に書式は変わらない．

SubscriberInt.cc はパラメータを持たない代わりに一つの入力を持つので，FlowDesigner では以下のように表示される．

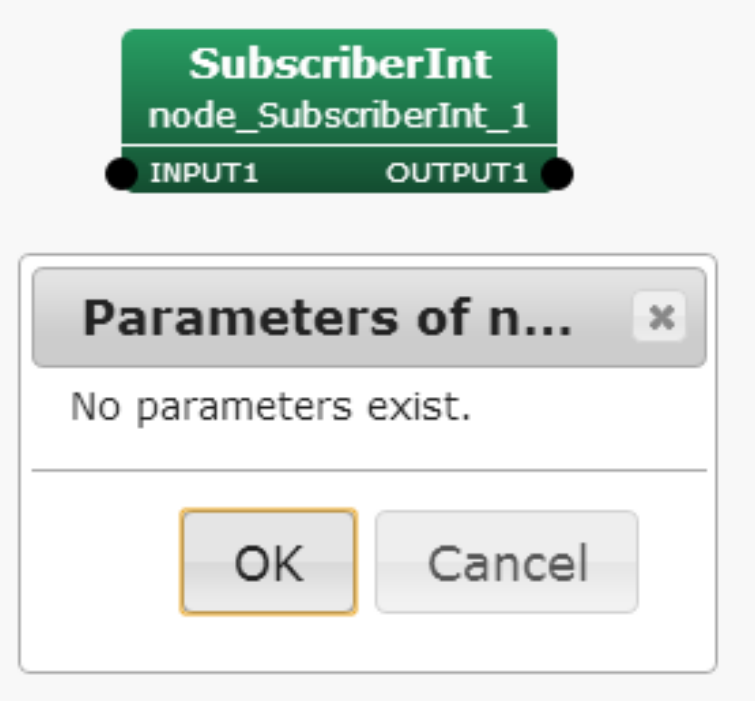


図 13.2: SubscriberInt ノード

```
input1ID     = addInput("INPUT1");
```

FlowDesigner の GUI 側で設定した INPUT1 の ID を，クラス側で定義した input1ID に登録し，GUI のポートと読み込む入力データの対応づけを行う．

```
ObjectRef inputtmp = getInput(input1ID, count);
input1 = dereference_cast<int> (inputtmp);
```

ID に対応した入力ポートから，データを FlowDesigner の独自の型として受け取り，それを int 型にキャストしている．出力側で FlowDesigner の独自の型である ObjectRef （Int 型，Float 型，Bool 型，String 型，etc...）を渡したように，受け取る側でもこのような手順でキャストしなおす．以下，他の型での例を挙げる．

- int 型

  ```
  ObjectRef inputtmp = getInput(input1ID, count);
  input1 = dereference_cast<int> (inputtmp);
  ```

- float 型

  ```
  ObjectRef inputtmp = getInput(input1ID, count);
  input1 = dereference_cast<float> (inputtmp);
  ```

- bool 型

  ```
  ObjectRef inputtmp = getInput(input1ID, count);
  input1 = dereference_cast<bool> (inputtmp);
  ```

- string 型

  ```
  ObjectRef inputtmp = getInput(input1ID, count);
  const String &input1 = object_cast<String> (inputtmp);
  ```

  (これで input1 は string 型)

- Vector 型

  ```
  RCPtr<Vector<float> > input1 = getInput(input1ID, count);
  ```

  （使用時，(*input1)[i]．Vector<int> 型も同様.）

- Matrix 型

  ```
  RCPtr<Matrix<float> > input1 = getInput(input1ID, count);
  ```

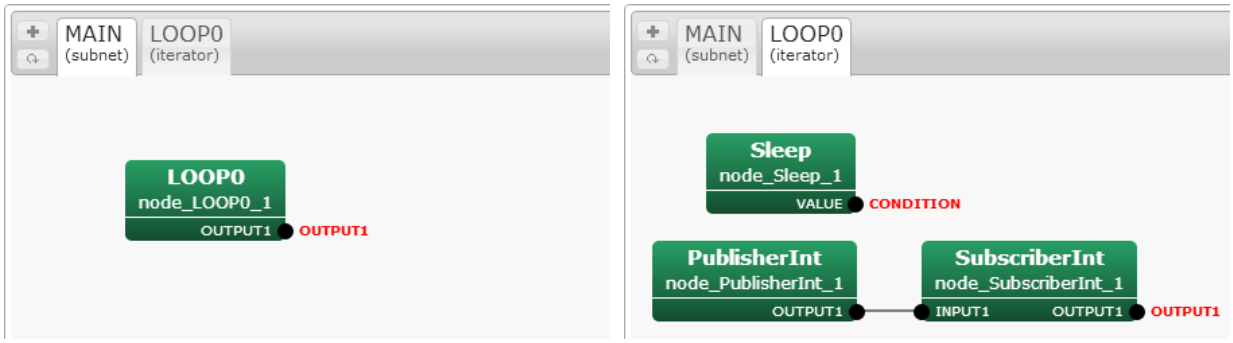
  （使用時，(*input1)(i,j)．Matrix<int> 型も同様）

出力部分については PublisherInt.cc と同じ．作り終えたら，前章と同様の手順で，ソースコンパイルからインストール．FlowDesigner を起動し，SubscriberInt.cc が正しくインストールされていることを確認されたい．

PublisherInt.cc と SubscriberInt.cc を使って N ファイルを作成

最後に作成した PublisherInt.cc と SubscriberInt.cc を使って，FlowDesigner のネットワークファイル (N ファイル) を作ってみる．

完成させたいネットワークファイルは以下の通り．この図がわかれば，あとは不要なので次章へ進んでもらいたい．



(a) MAIN(subnet) シート　　(b) LOOP0(iterator) シート

図 13.3: PublisherInt + SubscriberInt　ネットワークファイル

まずは FlowDesigner を起動．新規ファイルの場合，起動時には MAIN というシートのみがあると思われる．文字通りこれがプログラムのメインである．

```
Networks > Add Iterator
```

でループ処理シートを加える．(シート名は適当に決める．ここでは LOOP0 とする.)

まず MAIN シート側で，

```
右クリック > New Node > Subnet > LOOP0
```

でメイン側から LOOP0 の処理が実行できるようにする．次に LOOP0 シートに移動し，繰り返し処理のサンプリング周期を定める．(イベントベースでも，時間ベースでも可能.）ここでは， Sleep を使って，時間周期的に繰り返し演算を行わせてみる．

右クリック > New Node > Flow > Sleep で Sleep ノードの配置
Sleep を左クリック > パラメータ SECONDS を適当に設定（例：10000） > OK
Sleep の出力端子を "Ctrl" を押しながらクリック

Sleep の出力端子が CONDITION になったことを確認．これがループ処理のトリガとなっており，Sleep の処理が終了した時に新しいループが始まることを意味している．次にメインの処理を記述する．

右クリック > New Node > HARK_TEST > PublisherInt
右クリック > New Node > HARK_TEST > SubscriberInt

で LOOP0 シートに 2 つのノードを配置する．

PublishInt を左クリック > PublisherInt のパラメータ PARAM1 を適当に設定 (例：123) > OK
PublisherInt の出力端子を SubscriberInt の入力に接続
SubscriberInt の出力端子を "Shift" を押しながらクリック

SubscriberInt の出力端子が OUTPUT1 になったことを確認．これがループ処理の出力になる．この OUTPUT1 を出力に加えたことによって，MAIN シートの LOOP0 ブロックに出力ポートが一つできる．MAIN シートに戻り，以下のようにする．

LOOP0 の出力端子を "Shift" を押しながらクリック

すると OUTPUT1 が MAIN シートからも出力されるようになり，正しく演算をすることができる．

適当な名前をつけて保存
「実行」を押す

以下のような出力がコンソールに現れる．

```
Published : [0 , 123]
Subscribed : [0 , 123]
Published : [1 , 123]
Subscribed : [1 , 123]
Published : [2 , 123]
Subscribed : [2 , 123]
...
```

これで，整数をやりとりできるネットワークファイルが完成した．ここまでが基本的なノードの作り方とノードの作り方である．ここからは，入出力の追加や，複数のフレーム間の処理など，ノードを作っていく上で重要な応用について触れていく．

ノードに内部パラメータを追加する

PublisherInt.cc では，PARAM1 という内部パラメータが一つ存在していた．ここではパラメータを複数設けるように PublisherInt.cc を改変してみる．int 型ではつまらないので，比較的難しい Vector 型をパラメータとして読み込む方法を紹介する．(名前は PARAM2 とする.）

目標は Vector 型変数をパラメータとして読み込んで，それを以下のように PARAM1 倍して，その結果を出力ポートから出すようにノードを改変することである．

PARAM1 * PARAM2

これには Vector 型の内部パラメータの追加と， Vector 型の出力ポートの追加の両者が要素として含まれるので，二つの章に分割して考える．

本章では内部パラメータの追加の仕方を説明する．

{$PACKAGE}/hark_test/src/PublisherInt.cc

を開く．Vector 型変数を扱うので，まずは Vector.h を include する．

```
#include <Vector.h>
```

FlowDesigner の GUI の設定を改変する．

/*Node ... END*/

までの中に以下を追記する．

```
 *
 * @parameter_name PARAM2
 * @parameter_type Vector<int>
 * @parameter_value <Vector<int> 0 1 2>
 * @parameter_description OUTPUT2 = PARAM1 * PARAM2
 *
```

ここで @parameter_value に注目されたい．この <Vector<int> 0 1 2> の書式が厳格であり，特にスペースなどを必ず入れることには注意されたい．

次にクラスのメンバ変数に以下を追加する．

```
Vector<int> param2;
```

コンストラクタに以下を追記．

```
param2 = object_cast<Vector<int> >(parameters.get("PARAM2"));
```

あとは，次の例のように Vector として使用することができる．

```
for(int i = 0; i < param2.size(); i++){
  cout << param2[i] << " ";
}
```

本節では文章のみで改変部分を説明したが，最終的なソースは次節の最後に掲載されている．

ノードに出力を追加する

前節では Vector 型の変数を内部パラメータとして読み込む方法を示した．本節では，その Vector 型変数に PARAM1 を乗じて，Vector 型として出力できるように，さらに PublisherInt.cc を改変する．

まずは FlowDesigner の GUI の設定で出力を以下のように追加する．

```
 *
 * @output_name OUTPUT2
 * @output_type Vector<int>
 * @output_description OUTPUT2 = PARAM1 * PARAM2
 *
```

次に，クラスのメンバ変数に出力ポートの ID 用の変数を加える．

```
int output2ID;
```

コンストラクタに以下を追記．

```
output2ID    = addOutput("OUTPUT2");
```

calculate のメインルーチン内で，出力の設定．

```
RCPtr<Vector<int> > output2(new Vector<int>(param2.size()));
(*(outputs[output2ID].buffer))[count] = output2;
```

ここで，PARAM1 * PARAM2 という演算を行って出力を行うことを想定しているので，出力される成分数は PARAM2 と同じになる．よって，output2 は param2.size() 個のサイズとなる．

確保した output2 に PARAM1 * PARAM2 を代入．

```
for(int i = 0; i < param2.size(); i++){
  (*output2)[i] = param1 * param2[i];
}
```

これで PARAM1 * PARAM2 の演算結果を OUTPUT2 から出力できる.

最終的な PublisherInt.cc の改訂版を以下に示す.

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>
#include <Vector.h>

using namespace std;
using namespace FD;

class PublisherInt;

DECLARE_NODE(PublisherInt);
/*Node
 *
 * @name PublisherInt
 * @category HARK_TEST
 * @description This block outputs the same integer as PARAM1.
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description This output the same integer as PARAM1.
 *
 * @output_name OUTPUT2
 * @output_type Vector<int>
 * @output_description OUTPUT2 = PARAM1 * PARAM2
 *
 * @parameter_name PARAM1
 * @parameter_type int
 * @parameter_value 123
 * @parameter_description Setting for OUTPUT1
 *
 * @parameter_name PARAM2
 * @parameter_type Vector<int>
 * @parameter_value <Vector<int> 0 1 2>
 * @parameter_description OUTPUT2 = PARAM1 * PARAM2
 *
END*/

class PublisherInt : public BufferedNode {
  int output1ID;
  int output2ID;
  int output1;
  int param1;
  Vector<int> param2;

public:
  PublisherInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    output2ID    = addOutput("OUTPUT2");
    output1 = 0;
    param1 = dereference_cast<int>(parameters.get("PARAM1"));
    param2 = object_cast<Vector<int> >(parameters.get("PARAM2"));
    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    output1 = param1;
    cout << "Published : [" << count << " , " << output1 << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(output1));

    RCPtr<Vector<int> > output2(new Vector<int>(param2.size()));
    (*(outputs[output2ID].buffer))[count] = output2;

    cout << "Vector Published : [";
    for(int i = 0; i < param2.size(); i++){
      (*output2)[i] = param1 * param2[i];
      cout << (*output2)[i] << " ";
    }
    cout << "]" << endl;

    // Main loop routine ends here.

  }
};
```

ノードに入力を追加する

新しい PublisherInt.cc にて Vector 型の変数を出力できた．本章では，その Vector 型変数を入力として受け取る新しい SubscriberInt.cc を作成する．

{$PACKAGE}/hark_test/src/SubscriberInt.cc

を適当なエディタで開く．Vector 型変数を扱うので，まずは Vector.h を include する．

```
#include <Vector.h>
```

FlowDesigner の GUI の設定を改変する．

/*Node ... END*/

までの中に以下を追記する．

```
 *
 * @input_name INPUT2
 * @input_type Vector<int>
 * @input_description input for a Vector
 *
```

クラスのメンバ変数を追加.

```
int input2ID;
```

コンストラクタに以下を追記.

```
input2ID     = addInput("INPUT2");
```

calculate のメインルーチン内で入力データの読み込み.

```
RCPtr<Vector<int> > input2 = getInput(input2ID, count);
```

あとは，メインルーチン内で以下のように自由に input2 を用いることができる．

```
for(int i = 0; i < (*input2).size(); i++){
  cout << (*input2)[i] << " ";
}
```

最終的な SubscriberInt.cc の改訂版を以下に示す.

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>
#include <Vector.h>

using namespace std;
using namespace FD;

class SubscriberTutorial;

DECLARE_NODE(SubscriberTutorial);
/*Node
 *
 * @name SubscriberTutorial
 * @category HARKD:Tutorial
 * @description This block inputs an integer and outputs the same number with print.
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @input_name INPUT2
 * @input_type Vector<int>
 * @input_description input for a Vector
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description Same as input
 *
END*/

class SubscriberTutorial : public BufferedNode {
  int input1ID;
  int input2ID;
  int output1ID;
  int input1;

public:
  SubscriberTutorial(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    input1ID     = addInput("INPUT1");
    input2ID     = addInput("INPUT2");
    output1ID    = addOutput("OUTPUT1");
    input1 = 0;
    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    ObjectRef inputtmp = getInput(input1ID, count);
    input1 = dereference_cast<int> (inputtmp);

    cout << "Subscribed : [" << count << " , " << input1 << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(input1));

    RCPtr<Vector<int> > input2 = getInput(input2ID, count);

    cout << "Vector Received : [";
    for(int i = 0; i < (*input2).size(); i++){
      cout << (*input2)[i] << " ";
    }
    cout << "]" << endl;

    // Main loop routine ends here.

  }
};
```

新しい PublisherInt.cc と SubscriberInt.cc を使って N ファイルを作成

基本的な手順は前節の N ファイル作成方法と同じ．まずはコンパイル・インストールが正しくできるか確認する．インストールできたら flowdesigner を起動.

前節で作った N ファイルを開く．PublisherInt ノードの出力部に OUTPUT2 が，SubsctiberInt ノードの入力部に INPUT2 が追加されているのが確認できる.（確認できなければ，インストールが正しくできていない

ので，前節のトラブルシュートを参照）それらを接続する．

また以下の設定が必要になる．

```
PublishInt を左クリック > PublisherInt の PARAM2 の type を object に設定 > OK
```

これで保存して,「実行」を押す．

```
Published : [0 , 123]
Vector Published : [0 123 246 ]
Subscribed : [0 , 123]
Vector Received : [0 123 246 ]
Published : [1 , 123]
Vector Published : [0 123 246 ]
Subscribed : [1 , 123]
Vector Received : [0 123 246 ]
Published : [2 , 123]
Vector Published : [0 123 246 ]
Subscribed : [2 , 123]
Vector Received : [0 123 246 ]
...
```

がコンソールに表示され，正しく Vector がやりとりされているのがわかる．

以下，作成したノードのキャプチャである．

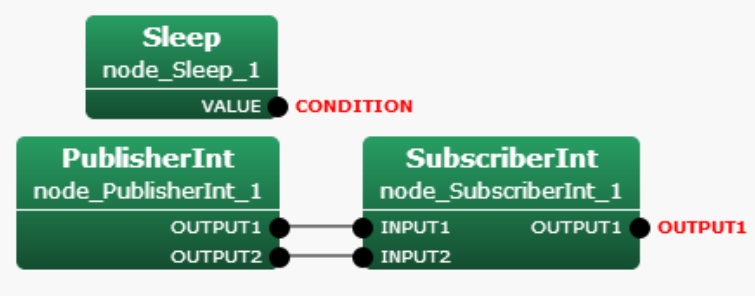


図 13.4: SubscriberInt + PublisherInt ネットワークファイル

複数のフレームにまたがる処理 (lookBack, lookAhead オプション)

これまでは入力されたベクトルをそのまま定数倍するなど，そのフレームだけで処理できる計算を扱っていた．しかしながら，フレーム間で平均を取ったり微分を計算したりする時などは，フレームをまたいだ処理が必要になる．その場合の注意点についてこの章では触れる．

たとえば，今いるフレームから前後 2 フレーム，計 5 フレームのトータルを求めるブロックを作りたいとする．

$$OUTPUT1(t) = INPUT1(t-2)+INPUT1(t-1)+INPUT1(t)+INPUT1(t+1)+INPUT1(t+2)| \quad (13.1)$$

getInput は，入力ポートの ID と count 値それぞれに対して保持されているので，以下のように計算することができる．

```
total = 0.0;
for(int i = -2; i <= 2; i++){
        ObjectRef inputtmp = getInput(input1ID, count + i);
        input1 = dereference_cast<int> (inputtmp);
        total += input1;
}
```

ここで，getInput の中身が ”count + i” となっていることで，前後 2 フレーム目までの処理が行える．しかしながらこれではエラーである．

一つ目の理由として，1 フレーム目と 2 フレーム目で count 値が -2, -1 を要求してしまうため，それは存在しないというエラーである．これは単純に

```
if(count >= 2)
```

で囲むことで解決できる．

二つ目の理由は， FlowDesigner では，特に要求のない限り，現在のフレームから前後 1 フレームしか保持しない仕様になっているため，2 フレーム以上離れたフレームを見ようとすると，それは存在しないというエラーを出すためである．これはコンストラクタに以下のように記述することで，その分のフレーム数の情報をバッファに保持してくれる．

```
inputsCache[input1ID].lookAhead = 2;
inputsCache[input1ID].lookBack = 2;
```

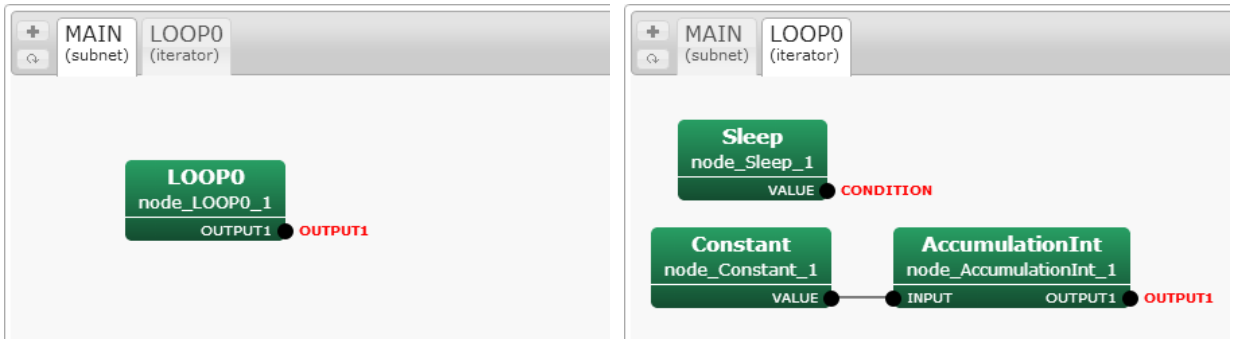
上記は前後 2 フレームまでのバッファを確保するための宣言である．自分の計算にあわせて適切に設定する．

これでフレーム間の計算が行える．

以下に，int 型の入力を，SUM_BACKWARD フレーム前から，SUM_FORWARD フレーム後までの合計を求める “AccumulationInt” ノードを示す．

内部パラメータとして，上記の lookAhead と lookBack を変更できるようにしているので，十分に確保しなければエラーが起こることを確認されたい．

以下に FlowDesigner のノード構築例を示す．



(a) MAIN(subnet) シート　　(b) LOOP0(iterator) シート

図 13.5: AccumulationInt ネットワークファイル

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class AccumulationInt;

DECLARE_NODE(AccumulationInt);
/*Node
 *
 * @name AccumulationInt
 * @category HARKD:Tutorial
 * @description This block takes a summation over several frames of the input.
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description total
 *
 * @parameter_name SUM_FORWARD
 * @parameter_type int
 * @parameter_value 5
 * @parameter_description Forward buffer for summation
 *
 * @parameter_name SUM_BACKWARD
 * @parameter_type int
 * @parameter_value -5
 * @parameter_description Backward buffer for summation
 *
 * @parameter_name LOOK_FORWARD
 * @parameter_type int
 * @parameter_value 0
 * @parameter_description Forward buffer for summation
 *
 * @parameter_name LOOK_BACKWARD
 * @parameter_type int
 * @parameter_value 0
 * @parameter_description Backward buffer for summation
 *
 * @parameter_name IN_ORDER
 * @parameter_type bool
 * @parameter_value true
 * @parameter_description inOrder setting
 *
END*/

class AccumulationInt : public BufferedNode {
  int input1ID;
  int output1ID;
  int input1;
  int sum_forward;
  int sum_backward;
  int look_forward;
  int look_backward;
  int total;
  bool in_order;

public:
  AccumulationInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    input1ID     = addInput("INPUT1");
    output1ID    = addOutput("OUTPUT1");
    input1 = 0;
    sum_forward   = dereference_cast<int>(parameters.get("SUM_FORWARD"));
    sum_backward  = dereference_cast<int>(parameters.get("SUM_BACKWARD"));
    look_forward  = dereference_cast<int>(parameters.get("LOOK_FORWARD"));
    look_backward = dereference_cast<int>(parameters.get("LOOK_BACKWARD"));
    inputsCache[input1ID].lookAhead = look_forward;
    inputsCache[input1ID].lookBack = look_backward;
    in_order = dereference_cast<bool> (parameters.get("IN_ORDER"));
    inOrder = in_order;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    total  = 0;

    if(count + sum_backward >= 0){
      for(int i = sum_backward; i <= sum_forward; i++){
        ObjectRef inputtmp = getInput(input1ID, count + i);
        input1 = dereference_cast<int> (inputtmp);
        total += input1;
      }
    }

    cout << "AccumulationInt : [" << count << " , " << input1 << " , " << total << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(total));

  }
};
```

毎フレーム計算しない処理 (inOrder オプション)

前節とは逆に，本章では，数フレームに一回しか行わない処理をする場合どうなるかを見ていく．例えば以下のようなソースを考える．

```
if(count % 3 == 0){
  ObjectRef inputtmp = getInput(input1ID, count);
  input1 = dereference_cast<int> (inputtmp);
}
```

この場合，getInput は count 毎ではなく，3 回に 1 回しか読まれないことになる．FlowDesigner では，仕様として，前段のノードの処理が，後段の getInput の要求に合わせて処理されるという特徴を持つ．

つまり，上記のようなソースの場合，そのノードは 3 回に 1 回しか入力を要求しないので，その前のノードも，要求された時に 3 回分をまとめて 1 回の処理として演算する．

例を示す．まず，準備として，以下のような SubscriberIntWithPeriod ノードを作成する．

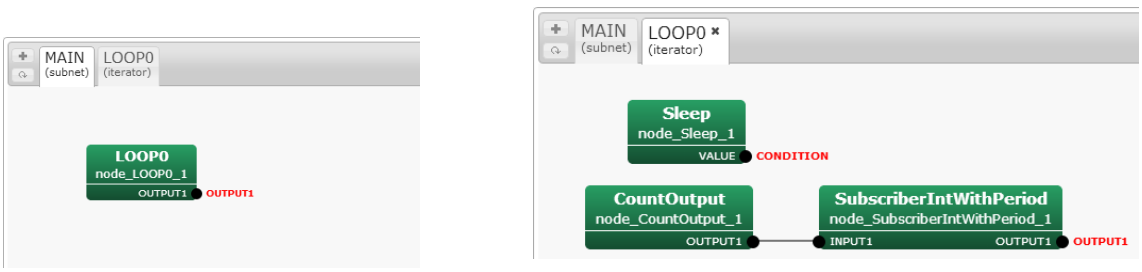
```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class SubscriberIntWithPeriod;

DECLARE_NODE(SubscriberIntWithPeriod);
/*Node
 *
 * @name SubscriberIntWithPeriod
 * @category HARKD:Tutorial
 * @description This block inputs an integer and outputs the same number with print with specific period.
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description Same as input
 *
 * @parameter_name PERIOD
 * @parameter_type int
 * @parameter_value 1
 * @parameter_description Period of INPUT1 subscription
 *
 * @parameter_name IN_ORDER
 * @parameter_type bool
 * @parameter_value true
 * @parameter_description inOrder setting
 *
END*/

class SubscriberIntWithPeriod : public BufferedNode {
  int input1ID;
  int output1ID;
  int input1;
  int period;
  bool in_order;

public:
  SubscriberIntWithPeriod(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    input1ID     = addInput("INPUT1");
    output1ID    = addOutput("OUTPUT1");
    input1 = 0;
    period   = dereference_cast<int>(parameters.get("PERIOD"));
    in_order = dereference_cast<bool> (parameters.get("IN_ORDER"));
    inOrder = in_order;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    if(count % period == 0){
      ObjectRef inputtmp = getInput(input1ID, count);
      input1 = dereference_cast<int> (inputtmp);
    }

    cout << "Subscribed : [" << count << " , " << input1 << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(input1));

    // Main loop routine ends here.

  }
};
```

SubscriberIntWithPeriod ノードでは単純に PERIOD で設定した周期で getInput して，その値を表示する．

次に，現在の count 値をそのまま出力する CountOutput ノードを作成する．

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class CountOutput;

DECLARE_NODE(CountOutput);
/*Node
 *
 * @name CountOutput
 * @category HARKD:Tutorial
 * @description This block outputs the count number
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description This output the same integer as PARAM1.
 *
 * @parameter_name IN_ORDER
 * @parameter_type bool
 * @parameter_value true
 * @parameter_description inOrder setting
 *
END*/

class CountOutput : public BufferedNode {
  int output1ID;
  bool in_order;

public:
  CountOutput(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    in_order = dereference_cast<bool> (parameters.get("IN_ORDER"));
    inOrder = in_order;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    cout << "CountOut : [" << count << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(count));

  }
};
```

二つのノードが完成したら，以下のように FlowDesigner のネットワークファイルを構築する．



(a) MAIN(subnet) シート

(b) LOOP0(iterator) シート

図 13.6: SubscriberIntWithPeriod ネットワークファイル

これで，準備完了．

まず，CountOutput ノードと，SubscriberIntWithPeriod ノードの内部パラメータである IN_ORDER を両方とも ”false” に設定する．(これが FlowDesigner のデフォルト値．) SubscriberIntWithPeriod の PERIOD を 3 にし，ネットワークファイルを実行．すると，コンソールへの出力は次のようになる．

```
CountOut : [0]
Subscribed : [0 , 0]
Subscribed : [1 , 0]
Subscribed : [2 , 0]
CountOut : [3]
Subscribed : [3 , 3]
Subscribed : [4 , 3]
Subscribed : [5 , 3]
CountOut : [6]
Subscribed : [6 , 6]
Subscribed : [7 , 6]
Subscribed : [8 , 6]
CountOut : [9]
Subscribed : [9 , 9]
...
```

このように，後段の SubscriberIntWithPeriod が 3 回に 1 回しか入力を要求しないので，CountOut は 3 回に 1 回しか処理されていないことがわかる．

これでは print デバッグもやりにくく，CountOut の中に微分やカウント演算などの処理がある場合は正しく処理することができない．(この問題を確認したい場合は実際に CountOut ノードに calculate が一回呼び出されるごとにカウントするカウンタを実装すれば確認可能．全ての count に対して，そのカウンタは動作しないことが確認できる．)

これを解決するのが "inOrder" オプションである．

先ほどのネットワークファイルで IN_ORDER を true にする．実行すると次のような結果が得られる．

```
CountOut : [0]
Subscribed : [0 , 0]
Subscribed : [1 , 0]
Subscribed : [2 , 0]
CountOut : [1]
CountOut : [2]
CountOut : [3]
Subscribed : [3 , 3]
Subscribed : [4 , 3]
Subscribed : [5 , 3]
CountOut : [4]
CountOut : [5]
CountOut : [6]
Subscribed : [6 , 6]
Subscribed : [7 , 6]
Subscribed : [8 , 6]
CountOut : [7]
CountOut : [8]
CountOut : [9]
Subscribed : [9 , 9]
...
```

このように，CountOut も 3 回に 1 回の呼び出しに合わせてループ処理が正しく 3 回実行されている．
全てのノードに対して，入力の呼び出しによらず，count 回正しく演算を行いたい場合は，このように，

```
inOrder = true;
```

をコンストラクタに必ず入れる．

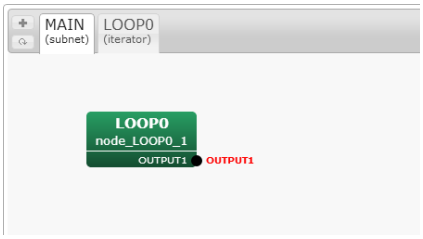
入力ポートへ入る入力数を自由に変更したい（translateInput）

いきなり例を挙げる．次のような，3 つの int 型の整数を入力に取って，その合計を出力するノードを考える．

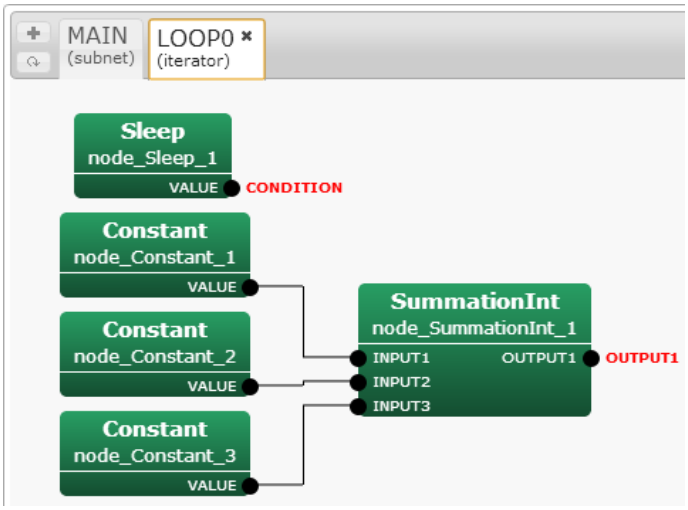
```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class SummationInt;

DECLARE_NODE(SummationInt);
/*Node
 *
 * @name SummationInt
 * @category HARKD:Tutorial
 * @description This block outputs INPUT1 + INPUT2 + INTPUT3
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @input_name INPUT2
 * @input_type int
 * @input_description input for an integer
 *
 * @input_name INPUT3
 * @input_type int
 * @input_description input for an integer
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description Same as input
 *
END*/

class SummationInt : public BufferedNode {
  int input1ID;
  int input2ID;
  int input3ID;
  int output1ID;
  int input1;
  int input2;
  int input3;
  int total;

public:
  SummationInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    input1ID    = addInput("INPUT1");
    input2ID    = addInput("INPUT2");
    input3ID    = addInput("INPUT3");
    output1ID   = addOutput("OUTPUT1");
    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    input1 = 0;
    input2 = 0;
    input3 = 0;

    ObjectRef inputtmp1 = getInput(input1ID, count);
    input1 = dereference_cast<int> (inputtmp1);

    ObjectRef inputtmp2 = getInput(input2ID, count);
    input2 = dereference_cast<int> (inputtmp2);

    ObjectRef inputtmp3 = getInput(input3ID, count);
    input3 = dereference_cast<int> (inputtmp3);

    total = input1 + input2 + input3;

    cout << "SummationInt : [" << count << " , " << total << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(total));

    // Main loop routine ends here.

  }
};
```

このノードはこれまで説明してきた方法で簡単に作ることができる．ノードが作成できたら，次のようにネッ

トワークファイルを作る．



(a) MAIN(subnet) シート



(b) LOOP0(iterator) シート

図 13.7: SummationInt ネットワークファイル

この場合は 1 + 1 + 1 なので 3 が出力されるのは当たり前である．

さて，ここで，このノードから 3 つの変数ではなく，2 つの変数の足し算のみをやりたいとする．この場合，一つの入力ポートを開放して二つのポートに整数を入れて次の図のようにすれば演算できそうである．しかし，実行してみると残念ながら入力ポートに全部情報が入っていないということでエラーとなる．

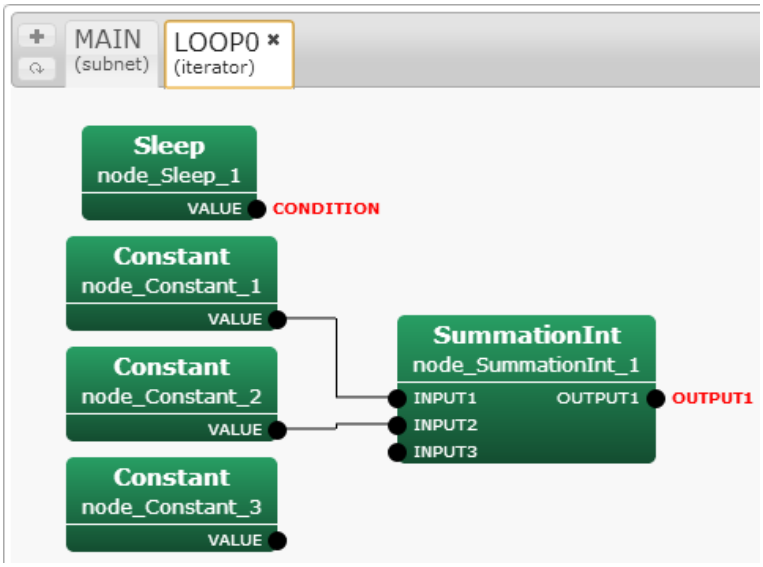


図 13.8: SummationInt ネットワークファイル（入力ポート不完全）

FlowDesigner では，特別な場合を除いて，例えノードの中身の処理の中で使われないデータであっても，`getInput` で読まれる入力ポートについては全て前段から情報を受け取らなければならない仕様がある．これでは，汎用性がないし，型に合う全ての入力を用意するのは時として面倒な場合がある．

そこで，`translateInput` という `BufferedNode` クラスのさらに親クラスである `Node` クラスのメンバ関数

を使うことで，入力ポート数をダイナミックに変更できる方法がある．同じ例を用いて，入力ポート数が可変のノードを作る．

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>

using namespace std;
using namespace FD;

class SummationInt;

DECLARE_NODE(SummationInt);
/*Node
 *
 * @name SummationInt
 * @category HARKD:Tutorial
 * @description This block outputs INPUT1 + INPUT2 + INTPUT3
 *
 * @input_name INPUT1
 * @input_type int
 * @input_description input for an integer
 *
 * @input_name INPUT2
 * @input_type int
 * @input_description input for an integer
 *
 * @input_name INPUT3
 * @input_type int
 * @input_description input for an integer
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description Same as input
 *
END*/

class SummationInt : public BufferedNode {
  int input1ID;
  int input2ID;
  int input3ID;
  int output1ID;
  int input1;
  int input2;
  int input3;
  int total;

public:
  SummationInt(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params), input1ID(-1), input2ID(-1), input3ID(-1)
  {
    output1ID    = addOutput("OUTPUT1");
    inOrder = true;
  }

  virtual int translateInput(string inputName) {
    if (inputName == "INPUT1") {return input1ID = addInput(inputName);}
    else if (inputName == "INPUT2") {return input2ID = addInput(inputName);}
    else if (inputName == "INPUT3") {return input3ID = addInput(inputName);}
    else {throw new NodeException(this, inputName+ " is not supported.", __FILE__, __LINE__);}
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    input1 = 0;
    input2 = 0;
    input3 = 0;

    if (input1ID != -1){
      ObjectRef inputtmp1 = getInput(input1ID, count);
      input1 = dereference_cast<int> (inputtmp1);
    }

    if (input2ID != -1){
      ObjectRef inputtmp2 = getInput(input2ID, count);
      input2 = dereference_cast<int> (inputtmp2);
    }

    if (input3ID != -1){
      ObjectRef inputtmp3 = getInput(input3ID, count);
      input3 = dereference_cast<int> (inputtmp3);
    }

    total = input1 + input2 + input3;

    cout << "SummationInt : [" << count << " , " << total << "]" << endl;
    (*(outputs[output1ID].buffer))[count] = ObjectRef(Int::alloc(total));

    // Main loop routine ends here.
  }
};
```

変更箇所は以下の通り．

- コンストラクタで，入力ポート ID を-1 で初期化した．
  こうしておくことで，入力ポートが解放されている時に，ID が-1 であるよう名前付けしておく．
- `translateInput` を加えた．
  この関数が接続されている入力ポートの数だけ呼び出され，引数として，どのポートが接続されているのかが与えられる．これで，入力ポートが接続されていれば，本当の入力ポートの ID が取得できる．
- `calculate` 関数の中で ID が-1 以外の時に入力を読み込むように処理．
  これで，入力ポートの開放，未開放に合わせた処理が実行できる．

このソースをコンパイルして FlowDesigner で使用すると，入力の数によらず，正しく演算できていることがわかる．

See Also

- ソースコンパイルからのインストール
  HARK 講習会資料「HARK のインストール」参照
- 基本的なノードの作り方
  HARK 講習会資料「ノードの作成」参照

13.2 システムの処理速度をあげたい

Problem

HARK の計算速度を上げるためには？

Solution

処理速度は，基本的にユーザーが構築した HARK のノードの複雑さや，ユーザーの作ったノードのアルゴリズムが支配的である．

例として，LocalizeMUSIC の中の固有値展開の処理や，SaveFeatures のセーブの処理，また，cout や cerr による表示をたくさん行うほど，１カウントにかかる処理時間が長くなる．速度向上には，シンプルに目的を達成できるようなノード構築が第一である．(現在 HARK で提供されているノードは，リアルタイム性を考慮し，アルゴリズムが適宜改良されている．)

シンプルなノード構築をした上で，若干ではあるが，さらに処理速度を向上させるために，以下の２つの方法が主に考えられる．

1) IN_ORDER_NODE_SPEEDUP をコメントイン
   この関数はノードクラスの最後に呼び出す．(LocalizeMUSIC などの cc ファイルに具体的に記述されているので，それをコメントインする)

2) コンパイル時の最適化オプションを変更する．
   make する際にオプションをつけることで実現する．-O，-O1，-O2，-O3 の順により強い条件で最適化を行うので，それだけ高速化される．
   具体的には -O2 の場合
   /src/Makefile.am
   に以下を追加する．

```
libhark_d_la_CXXFLAGS = @GTK_CFLAGS@ -O2
CXXFLAGS = -g -O2
CFLAGS = -g -O2
FFLAGS = -g -O2
```

Discussion

以下それぞれに関してパフォーマンスの評価を行う．評価として，-O，-O1，-O2，-O3 のオプションをつけてコンパイルしたものと，それに更に IN_ORDER_NODE_SPEEDUP を付与したものの計８パターンで処理時間を比較した．

比較に用いたアルゴリズムとしては

```
int count_time = 100000000;
for (i = 0; i < count_time; i++)
  n = n + i;
```

表 13.1: Comaparison of Elapsed Time without IN_ORDER_NODE_SPEEDUP

Option	O3	O2	O1	O
	14.2408	12.7574	14.0147	14.1765
	13.9518	14.0789	14.2417	14.3901
	13.912	14.0633	14.5486	13.7121
	14.3929	13.9978	14.2038	14.1017
	13.7976	14.3931	13.8478	14.2374
	14.0315	13.9962	14.5201	14.1924
	14.3108	14.0069	14.1044	14.1694
	14.0055	14.3397	14.2014	14.5729
	14.004	14.0419	14.467	14.1911
	14.4457	13.8734	14.1159	14.2177
Total	141.0926	139.5486	142.2654	141.9613
Average	**14.10926**	**13.95486**	**14.22654**	**14.19613**

表 13.2: Comaparison of Elapsed Time with IN_ORDER_NODE_SPEEDUP

Option	O3 + speedup	O2 + speedup	O1 + speedup	O + speedup
	14.0007	13.8055	14.3469	14.4444
	14.3702	13.5448	13.9894	14.1628
	14.0753	14.371	14.4229	13.8679
	12.9333	13.8942	14.1801	14.5209
	14.398	13.8926	13.7115	14.0369
	13.6696	14.1745	14.5278	14.7882
	14.0837	14.0613	13.9905	14.5343
	14.4443	14.018	14.0915	14.1182
	13.0798	14.4962	14.4936	14.5952
	13.6339	14.1081	14.1904	14.2751
Total	138.6888	140.3662	141.9446	143.3439
Average	**13.86888**	**14.03662**	**14.19446**	**14.33439**

という単純処理を各ノードに行わせ，そのノードを 100 個直列につなげたもので処理時間の計測を行った．

`IN_ORDER_NODE_SPEEDUP` のない場合の結果を Table 13.1 に，ある場合の結果を Table 13.2 に示す．結果の通り，計算時間の顕著な差は見られなかったが，最適化オプションと `IN_ORDER_NODE_SPEEDUP` の組み合わせにより，3 パーセントほどの速度の向上が見られる．

See Also

参照項なし

13.3 他のシステムと HARK を接続したい

Problem

HARK にて，音源定位や音源分離ができるのがわかったが，その情報を他のシステムに使いたいがどうすればいいか.

この質問に対しては,

- HARK で作った異なるシステム同士の接続
- ソケット通信を用いた他のシステムとの接続
- ROS(Robot Operating System : `http://www.ros.org/wiki/`) との接続

を別々に説明する.

Solution 1

: HARK で作った異なるシステム同士の接続

HARK において一度作ったシステムを一つのサブモジュールとして，複数のサブモジュールの集合を一つのネットワークとして処理したい場合が多くある．例えば，同じアルゴリズムを並列処理させる場合，全ての並列モジュール群を一つの subnet シートに記述するのは膨大な数のノードになってわかりにくいし，手間がかかる.

本稿では，subnet シートに書かれている一つのシステムを一つのノードとして扱う方法を紹介する.

1) **接続したいシステムを記述**
   これは，既存の n ファイルに
   `"Networks" -> "Add Network"`
   として subnet シートを追加して作っても良いし，既存の n ファイルを使用しても構わない．この時，このシートに関する**入力と出力を指定しておく**（接続した後に入出力を変えることも可能なので強制ではない）.

2) **作った subnet シートをエクスポート（注１）**
   作ったシートがアクティブになった状態で
   `"Networks" -> "Export Network"`
   とし，適当な名前をつけて作った subnet シートのみの n ファイルを保存する．このとき，後でインポートした時に subnet シートの名前が保持されるので，エクスポートする前にわかりやすいシート名にすることをおすすめする.

3) **接続したいファイル側でインポート**
   作った subnet シートを一つのノードとして使う側のファイルで
   `"Networks" -> "Import Network"`
   とし，エクスポートしたファイルを選択する．新しい subnet シートとしてファイルが読み込まれる.

4) **サブモジュールとして使用**
   インポートした subnet シートとは別のシートを開き，サブモジュールとして使用する側のシートを用意．新しいシートで右クリックし,
   `"New Node" -> "Subnet"`

から自分の作った subnet シートと同じ名前のノードを選択することで，インポートした subnet シートと同じ入出力数と名前を持ったノードが使用できる．

5) **サブモジュールの改変（オプション）**
インポートした subnet シートは適宜改変することができる．この改変例については（注２）を参照．

この大きなシステムのサブモジュール化により，同じブロック構成を作る手間が省けるし，大規模なネットワークファイルが見やすくなる．

この機能の使用例として以下のような場合が考えられる．

A) **同じ処理を複数回使用する**
同じ処理を繰り返したり，並列処理をさせる場合，その処理をモジュール化しておけば，メインのシートではそのノードを繰り返し使用するだけでいい．極端な例だが，同じ処理を 100 回連続して繰り返す場合，100 回接続するよりも，10 回の処理のシートをサブモジュールとしてそのモジュールを 10 回呼び出す方が，構成が簡易になるのは容易に想像がつく．

B) **いつも使っている処理をテンプレートとして使用する**
例えば音源定位の部分はいつも同じブロック構成を使用し，音源分離の部分だけいつも改変しているような場合，わざわざ音源定位の処理を毎回記述するよりも，その処理を一回テンプレートとしてエクスポートしておいて，一つのノードとして毎回読み込んで同じものを使う方が楽である．

（注１）エクスポート・インポートはわざわざしなくても，エクスポートすべき subnet シートのノードをインポートする側にコピー・ペーストするだけでもいい．しかし，大きな subnet シートのノードを全てコピー・ペーストするのは手間がかかるため，エクスポートを行うよう説明した．

（注２）以下にサブモジュールの改変例について示す．

5-1) **入出力数や名前の変更**
これは当たり前のことかもしれないが，インポートした後に入出力数や subnet シート名を変更し，ノードとして使用する側の仕様を変更することができる．入出力数や名前の変更により，ノードとして使用するメインのシート側の入出力数や名前が変更になるため，接続の因果関係を崩さないようにネットワークを適宜構築しなおすことが必要になる．

5-2) **サブモジュールの中身のパラメータを一つ一つ変える**
インポートしたサブモジュールを複数回使用する場合，その subnet シートの中で設定されているパラメータを，呼び出すごとに変更できるように引数を与えたいことはよくある．この場合，パラメータの型である ‘‘subnet_param’’ を利用する．使用方法は簡単で，インポートした subnet シートで引数を取って一つ一つ変更したいパラメータの型を ’’subnet_param’’ として，値には適当な名前をつける．すると，ノードとして呼び出した方のブロックに，その名前のついたパラメータが追加される．あとは呼び出す側でその引数に具体的な値を入れればよい．

Solution 2

：ソケット通信を用いた他のシステムとの接続

本章では，ソケット通信を用いた HARK 以外のシステムとの接続について説明する．具体的に，現在の HARK は，ソケット通信を用いて，音声認識エンジンである Julius との接続を実現している．通信を担うノードとして，HARK には以下のものが標準で入っている．

1) SpeechRecognitionClient

2) SpeechRecognitionSMNClient

いずれのクライアントも，ソケットを介して，音声の特徴量ベクトルなどの情報をメッセージ送信しており，あとは Julius 側でその処理を行うようになっている．

詳しくは，上記 2 つのソースを参照されたい．ただし，上記のノードは，一つの音声区間をまとめて送信するなど，多少複雑なことをやっているので，次節では，簡単にクライアントとしての HARK ノードの作り方と，サーバとしての HARK ノードの作り方を見ていく．

Solution 2-1 : ソケット通信を用いた他のシステムとの接続（クライアント編）

Julius との通信の場合，Julius はサーバプログラムとして，常に HARK からのメッセージを listen している．この例を簡単なサーバプログラムと HARK ノードを作ることで構築してみる．

まずはサーバプログラム `listener.c` を作る．(これが Julius 側の他のシステムに対応する.)

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

main(int argc, char *argv[])
{
    int    i;
    int    fd1, fd2;
    struct sockaddr_in    saddr;
    struct sockaddr_in    caddr;
    int    len;
    int    ret;
    char   buf[1024];

    if (argc != 2){
        printf("Usage: listener PORT_NUMBER\n");
        exit(1);
    }

    if ((fd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }

    bzero((char *)&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(atoi(argv[1]));

    if (bind(fd1, (struct sockaddr *)&saddr, sizeof(saddr)) < 0){
        perror("bind");
        exit(1);
    }

    if (listen(fd1, 1) < 0) {
        perror("listen");
        exit(1);
    }

    len = sizeof(caddr);

    if ((fd2 = accept(fd1, (struct sockaddr *)&caddr, &len)) < 0) {
        perror("accept");
        exit(1);
    }
    close(fd1);

    ret = read(fd2, buf, 1024);

    while (strcmp(buf, "quit\n") != 0) {
      printf("Received Message : %s", buf);
      ret = read(fd2, buf, 1024);
      write(fd2, buf, 1024); // This returns a responce.
    }

    close(fd2);
}
```

中身は普通のソケット通信のプログラムである．このプログラムによって，fd2 に書き込まれたメッセージを printf 表示する．ソースがカット＆ペーストできたら，コンパイルをしておく．

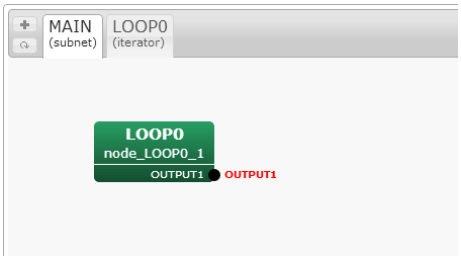
```
> gcc -o listener listener.c
```

次に，HARK 側のクライアントノードを作成する．以下のソースをカット＆ペーストし，TalkerTutorial.cc を作成する．

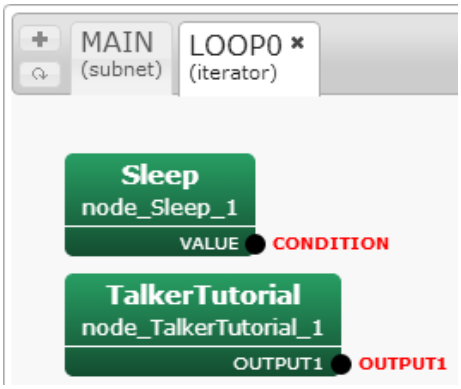
```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>
#include <string.h>
#include <sstream>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <csignal>

using namespace std;
using namespace FD;

class TalkerTutorial;

DECLARE_NODE(TalkerTutorial);
/*Node
 *
 * @name TalkerTutorial
 * @category HARKD:Tutorial
 * @description This block outputs the same integer as PARAM1 and sends it through socket.
 *
 * @output_name OUTPUT1
 * @output_type int
 * @output_description This output the same integer as PARAM1.
 *
 * @parameter_name PARAM1
 * @parameter_type int
 * @parameter_value 123
 * @parameter_description Setting for OUTPUT1
 *
 * @parameter_name PORT
 * @parameter_type int
 * @parameter_value 8765
 * @parameter_description Port number for socket connection
 *
 * @parameter_name IP_ADDR
 * @parameter_type string
 * @parameter_value 127.0.0.1
 * @parameter_description IP address for socket connection
 *
END*/

bool exit_flag2 = false;

class TalkerTutorial : public BufferedNode {
  int output1ID;
  int param1;
  int port;
  string ip_addr;
  struct sockaddr_in    addr;
  struct hostent *hp;
  int    fd;
  int    ret;

public:
  TalkerTutorial(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    param1  = dereference_cast<int>(parameters.get("PARAM1"));
    port    = dereference_cast<int>(parameters.get("PORT"));
    ip_addr = object_cast<String>(parameters.get("IP_ADDR"));

    signal(SIGINT,  signal_handler);
    signal(SIGHUP,  signal_handler);
    signal(SIGPIPE, signal_handler);

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }

    bzero((char *)&addr, sizeof(addr));
    if ((hp = gethostbyname(ip_addr.c_str())) == NULL) {
        perror("No such host");
        exit(1);
    }
    bcopy(hp->h_addr, &addr.sin_addr, hp->h_length);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(port);

    if (connect(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0){
        perror("connect");
        exit(1);
    }

    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    ostringstream message;
    message << "[" << count << " , " << param1 << "]" << endl;
    string buf = message.str();
```

こちらも中身は単純なソケット通信のクライアントノードになっているのみである．count ループ毎に message に格納された文字列をソケット通信している．カット＆ペーストが終わったら，ソースコンパイルからインストールをする．

クライアントとサーバが揃ったところで，Flowdesigner のノードを構築してみる．Sleep で特定の時間周期毎にソケット通信を行うような下記のような簡単なノードである．



(a) MAIN(subnet) シート



(b) LOOP0(iterator) シート

図 13.9: ネットワークファイル：TalkerTutorial

実行するためには，まずはサーバプログラムから起動する．自分で適当なポート番号を決め（ここでは 8765 とする）コマンドライン上で次のようにする．

```
> ./listener 8765
```

次に，Flowdesigner のネットワークファイルの設定に移る．新しいコンソールを起動し，Flowdesigner で先ほど作ったネットワークファイルを起動．

```
Sleep ノード左クリック > SECONDS を float 型で適当な周期に（ここでは 10000）
TalkerTutorial ノードを左クリック > PARAM1 を int 型で適当な値に
TalkerTutorial ノードを左クリック > PORT を int 型で決めたポート番号に（ここでは 8765）
TalkerTutorial ノードを左クリック > IP_ADDR を string 型で IP アドレスまたはホスト名に（ここでは 127.0.0.
```

IP アドレスの設定は同じマシンでサーバとクライアントが動いていることを仮定した．離れたマシンで通信を行うこともちろん可能である．

```
「実行」ボタンクリック
```

すると，動いているサーバに HARK からのメッセージが届き，それぞれのコンソールに以下のように表示される．

サーバ側 (listener)

```
>eceived Message : [0 , 123]
Received Message : [1 , 123]
Received Message : [2 , 123]
...
```

クライアント側 (Flowdesigner：TalkerTutorial)

```
>ent Message : [0 , 123]
Sent Message : [1 , 123]
Sent Message : [2 , 123]
...
```

このように，ソケット通信を用いて，他のシステムに応答を与えることが可能である．

Solution 2-1 : ソケット通信を用いた他のシステムとの接続（サーバ編）

次に，あるクライアントプログラムから，ソケット通信でデータを受け取れるようなサーバノードを作成することを考える．

先ほどと同様に，簡単なクライアントプログラムを用意した後，サーバノードを作ってみる．

まずはサーバプログラム Talker.c を作る．簡単なソケット通信のクライアントプログラムである．

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

main(int argc, char *argv[])
{
    struct sockaddr_in    addr;
    struct hostent *hp;
    int    fd;
    int    len;
    int    port;
    char   buf[1024];
    // int    ret;

    if (argc != 3){
printf("Usage: talker SERVER_NAME PORT_NUMBER\n");
exit(1);
    }

    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }

    bzero((char *)&addr, sizeof(addr));

    if ((hp = gethostbyname(argv[1])) == NULL) {
perror("No such host");
exit(1);
    }
    bcopy(hp->h_addr, &addr.sin_addr, hp->h_length);
    addr.sin_family = AF_INET;
    addr.sin_port = htons(atoi(argv[2]));

    if (connect(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0){
        perror("connect");
        exit(1);
    }

    while (fgets(buf, 1024, stdin)) {
        write(fd, buf, 1024);
        // ret = read(fd, buf, 1024); // This listens a responce.
// buf[ret] = '\0';
        printf("Sent Message : %s",buf);
    }
    close(fd);
    exit(0);
}
```

プログラムから見ても明らかなように，コンソール上で，改行までの文字列を読み込み，その文字列を fd というディスクリプタで表されるソケットに流している．

ファイルができたら，以下でコンパイル．

```
> gcc -o talker talker.c
```

ここから HARK でサーバノードを構築する．重要なことは，クライアントからメッセージが送られてくるタイミングは未知なので，一回メッセージを受信するごとに一連の処理をするようにサーバノードを作成しなければならないことである．そこで，今までは Sleep を条件としたループ演算を行ってきたが，サーバー自身がループ処理のトリガとなるようにノードを作る．以下がその例である．

```
#include <iostream>
#include <BufferedNode.h>
#include <Buffer.h>
#include <string.h>
#include <sstream>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <csignal>

using namespace std;
using namespace FD;

class ListenerTutorial;

DECLARE_NODE(ListenerTutorial);
/*Node
 *
 * @name ListenerTutorial
 * @category HARKD:Tutorial
 * @description This block listens to messages from socket and outputs it.
 *
 * @output_name OUTPUT1
 * @output_type string
 * @output_description Same as the message received from socket.
 *
 * @output_name CONDITION
 * @output_type bool
 * @output_description True if we haven't reach the end of file yet.
 *
 * @parameter_name PORT
 * @parameter_type int
 * @parameter_value 8765
 * @parameter_description Port number for socket connection
 *
END*/

bool exit_flag = false;

class ListenerTutorial : public BufferedNode {
  int output1ID;
  int conditionID;
  int port;

  int    fd1, fd2;
  struct sockaddr_in    saddr;
  struct sockaddr_in    caddr;
  int    len;
  int    ret;
  char   buf[1024];

public:
  ListenerTutorial(string nodeName, ParameterSet params)
    : BufferedNode(nodeName, params)
  {
    output1ID    = addOutput("OUTPUT1");
    conditionID  = addOutput("CONDITION");
    port    = dereference_cast<int>(parameters.get("PORT"));

    signal(SIGINT, signal_handler);
    signal(SIGHUP, signal_handler);
    signal(SIGPIPE, signal_handler);

    if ((fd1 = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("socket");
exit(1);
    }

    bzero((char *)&saddr, sizeof(saddr));

    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(port);

    if (bind(fd1, (struct sockaddr *)&saddr, sizeof(saddr)) < 0){
```

```
perror("bind");
exit(1);
    }

    if (listen(fd1, 1) < 0) {
perror("listen");
exit(1);
    }

    len = sizeof(caddr);

    if ((fd2 = accept(fd1, (struct sockaddr *)&caddr, (socklen_t *)&len)) < 0) {
perror("accept");
exit(1);
    }
    close(fd1);

    inOrder = true;
  }

  void calculate(int output_id, int count, Buffer &out)
  {

    // Main loop routine starts here.

    Buffer &conditionBuffer = *(outputs[conditionID].buffer);
    conditionBuffer[count] = (exit_flag ? FalseObject : TrueObject);

    ret = read(fd2, buf, 1024);
    cout << "Count : " << count << " , Received Message : " << buf << endl;

    ostringstream message;
    message << buf << endl;
    string output1 = message.str();

    (*(outputs[output1ID].buffer))[count] = ObjectRef(new String(output1));

    write(fd2, buf, 1024);

    if(exit_flag){
      cerr << "Operation closed..." << endl;
      close(fd2);
      exit(1);
    }

    // Main loop routine ends here.

  }

  static void signal_handler(int s)
  {
    exit_flag = true;
  }

};
```

ここで，CONDITION という新たな bool 変数を出力するポートが加わったことが，通常のノードとは異なる．この CONDITION ポートは，強制終了があったり，ソケット通信が中断されたりする時以外は常に true を返す出力となっている．このポートを Flowdesigner のネットワークファイルのループのトリガにする．

実際にネットワークファイルを構築してみる．上記のソースをカット＆ペーストしたら，コンパイル＆インストール．Flowdesigner を起動．以下のノードを作成する．

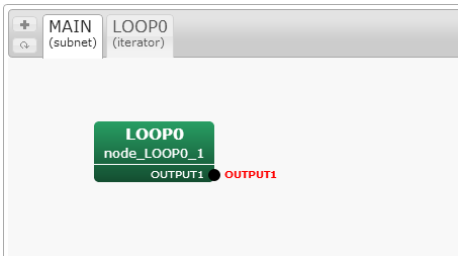
この中で，CONDITION 出力ポートがネットワークファイルのループの CONDITION になっていることに着目されたい．つまり，このネットワークファイルのループ周期は，ListenerTutorial の処理周期と同じになるということである．

ListenerTutorial ノードは，calculate 関数内の

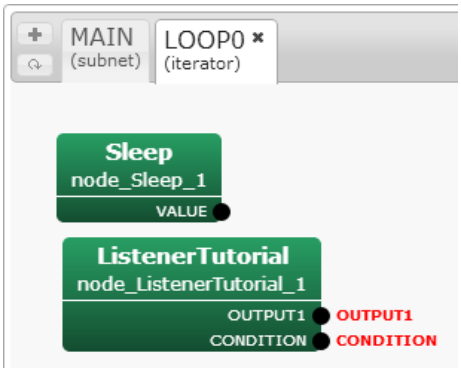
```
ret = read(fd2, buf, 1024);
```

で，メッセージを受信するまで，処理をサスペンドしている．メッセージを受信すると，calculate 関数の全てが処理され，一つの count の処理が終了する．

ネットワークファイルの CONDITION を ListenerTutorial ノードの CONDITION ポートにしたことで，メッセージの一回の受信に合わせて一連の処理を行うことができるようになる．これで，サーバノードはメッセー



(a) MAIN(subnet) シート



(b) LOOP0(iterator) シート

図 13.10: ネットワークファイル：ListenerTutorial

ジ受信のイベントに合わせて対応することができ，満たすべき仕様を実現することができた．

実際に動かしてみよう．まずはサーバプログラム（先ほど作った Flowdesigner のネットワークファイル）から起動する．

ListenerTutorial ノードを左クリック > PORT を int 型で決めたポート番号に（ここでは 8765）
「実行」ボタンをクリック

CONDITION の設定を忘れないように．

次にクライアントプログラムを起動する．新しいコンソールを起動し，先ほどコンパイルした talker.c のあるディレクトリへ移動．コマンドライン上で次のようにする．

```
> ./talker 127.0.0.1 8765
```

ここで，IP アドレスの設定は同じマシンでサーバとクライアントが動いていることを仮定し，127.0.0.1 とした．離れたマシンで通信を行うこともちろん可能である．

talker の方のコンソールに適当な文字を入力してみよう．

すると，動いている Flowdesigner のサーバノードにクライアントコンソールからのメッセージが届き，それぞれのコンソールに以下のように表示される．(例えば，“hoge1”，“hoge2”，“hoge3” と入力した場合)

サーバ側 (talker.c)

```
>oge1
Sent Message : hoge1
hoge2
Sent Message : hoge2
hoge3
Sent Message : hoge3
...
```

クライアント側 (Flowdesigner : ListenerTutorial)

```
>ount : 0 , Received Message : hoge1
Count : 1 , Received Message : hoge2
Count : 2 , Received Message : hoge3
...
```

このように，HARK のノード自身がサーバとなって，他のシステムからのメッセージを受信することも可能である．

Solution 3

: ROS(Robot Operating System) との接続

Willow Garage 社が開発しているオープンソースのロボット用プラットフォーム ROS を使ってシステムを作成している場合，HARK から得た音源定位や分離結果を本節に述べる方法で ROS に渡すことができる．

ROS は web 上のドキュメントが充実しているので，(1) ROS システムは既にインストール済み，(2) topic への publish，topic からの subscribe 等の基本的な概念は既知とする．具体的には， ROS tutorial の Beginner Level がすべて終わっていると仮定する．

接続方法には 2 種類あるので，それぞれについて解説する．また，ROS ノードの実装は Python を用いる．

(1) 標準出力を使う

HARK のネットワークは単体で実行可能であることを利用して，ノード中でサブプロセスとして HARK を実行する方法である．その上で， HARK から定位結果などを標準出力として出力させる (定位結果がほしいなら， LocalizeMUSIC の DEBUG プロパティをオンにすればよい)．

例えば以下のコードでネットワーク `/home/hark/localize.n` を python スクリプト中から実行できる．

```
import subprocess
p = subprocess.Popen( "/home/hark/localize.n",
                      cwd = "/home/hark/",
                      stderr = subprocess.STDOUT,
                      stdout = subprocess.PIPE)
```

続いて，以下のコードでネットワークの出力を python スクリプト中で処理できる．

```
while not rospy.is_shutdown():
    line = p.stdout.readline()
    line から情報を得る
```

こうして HARK から得た情報を使って適切なトピックへ publish すればよい．

(2) ソケット通信を使う

もう一つの方法は，ROS と HARK をソケット通信で接続することである．ROS 側にソケット通信のコードを，HARK 側にソケット通信のノードを作成する必要があるので手間は大きいが，全体の構成はクリアになる．HARK のノードを作成するのは hark-document のチュートリアルを参照されたい．

ここでは，情報を受信する python スクリプトの断片を示す．

```
import socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.bind( ('localhost', 12345) )
sock.listen(1)
client, addr = sock.accept()

while 1:
    received = client.recv( 1024 )
```

(3) トピックを介して通信

最後は, ROS システムを活用する通信方法である．ROS のトピックに対してメッセージを publish するノードを作成すれば, 定位結果や分離音を直接 ROS に送信する事が可能である．

See Also

1. ROS

2. hark-document チュートリアル 3 (ノード作成)

13.4 モータを制御したい

Problem

HARK から直接モータの制御をしたい.

Solution

HARK 自体は聴覚処理のためのソフトウェアであるため，モータを制御するためのモジュールは含まれていない．しかし，比較的簡単な制御を行う場合は，モータを制御するモジュールを自作することで対応が可能である．ここでは，モータの制御をサーバ/クライアント 2 つのアプリケーションに分け，そのうちのクライアント部をモジュールとして HARK に実装する方法を説明する．

以下はクライアントモジュールの主な処理の例である．

---

function calculat

1. *if count = 0 then*
2. *set_IP_address_and_PORT_number*
3. *connect_to_server*
4. *endif*
5. *obtain_input*
6. *generate_control_input*
7. *send_control_input*
8. *check_motor_state*

---

function destructor

1. *disconnect_to_server*

---

1 つ目の calcurate 関数は初めて呼ばれたときに TCP/IP 接続に関係する処理を 1 度だけ行い，それ以外の時は同じ処理を繰り返す．TCP/IP 接続を行うため，この関数が初めて呼ばれたとき，指定された IP アドレス・ポートに設定を行い，サーバアプリケーションに接続する（2・3 行目）．一度サーバアプリケーションに接続すると，入力から制御コマンドを生成する処理を繰り返す．はじめに入力を受け取り，そのデータを元にモータ制御用のコマンドを作成する．そのコマンドをサーバアプリケーションに送り，正しくコマンドが送れたか確認を行う．

2 つ目の destructor 関数はモジュールのデストラクタを表し，ここではサーバとの接続を切断する処理を行う．

サーバアプリケーションはクライアントモジュールからの制御コマンドを受け取り，それに従い実際にモータを駆動させる．このサーバアプリケーションはモータ依存となるため説明は省略する．

Discussion

Flowdesigner はデータフロー指向であるため，1 つでも時間のかかる処理を行うノードがある場合，リアルタイム処理を行うことが困難となる場合がある．そこで，Flowdesigner 上での処理はモータを制御するコマンドの生成にとどめ，実際にモータを動かす処理を別のサーバアプリケーションで行うことで負荷を低減させることができる．また，サーバとクライアントとして切り分けることで，別のモータを制御したい場合でもインターフェースを変更するだけで対応可能となる．

See Also

実際にモータ制御用モジュールを作成する際には，12.1 節の「ノードを作りたい」が参考になる．また，より複雑なモータの制御や他のセンサなどと組み合わせたい場合には，12.2 節の「ロボットを制御したい」が参考となる．

第14章　サンプルネットワーク

14.1　はじめに

本章では，よく使うモジュールを使ったネットワークのサンプルを紹介する．サンプルネットワークは HARK wiki からダウンロードすることができる．標準的なネットワーク構成が一通り揃っているので，作りたいネットワークが，既にサンプルとしてあるかもしれない．作りたいネットワークそのものが無くても，作りたいネットワークに近いカテゴリーのサンプルネットワークを修正し，目的のネットワークを作ると比較的早くネットワークを完成させられる．サンプルネットワークは，機能ごとにカテゴリー化している．

14.1.1　サンプルネットワークのカテゴリー

サンプルネットワークの全カテゴリーを表 14.1 に示す．カテゴリーは 5 つあり，各カテゴリーのサンプルネットワークとサンプルの実行に必要なファイルは，表中右列のサンプルディレクトリに挙げたディレクトリに保存してある．

サンプルネットワークの実行には，各ネットワークファイル名に対応したスクリプトを実行する．ネットワークファイルに与える引数の設定値や，必要な設定ファイルを記述してある．以下で “demo.n” というネットワークファイルの場合の使い方を説明する．

For Ubuntu

ネットワークを実行するスクリプトファイル名は，demo.sh である．

For Windows

ネットワークを実行するバッチファイル名は，demo.bat である．

ただし，これらのスクリプトやバッチファイルには，例えば IP アドレスなどの使用環境に依存した設定項目を含む場合があり，確認なしに実行すると思わぬ結果を招く場合がある．正しい設定方法は，各サンプルネットワークの解説中に示してある．各サンプルのカテゴリーの概要を以下に示す．

表 14.1: サンプルネットワークカテゴリー

	カテゴリー名	サンプルディレクトリ名
1	録音ネットワーク	Record
2	音源定位ネットワーク	Localize
3	音源分離ネットワーク	Separation
4	音響特徴量抽出ネットワーク	FeatureExtraction
5	音声認識ネットワーク	Recognition

- **録音ネットワーク**
  HARK の AudioIO カテゴリーのモジュールを使ったサンプルである．基本的な録音サンプルとしてモノラル録音，ステレオ録音を含む．ステレオ録音とモノラル録音は，ほとんどのハードウェア環境で動作する．
- **音源定位ネットワーク**
  HARK の Localization カテゴリーのモジュールを使ったサンプルである．特に LocalizeMUSIC の使用サンプルとなっている．音源定位結果を DisplayLocalization を使って画面表示し，SaveSourceLocation を使ってファイルに保存するサンプルを用意した．オフラインで音源定位処理を確認できるように，8ch 録

音音声を予め用意した．オフライン処理のため， AD/DA を必要とせず，HARK インストール済みであればどの計算機でも動作可能である．

- **音源分離ネットワーク**
  HARK の Separation カテゴリーのモジュールを使ったサンプルである．特に GHDSS または，GHDSS と HRLE の使用サンプルとなっている．オフライン処理のため， AD/DA を必要とせず，HARK インストール済みのパソコンであればどのパソコンでも動作可能である．

- **音響特徴量抽出ネットワーク**
  HARK の FeatureExtraction カテゴリーのモジュールを使ったサンプルである．特に MSLSExtraction と MFCCExtraction の使用サンプルとなっている．1ch 録音音声からオフラインで音響特徴を抽出するサンプルを用意した．

- **音声認識ネットワーク**
  HARK の 音声認識 (Automatic Speech Recognition; ASR) と (Missing Feature Mask; MFM) カテゴリーのモジュールを使ったサンプルである．具体的には 信頼度の低い特徴量を隠すマスクを生成するモジュール MFMGeneration と、計算した特徴量を Julius へ送信するモジュール SpeechRecognitionClient のを用いたサンプルである．8ch 録音音声にオフライン音源定位処理を適用するサンプルを用意した．

14.1.2 ドキュメントの表記とサンプルネットワークの実行方法

本節説明の作業例は，矩形領域で囲った中に示す．行頭の > はコマンドプロンプトを表す．> に続く行はユーザの入力を，> 無しで文字から始まる行はシステムからのメッセージを表す．例えば，

```
> echo Hello World!
Hello World!
```

図 14.1: サンプルネットワーク起動スクリプトの実行例.

という作業例で，1 行目の先頭 > は，コマンドプロンプトを表している．作業環境によってプロンプトの表示が異なるので，各自の環境に合せて読み換える必要がある．1 行目のプロンプト以降の部分は，ユーザが実際に入力する部分である．ここでは， echo Hello World! の 17 文字（スペースを含む）がユーザの入力部分である．行末では，Enter キーを入力する．2 行目は，システムの出力である． 1 行目の行末で Enter キーの入力後，表示される部分である．

14.2 録音ネットワークサンプル

デバイスを用いてオンラインで録音する場合，その方法は Ubuntu と Windows で大きく異なるため，ここでは分けて説明する．サンプルネットワークファイルも，OS に応じて RecordLin と RecordWin とに分けて入れてある．

14.2.1 Ubuntu

Ubuntu で用いる録音ネットワークサンプルは，表 14.2 に示すように 4 種類ある．左列に，ネットワークファイルを，右列に処理内容を示している．これらのサンプルを実行するには，ターミナルで ./ファイル名 [録音時間] とすればよい．図 14.2 にその実行例を示す．

表 14.2: 録音ネットワーク一覧.

ネットワークファイル名	処理内容
demoALSA_2ch.n	ALSA 準拠デバイスを用いた 2ch 録音
demoWS_8ch.n	無線 RASP を用いた 8ch 録音
demoTDBD_8ch.n	TD-BD-16AD-USB を用いた 8ch 録音
demoRASP24_8ch.n	RASP24 を用いた 8ch 録音

```
> ./demoALSA_2ch.n 100
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network  :MAIN

 <Bool 1 >
```

図 14.2: demoALSA1ch の実行例.

ただし，各サンプルネットワークファイルで使用するモジュールは全て共通であり、違いは，AudioStreamFromMic モジュールに指定するパラメータだけであることに注意されたい．各サンプルで設定されている DEVICE パラメータ (ALSA の場合は plughw:1,0, RASP の場合は 127,0,0,1 など) は，環境によって変化する．そのため，この DEVICE パラメータは各自で設定を変更する必要がある．

ネットワークファイルは二つのサブネットワーク (MAIN, MAIN_LOOP) から構成されており，MAIN (subnet) に 1 個，MAIN_LOOP (iterator) に 6 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.3, 14.4 に示す．AudioStreamFromMic モジュールで取り込んだマルチチャンネルの音声から ChannelSelector モジュールを使って特定のチャンネルだけを選び，SaveWavePCM でファイルに書き出す単純なネットワーク構成である．ネットワークに SaveWavePCM モジュールが 2 個あるが，これはマルチチャンネルの音声を録音する際に二つの方法があるからである (モノラル音声の場合はどちらも同じ)．AudioStreamFromMic モジュールからの出力を直接 SaveWavePCM モジュールに接続した場合は，1 つのファイルに全てのチャンネルの音声が含まれる．一方で MatrixToMap を介して接続した場合は，各チャンネル毎にファイルが作成される．Iterate は設定した値の回数だけ実行を繰り返すために使うモジュールである．繰り返し回数が MAX_ITER より小さい間のみ true を出力する．そのため，このノードの出力を繰り返しの終了判定に用いることで取り込むフレーム数を指定し，録音時間を調節することができる．

MAIN (subnet) にある MAIN_LOOP モジュールのプロパティは，5 個ある．表 14.3 に一覧を示す．SAMPLING_RATE と GET_FRAMES が重要である．この表の通りに各パラメータ値が設定されている．GET_FRAMES の設定値が，`int` :ARG1 となっているが，この意味は，サンプルネットワークファイルの第一引数を整数型にキャストして代入する意味である．録音時間長は，取得フレーム数で指定し，実際の録音時間長を秒で表すと，

$$(LENGTH + (GET\_FRAMES - 1) * ADVANCE)/SAMPLING\_RATE \tag{14.1}$$

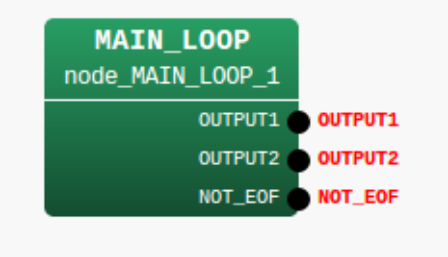


図 14.3: MAIN (subnet)

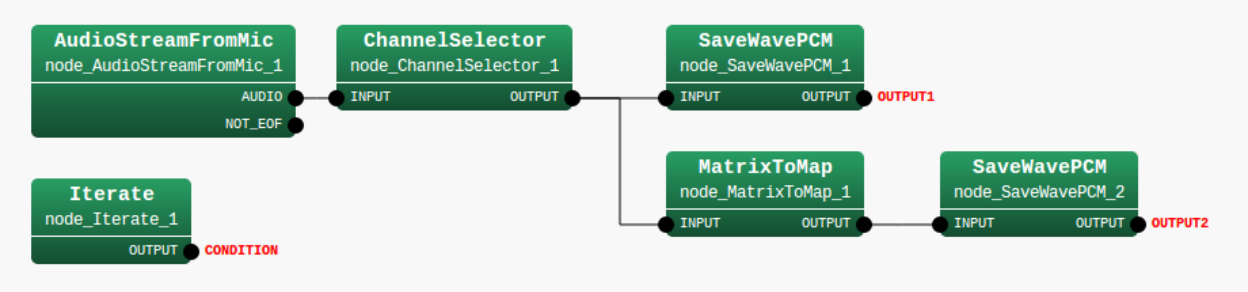


図 14.4: MAIN_LOOP (iterator)

である．

表 14.3: **MAIN_LOOP** のパラメータ表

パラメータ名	型	設定値	単位	説明
ADVANCE	`int`	160	[pt]	シフト長
LENGTH	`int`	512	[pt]	FFT 長
SAMPLING_RATE	`int`	16000	[Hz]	サンプリング周波数
GET_FRAMES	`subnet_param`	`int` :ARG1		録音フレーム数
DOWHILE	`bool`			空欄にする

ALSA 準拠デバイスによる録音

RecordLin ディレクトリにある demoALSA_2ch.n を実行する．録音が正常に終了すると，一つのファイルに 2ch の音声を録音した rec_all_0.wav というファイルと，各チャンネル毎に一つのファイルにした rec_each_0.wav, rec_each_1.wav というファイルを生成する．

うまく録音できないときは，次のチェックをする．

1. HARK 以外の録音ソフトで音の再生や録音は可能か確認する．再生や録音ができない場合は，OS やドライバの設定，オーディオインターフェースが正しく接続／設定されていない可能性がある．そちらを確認する．これには，例えば audacity やデバイスに付属するサンプルを用いる．
2. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．
3. PC のマイク端子がプラグインパワーに対応しているか確認する．プラグインパワーに対応していない時は，マイクロホンに電源を供給する必要がある．マイクロホンに電池を入れるタイプでは電池を入れ，スイッチを入れる．電池ボックスを接続するタイプでは電池ボックスを接続しスイッチを入れる．
4. 2 つ以上のオーディオインターフェースを接続している場合は，2 台目以降のオーディオインターフェースを外してから，録音する．接続オーディオインターフェースを 1 台にしても録音できない場合には，demo.n のプロパティを変更する．AudioStreamFromMic モジュールの DEVICE プロパティを plughw:0,0 plughw:0,1 plughw:1,0 などの設定を試す．

AudioStreamFromMic のパラメータを表 14.4 に示す．このモジュールで，録音デバイスを指定する．このデモでは，録音チャネル数を 2 ch にし， DEVICETYPE に ALSA，DEVICE は plughw:0,0 を指定した．

表 14.4: AudioStreamFromMic のパラメータ表

パラメータ名	型	設定値	単位	説明
LENGTH	subnet_param	LENGTH	[pt]	FFT 長
ADVANCE	subnet_param	ADVANCE	[pt]	シフト長
CHANNEL_COUNT	int	2	[ch]	録音チャネル数
SAMPLING_RATE	subnet_param	16000	[Hz]	サンプリング周波数
DEVICETYPE	string	ALSA		デバイスタイプ
DEVICE	string	plughw:0,0		デバイス名

無線 RASP による 8ch 録音

RecordLin ディレクトリにある demoWS_8ch.n を実行する．ただし，demoWS_8ch.n には，無線 RASP の IP アドレスとして 127.0.0.1 を指定しているが，これを適切な IP アドレスに変更する必要がある．また，もし無線 RASP の FPAA のコンフィグレーションが終っていなければ，ここで実行する．ws_fpaa_config 実行方法は無線 RASP のドキュメントを参考にする．録音が正常に終了すると，ALSA 準拠デバイスの時と同様に 8ch の音声ファイル rec_all_0.wav と，各チャンネル毎にファイルにした rec_each_0.wav，rec_each_1.wav，...，rec_each_7.wav というファイルが生成される．

うまく録音できないときは，次のチェックをする．

1. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．
2. RASP の IP アドレスにネットワーク接続されているか？ ping を使ってネットワーク接続を確認する．
3. AudioStreamFromMic に RASP の正しい IP アドレスを設定しているか？
4. RASP の FPAA の初期化が完了しているか？
5. HARK 以外の録音ソフトで音の再生や録音は可能か確認する．再生や録音ができない場合は，OS やドライバの設定，オーディオインターフェースが正しく接続／設定されていない可能性がある．そちらを確認する．

AudioStreamFromMic のパラメータを表 14.5 に示す．DEVICETYPE に 無線 RASP を表す WS を，DEVICE には IP アドレスを指定する．サンプルを実行するには，実際の無線 RASP のアドレスに変更する必要がある．

無線 RASP と接続する場合には，AudioStreamFromMic モジュールの CHANNEL_COUNT を 16 にして使用する必要がある．そのため， 8 ch 録音するためには，16 ch 中から 8 ch 分を選択しなければならない．この例では ChannelSelector で，0〜7 ch を選択している．

表 14.5: AudioStreamFromMic のパラメータ表

パラメータ名	型	設定値	単位	説明
LENGTH	subnet_param	LENGTH	[pt]	FFT 長
ADVANCE	subnet_param	ADVANCE	[pt]	シフト長
CHANNEL_COUNT	int	16	[ch]	録音チャネル数
SAMPLING_RATE	subnet_param	16000	[Hz]	サンプリング周波数
DEVICETYPE	string	WS		デバイスタイプ
DEVICE	string	127.0.0.1		デバイス名

TD-BD-16AD-USB による録音

RecordLin ディレクトリにある demoTDBD_16ch.n を実行する．録音が正常に終了すると，ALSA 準拠デバイスの時と同様に 8ch の音声ファイル rec_all_0.wav と，各チャンネル毎にファイルにした rec_each_0.wav, rec_each_1.wav，...，rec_each_7.wav というファイルが生成される．
うまく録音できないときは，次のチェックをする．

1. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．
2. デバイスに付属しているサンプルプログラムを実行してみる．うまく録音できない場合には，カーネルモジュールのインストールがちゃんとできているか，デバイスにアクセスできているかなどを確認する．

AudioStreamFromMic のパラメータを表 14.6 に示す．TD-BD-16AD-USB を用いる場合は，DEVICETYPE を TDBD16ADUSB に，DEVICE を SINICH にする．

表 14.6: AudioStreamFromMic のパラメータ表

パラメータ名	型	設定値	単位	説明
LENGTH	subnet_param	LENGTH	[pt]	FFT 長
ADVANCE	subnet_param	ADVANCE	[pt]	シフト長
CHANNEL_COUNT	int	16	[ch]	録音チャネル数
SAMPLING_RATE	subnet_param	16000	[Hz]	サンプリング周波数
DEVICETYPE	string	TDBD16ADUSB		デバイスタイプ
DEVICE	string	SINICH		デバイス名

このデバイスは 16 ch 録音が可能である．このサンプルでは，ChannelSelector で，0〜7 ch を選択して録音している．

RASP24 による録音

RecordLin ディレクトリにある demoRASP24_8ch.n を実行する．録音が正常に終了すると，ALSA 準拠デバイスの時と同様に 8ch の音声ファイル rec_all_0.wav と，各チャンネル毎にファイルにした rec_each_0.wav, rec_each_1.wav，...，rec_each_7.wav というファイルが生成される．
うまく録音できないときは，次のチェックをする．

1. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．
2. RASP の IP アドレスにネットワーク接続されているか？ ping を使ってネットワーク接続を確認する．
3. AudioStreamFromMic に RASP の正しい IP アドレスを設定しているか？

AudioStreamFromMic のパラメータを表 14.7 に示す．RASP24 を用いる場合は，量子化ビット数を 16, 32 bit から選ぶことができる．DEVICETYPE の RASP24-16 は 16 bit で，RASP24-32 は 32 bit である (RASP24 はデバイス名)．DEVICE には IP アドレスを指定する．RASP24 を選んだ場合，図 14.5 のように追加のオプションとして GAIN という項目が増える．GAIN は，入力がクリッピングしない範囲で選ぶ．

表 14.7: AudioStreamFromMic のパラメータ表

パラメータ名	型	設定値	単位	説明
LENGTH	subnet_param	LENGTH	[pt]	FFT 長
ADVANCE	subnet_param	ADVANCE	[pt]	シフト長
CHANNEL_COUNT	int	16	[ch]	録音チャネル数
SAMPLING_RATE	subnet_param	16000	[Hz]	サンプリング周波数
DEVICETYPE	string	RASP24-16		デバイスタイプ
GAIN	string	0dB		ゲイン
DEVICE	string	127.0.0.1		デバイス名

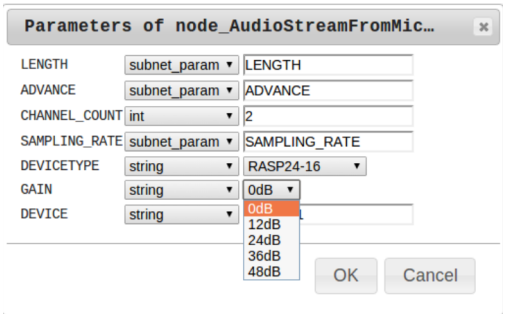


図 14.5: Additional parameter for RASP24

14.2.2 Windows

Windows で使用可能な録音ネットワークサンプルは，表 14.8 に示すように 2 種類ある．左列に，ネットワークファイルを，右列に処理内容を示している．これらのサンプルを実行するには，コマンドプロンプト batchflow ファイル名 [録音時間] とすればよい．図 14.6 にその実行例を示す．

表 14.8: 録音ネットワーク一覧.

ネットワークファイル名	処理内容
demoDS_4.n	Kinect を用いた 4ch 録音
demoRASP24_8ch.n	RASP24 を用いた 8ch 録音

```
> batchflow demoDS_4ch.n 100
```

図 14.6: demoDS_4ch の実行例.

ただし，各サンプルネットワークファイルで使用するモジュールは全て共通であり、違いは，AudioStreamFromMic モジュールに指定するパラメータだけであることに注意されたい．各サンプルで設定されている DEVICE パラメータ (DS の場合は kinect, RASP24 の場合は 127,0,0,1 など) は，環境によって変化する．そのため，この DEVICE パラメータは各自で設定を変更する必要がある．

ネットワークファイルに含まれるモジュールや，それらの構成は Ubuntu の場合と同じであるため，ここではそれぞれのデバイスを用いる際のパラメータ設定についてのみ述べる．

DS 準拠デバイスによる録音

RecordWin ディレクトリにある demoDS_4ch.n を実行する．録音が正常に終了すると，一つのファイルに 4ch の音声を録音した rec_all_0.wav というファイルと，各チャンネル毎に一つのファイルにした rec_each_0.wav, rec_each_1.wav というファイルを生成する．

うまく録音できないときはへの対処法も，Ubuntu の場合と同様である．

1. HARK 以外の録音ソフトで音の再生や録音は可能か確認する．再生や録音ができない場合は，OS やドライバの設定，オーディオインターフェースが正しく接続／設定されていない可能性がある．そちらを確認する．これには，例えば audacity やデバイスに付属するサンプルを用いる．

2. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．

3. PC のマイク端子がプラグインパワーに対応しているか確認する．プラグインパワーに対応していない時は，マイクロホンに電源を供給する必要がある．マイクロホンに電池を入れるタイプでは電池を入れ，スイッチを入れる．電池ボックスを接続するタイプでは電池ボックスを接続しスイッチを入れる．

4. 2 つ以上のオーディオインターフェースを接続している場合は，2 台目以降のオーディオインターフェースを外してから，録音する．

5. Kinect を用いる際は，ドライバを事前にインストールしておく必要がある．そのため，録音できない場合には，ドライバがインストールされていない可能性があるので，確認する．

AudioStreamFromMic のパラメータを表 14.9 に示す．このモジュールで，録音デバイスを指定する．このデモでは，録音チャネル数を 4 ch にし， DEVICETYPE に DS，DEVICE は Kinect を指定した．

表 14.9: AudioStreamFromMic のパラメータ表

パラメータ名	型	設定値	単位	説明
LENGTH	`subnet_param`	LENGTH	[pt]	FFT 長
ADVANCE	`subnet_param`	ADVANCE	[pt]	シフト長
CHANNEL_COUNT	`int`	4	[ch]	録音チャネル数
SAMPLING_RATE	`subnet_param`	16000	[Hz]	サンプリング周波数
DEVICETYPE	`string`	DS		デバイスタイプ
DEVICE	`string`	Kinect		デバイス名

RASP24 による録音

基本的な使い方は Ubuntu とほとんど同じである．RecordWin ディレクトリにある demoRASP24_8ch.n を実行する．録音が正常に終了すると，ALSA 準拠デバイスの時と同様に 8ch の音声ファイル rec_all_0.wav と，各チャンネル毎にファイルにした rec_each_0.wav，rec_each_1.wav，...，rec_each_7.wav というファイルが生成される．

うまく録音できないときは，次のチェックをする．

1. マイクロホンは接続されいるか．プラグが抜けていたり，緩んでいないか確認し，しっかり接続する．
2. RASP の IP アドレスにネットワーク接続されているか？ ping を使ってネットワーク接続を確認する．
3. AudioStreamFromMic に RASP の正しい IP アドレスを設定しているか？

AudioStreamFromMic のパラメータも表 14.7 を参考にするとよい．

14.3　音源定位ネットワークサンプル

ここでは, 2 種類の音源定位ネットワークサンプルを提供する (表 14.10)。すべてのファイルは `demo.sh` から実行できる。左から順に、ネットワークファイル、`demo.sh` の引数、処理内容を示している.

表 14.10: 音源定位ネットワーク一覧.

ネットワークファイル名	実行方法	処理内容
localization_offline_microcone.n	demo.sh offline	7ch 音声ファイルの音源定位
localization_online_microcone.n	demo.sh online	Microcone を用いたオンライン音源定位

14.3.1　オフライン音源定位

実行方法

Localization ディレクトリに移動して、音源定位を実行しよう。ここでは，3 つの 7ch 音声ファイルを順番に読み込み，定位結果を表示する．音声ファイルはそれぞれ 0，90 度方向からの音声 (`microcone_0deg.wav`, `microcone_90deg.wav`) と、移動しながら発話した音声 (`microcone_moving.wav`) である。すべて data ディレクトリにあるので確認しよう。

次のコマンド

```
> ./demo.sh offline
```

を実行すると、図 14.7 のような出力と，音源定位結果のグラフが表示されるはずだ.

```
UINodeRepository::Scan()
Scanning def /usr/lib/flowdesigner/toolbox
done loading def files
loading XML document from memory
done!
Building network  :MAIN
TF was loaded by libharkio2.
1 heights, 72 directions, 1 ranges, 7 microphones, 512 points
Source 0 is created.
  (中略)
Source 5 is removed.
UINodeRepository::Scan()
 (以下略)
```

図 14.7: オフライン音源定位の実行結果

実行結果の確認

実行後は，result_.*txt という名前のファイルと, log_*.txt という名前のファイルがそれぞれ 3 つずつできているはずだ。もしできていなければ実行に失敗しているので、次のチェックをしよう。

1. ../data ディレクトリに microcone_0deg.wav, microcone_90deg.wav, microcone_moving.wav かチェックする．これらのファイルは， HARK がサポートするマイクアレイ Microcone で実際に録音した音声だ．入力ファイルなので、これがないと実行できない。

2. ../config ディレクトリに microcone_loc.dat ファイルがあるかチェックする．これは LocalizeMUSIC に必要な伝達関数ファイルだ。詳しくは HARK document の LocalizeMUSIC の節を参照。

result_*.txt には，音源定位結果を SaveSourceLocation ノードを使って保存したテキストファイルで， フレーム番号と音源方向が記録されている．LoadSourceLocation ノードを使えば、これを後で表示させることもできる。詳しいフォーマットは HARK document の定位結果テキスト の節を参照.

log_*.txt には, LocalizeMUSIC の DEBUG プロパティを true すると出力される MUSIC スペクトルが記録されている。MUSIC スペクトルとは時刻・方向ごとに計算した音源の存在する信頼度のようなもので、値が大きいところに音源がある。詳しくは HARK documetn の LocalizeMUSIC の節を参照。次のように表示用プログラムを実行すれば、それぞれの MUSIC スペクトルを表示できる。

```
> python plotMusicSpec.py log_microcone_90deg.txt
> python plotMusicSpec.py log_microcone_0deg.txt
> python plotMusicSpec.py log_microcone_moving.txt
```

もし実行に失敗するようなら、次のコマンドを実行して必要なパッケージをインストールしよう。

For Ubuntu

```
sudo apt-get install python python-matplotlib
```

ちなみに、これで表示されるカラーバーの値が SourceTracker の THRESH の設定に使える。

ネットワークの解説

本サンプルに含まれるノードは，9 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 6 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.8,14.9 に示す．AudioStreamFromWave で取り込んだ音声波形を MultiFFT で分析し，LocalizeMUSIC で音源定位する．定位結果を SourceTracker でトラッキングし DisplayLocalization で画面表示し，SaveSourceLocation でファイルに書き出す.

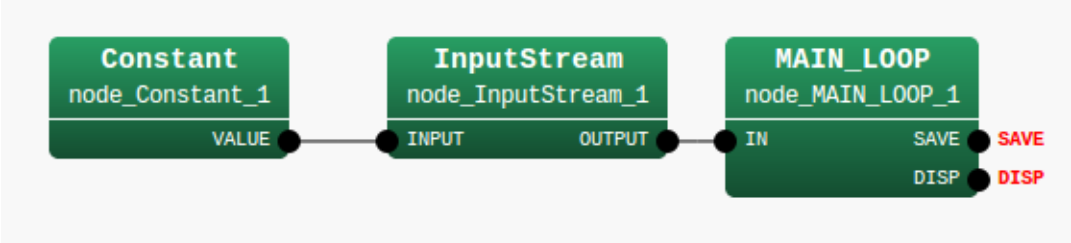


図 14.8: MAIN (subnet)

表 14.11 に主要なパラメータを示す．重要なパラメータは，伝達関数ファイルを表す A_MATRIX である．このファイルは HARK web ページで提供しているマイクロホンアレイを使う場合はダウンロードできるが、そうでない場合は自分で harktool を使って作成する必要がある.

14.3.2 オンライン音源定位

実行方法

まずは, Microcone を計算機の USB ポートに接続しよう。そして次のコマンドを実行しよう。

```
> cat /proc/asound/cards
 0 [AudioPCI       ]: ENS1371 - Ensoniq AudioPCI
                      Ensoniq AudioPCI ENS1371 at 0x2080, irq 16
 1 [Microcone      ]: USB-Audio - Microcone
                      DEV-AUDIO Microcone at usb-0000:02:03.0-1, high speed
```

上記のように Microcone と表示されれば、正しく接続されている。Microcone の左側が 1 なので、この場合のデバイス名は plughw:1 である。localization_online_microcone.n を flowdesigner で開き、AudioStreamFromMic の DEVICE を plughw:1 に設定しよう。

音源定位の実行は以下のコマンドで行える。

```
> ./demo.sh online
```

図 14.10 に示すような出力と，音源定位結果が表示されるはずだ.

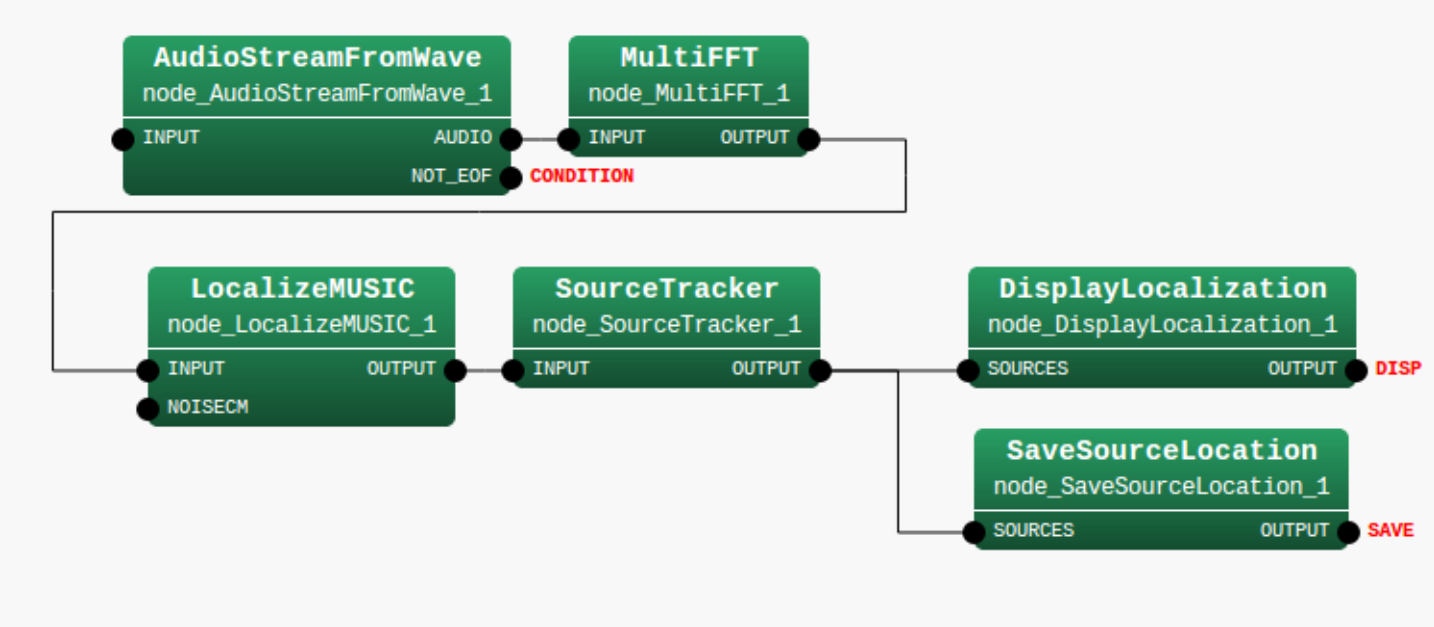


図 14.9: MAIN_LOOP (iterator)

```
UINodeRepository::Scan()
Scanning def /usr/lib/flowdesigner/toolbox
done loading def files
loading XML document from memory
done!
Building network  :MAIN
TF was loaded by libharkio2.
1 heights, 72 directions, 1 ranges, 7 microphones, 512 points
Source 0 is created.
Source 0 is removed.
 以下略
```

図 14.10: オンライン音源定位の実行例.

実行結果の確認

うまく定位できないときは，オフライン音源定位の場合と同じファイルのチェックを行おう。ほかにも，レシピ: うまく定位できない を見て問題を調べよう.

ネットワークの解説

本サンプルに含まれるモジュールは，7 個である．MAIN (subnet) に 1 個 MAIN_LOOP (iterator) に 6 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.11,14.12 に示す．AudioStreamFromMic で音声波形を取り込む。その出力は SaveWavePCM で音声ファイルに保存される。同時に MultiFFT でスペクトルに変換もされ，LocalizeMUSIC がフレームごとに音源定位を行う。それを， SourceTracker で 時間的連続性などを用いてトラッキングし、DisplayLocalization で音源定位結果を表示する．

表 14.12 に主要なパラメータを示す.

表 14.11: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	int	512
	ADVANCE	int	160
	SAMPLING_RATE	int	16000
	A_MATRIX	int	ARG2
	FILENAME	subnet_param	ARG3
	DOWHILE	bool	(空欄)
LocalizeMUSIC	NUM_CHANNELS	int	8
	LENGTH	subnet_param	LENGTH
	SAMPLING_RATE	subnet_param	SAMPRING_RATE
	A_MATRIX	subnet_param	A_MATRIX
	PERIOD	int	50
	NUM_SOURCE	int	1
	MIN_DEG	int	-180
	MAX_DEG	int	180
	LOWER_BOUND_FREQUENCY	int	500
	HIGHER_BOUND_FREQUENCY	int	2800
	DEBUG	bool	false

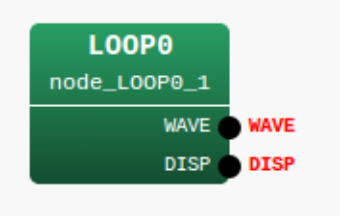


図 14.11: MAIN (subnet)

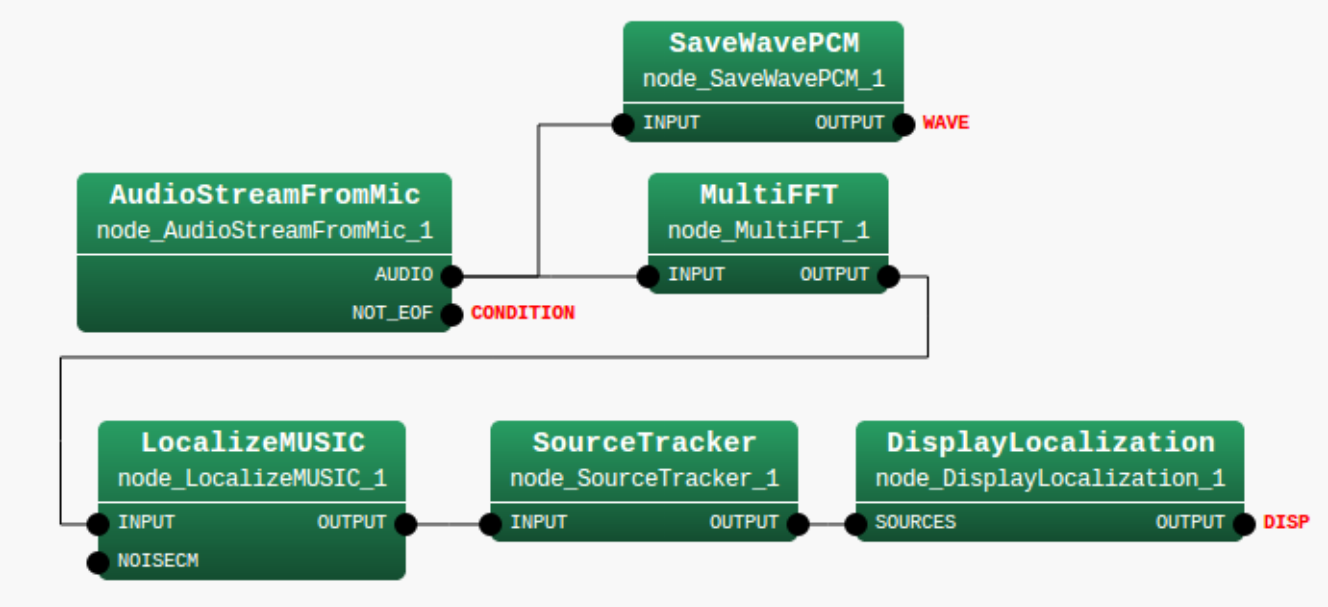


図 14.12: MAIN_LOOP (iterator)

表 14.12: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	int	512
	ADVANCE	int	160
	SAMPLING_RATE	int	16000
	A_MATRIX	string	ARG1
	DOWHILE	bool	(空欄)
LocalizeMUSIC	NUM_CHANNELS	int	8
	LENGTH	subnet_param	LENGTH
	SAMPLING_RATE	subnet_param	SAMPRING_RATE
	A_MATRIX	subnet_param	A_MATRIX
	PERIOD	int	50
	NUM_SOURCE	int	1
	MIN_DEG	int	-180
	MAX_DEG	int	180
	LOWER_BOUND_FREQUENCY	int	500
	HIGHER_BOUND_FREQUENCY	int	2800
	DEBUG	bool	false

14.4　音源分離ネットワークサンプル

音源分離ネットワークサンプルは，表 14.13 に示すように 4 種類ある．左列に，ネットワークファイルを，中列に，ネットワークを動かすためのシェルスクリプトファイルを，右列に処理内容を示している．AD/DA に ALSA 準拠のデバイスである Microsoft 社の Kinect を使ったサンプルを用意した．

表 14.13: 音源分離ネットワーク一覧.

ネットワークファイル名	ネットワーク実行スクリプト	処理内容
demoOfflineKinect.n	demoOfflineKinect.sh	4ch 音声ファイルの音源分離
demoOfflineKinectHRLE.n	demoOfflineKinectHRLE.sh	4ch 音声ファイルの音源分離と HRLE を用いた後処理
demoOnlineKinect.n	demoOnlineKinect.sh	Kinect で録音し音源分離
demoOnlineKinectHRLE.n	demoOnlineKinectHRLE.sh	Kinect で録音し音源分離と HRLE を用いた後処理

14.4.1　オフライン音源分離

はじめにオフライン音源分離のサンプルを紹介する．入力音声が，ファイルであるため，マルチチャンネル AD/DA を持っていない場合でも，音源分離処理を実行しながら確認できる．

Separation ディレクトリに含む demoOfflineKinect.sh を実行する．図 14.13 実行例を示す.

```
> demoOfflineKinect.sh
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
   ... skipped ...
```

図 14.13: demoOfflineKinect.sh の実行例.

実行後，音源分離され，分離音声が生成される．

うまく実行できないときは，次の項目をチェックをする．

1. ../config ディレクトリに kinect_loc.dat，kinect_sep.tff ファイルがあるかチェックする．Kinect のインパルス応答ファイルである．このファイルがなければ，サンプルは失敗する．

2. ../data ディレクトリに kinect_20words.wav ファイルがあるかチェックする．このファイルは，-45，0 度方向からの二話者同時発話を Kinect により録音した音声ファイルである．このファイルがなければ，音源分離の入力がないことになり，サンプルは失敗する．

本サンプルに含まれるモジュールは，12 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 9 個のモジュールがある．MAIN (subnet) には，Constant モジュールと InputStream ，と MAIN_LOOP (iterator) がある．MAIN_LOOP (iterator) は，図 14.14 に示すネットワークとなっている．AudioStreamFromWave モジュールでファイルから読み出した音声波形を MultiFFT で周波数領域に変換し，LocalizeMUSIC ，SourceTracker ，SourceIntervalExtender ，DisplayLocalization モジュールが音源定位を行う．定位結果と波形から，GHDSS モジュールで音源分離を行いし，Synthesize で時間領域の音声波形に変換し，SaveWavePCM で音声波形を保存している．

サンプルの中で重要なパラメータは，TF_CONJ_FILENAME である．マイクアレイ (このサンプルの場合は Kinect) のインパルス応答から harktool3 で作成したファイルを使用する．

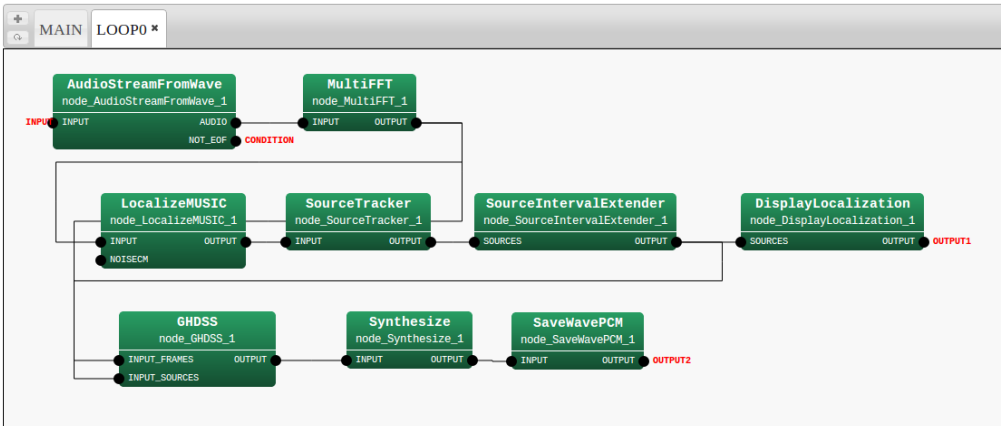


図 14.14: Sound source separation without HRLE

14.4.2 オフライン音源分離 (HRLE を使った後処理あり)

分離音声に後処理として，HRLE で雑音推定を行い，推定値に基き雑音除去処理を行うサンプルを紹介する．Separation ディレクトリに含まれる demoOfflineKinectHRLE.sh を実行する．実行後，音源分離され，後処

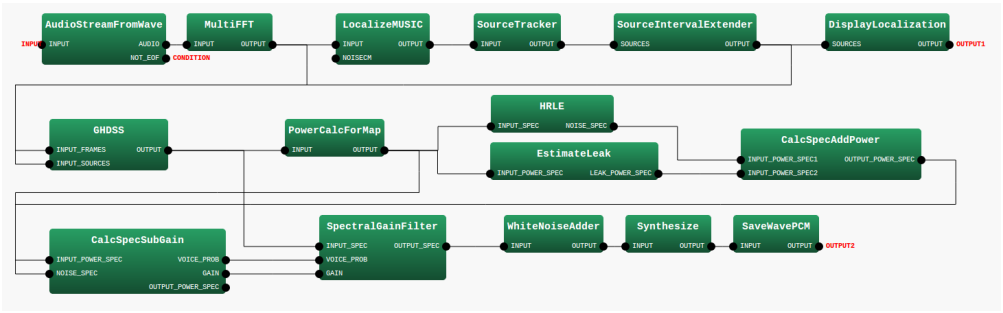


図 14.15: MAIN_LOOP (iterator)

理を行った分離音声がファイルに書き出される．

図 14.15 に示すのは，後処理を含めた demoOfflineKinectHRLE.n のネットワークである．AudioStreamFromWave モジュールでファイルから読み出した音声波形を MultiFFT で周波数領域へと変換し，GHDSS で音源分離し，後処理後，Synthesize で合成し，SaveRawPCM で音声波形を保存している．後処理は，HRLE，EstimateLeak，CalcSpecAddPower，CalcSpecSubGain，SpectralGainFilter，の組み合わせにより実現している．無指向雑音と，検出した音源のなかから非目的音からの干渉を推定し，帯域ごとにスペクトルのレベルを調整している．

14.4.3 オンライン音源分離 (HRLE を使った後処理有り・無し)

オンライン音源分離のサンプルを紹介する．ただし，このサンプルを実行するためには Kinect が必要である．

サンプルの実行には，Separation ディレクトリに含む demoOnlineKinect.sh を実行する．Kinect の ID は環境によって変化するため，事前に Kinect の ID を調べサンプルネットワークファイルに設定する．

ここまで準備ができたら，Separation ディレクトリ中の demoOnlineKinect.sh もしくは demoOnlineKinectHRLE.sh を実行する．正しく実行できた場合，適当に Kinect の周囲で発話すると，定位結果がグラフィカルに表示され，分離音がファイルに保存されてゆく．このサンプルを中止するには，Control キーと 'c' キーを同時に押す．

オフラインの音源分離サンプルは正しく実行できるのにオンラインの場合に正しく実行できない場合，AudioStreamFromMic モジュールのパラメータが不適切であることが疑われる．録音用のサンプルを見ながら確認すると良い．

14.5 音響特徴量抽出ネットワークサンプル

14.5.1 はじめに

HARK では，MSLSExtraction や MFCCExtraction で計算できる静的特徴量に加えて, 次の 4 種類の特徴量や処理の追加ができる．

1. 動的特徴量 (デルタ項): 静的特徴量の変化．表 14.14 では ΔMSLS と表記．Delta で計算する．
2. パワー: 入力音声の音量．表 14.14 では Power と表記．PowerCalcForMap で計算する．
3. デルタパワー: 入力音声の音量の変化．表 14.14 では Δ Power と表記．Delta で計算する．
4. 前処理: 高周波領域の強調 (PreEmphasis)，平均除去 など．表 14.14 では 前処理 と表記．

本節では 表 14.14 に示す 6 種類のサンプルを提供する．左列から順に, ネットワークファイル名, 生成する特徴量の種類や表記, 生成されるファイル名を示している．これらの実行は `demo.sh` に引数を与えることで行える．たとえば `demo1.n` を実行したいなら, 次のコマンドを実行すればよい．

```
> ./demo.sh 1
```

本節で提供するサンプルは, すべて 13 次元の MSLS をオフラインで生成するネットワークを元にしている．オンラインで特徴量抽出を行うには AudioStreamFromWave を AudioStreamFromMic に差し替えればよい．MFCC を特徴量に使うには, MFCCExtraction の代わりに MSLSExtraction を使えばよい．また, 次元数を変えたければ MSLSExtraction のプロパティを設定すればよい．詳しくは HARK Document の各ノードの説明を参照．

表 14.14: 音響特徴量抽出ネットワーク一覧.

ネットワーク ファイル名	特徴量 MSLS 13 次元	Δ MSLS 13 次元	Power 1 次元	ΔPower 1 次元	前処理 次元なし	対応する節	生成されるファイル
demo1.n	Yes					14.5.2	`MFBANK13_0.spec`
demo2.n	Yes	Yes				14.5.3	`MFBANK26_0.spec`
demo3.n	Yes		Yes			14.5.4	`MFBANK14_0.spec`
demo4.n	Yes	Yes	Yes	Yes		14.5.5	`MFBANK28_0.spec`
demo5.n	Yes	Yes		Yes		14.5.6	`MFBANK27_0.spec`
demo6.n	Yes	Yes		Yes	Yes	14.5.7	`MFBANK27p_0.spec`

14.5.2 MSLS

図 14.5.2 に実行例を示す．実行後， `MFBANK13_0.spec` というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 13 次元ベクトル列をフレーム数分格納している．失敗するときは，data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

```
> ./demo.sh 1
 MSLS
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network  :MAIN
```

図 14.16: 実行例

本サンプルに含まれるモジュールは，12 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 9 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.17, 図 14.18 に示す．処理の

概要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である． MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，取り込んだ音声波形は， MultiFFT によって分析され， MatrixToMap と PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている． MSLSExtraction は，MSLS 係数の他に δ MSLS 係数の格納領域をリザーブし，MSLSExtraction の FBANK_COUNT プロパティで指定した値の 2 倍のベクトルを特徴量として出力する（δ MSLS 係数の格納領域には 0 が入れられている）．そのため， MSLS 係数のみを求めるためには，ここでは不要な δ MSLS 係数領域を削除する必要がある．削除には FeatureRemover を用いている．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面方向の定位結果を与える．

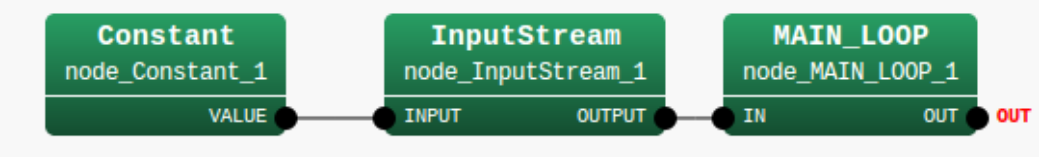


図 14.17: MAIN (subnet)

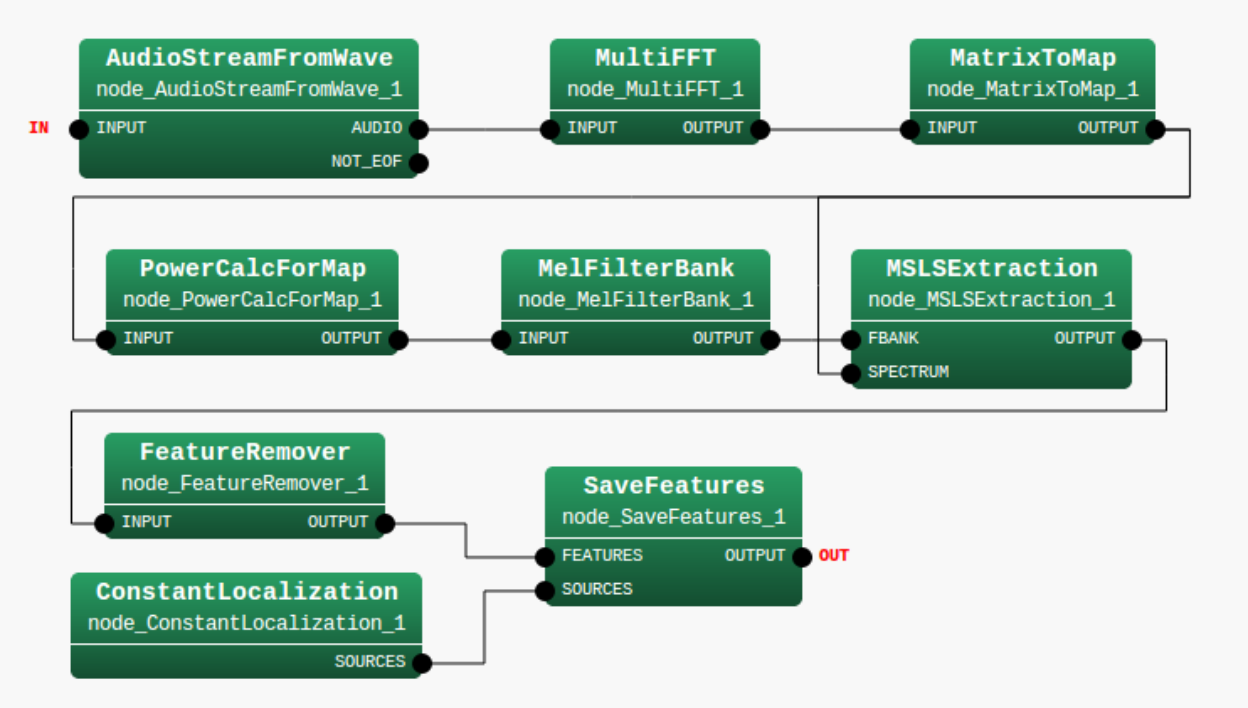


図 14.18: MAIN_LOOP (iterator)

表 14.15 に主要なパラメータを示す．重要なモジュールは，MSLSExtraction である．

14.5.3 MSLS+ΔMSLS

図 14.5.3 に実行例を示す．実行後，`MFBANK26_0.spec` というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 26 次元ベクトル列を格納している．うまく特徴抽出できないときは，data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

本サンプルに含まれるモジュールは，12 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 9 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.20,14.21 に示す．処理の概

表 14.15: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	DOWHILE	bool	(空欄)
MSLSExtraction	FBANK_COUNT	subnet_param	FBANK_COUNT
	NORMALIZE_MODE	string	Cepstral
	USE_POWER	bool	false

```
> ./demo.sh 2
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network :MAIN
```

図 14.19: 実行例

要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である． MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，取り込んだ音声波形は， MultiFFT によって分析され， MatrixToMap と PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている． MSLSExtraction は，MSLS 係数の他に δ MSLS 係数の格納領域をリザーブし，MSLSExtraction の FBANK_COUNT プロパティで指定した値の 2 倍のベクトルを特徴量として出力する．δ MSLS 係数の格納領域には 0 が入れられている．Delta により，δ MSLS 係数が計算され格納される．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面の定位結果を与える．

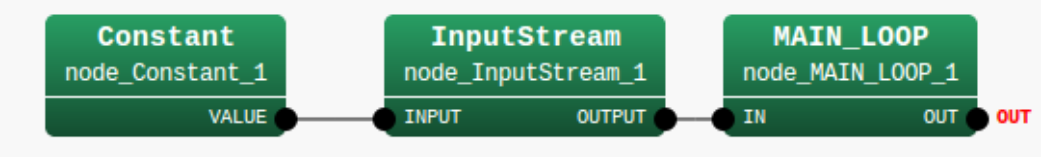


図 14.20: MAIN (subnet)

表 14.16 に主要なパラメータ表を示す.

表 14.16: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	DOWHILE	bool	(empty)
Delta	FBANK_COUNT	subnet_param	FBANK_COUNT

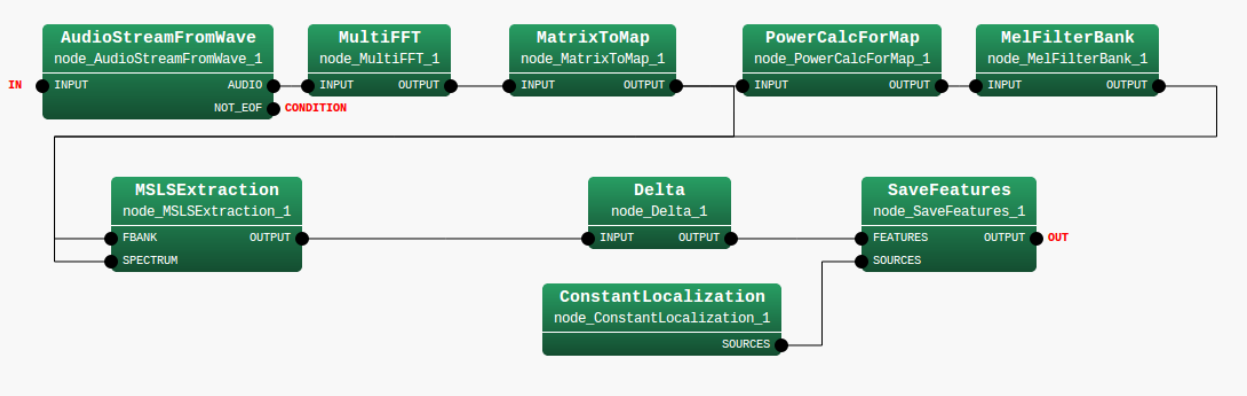


図 14.21: MAIN_LOOP (iterator)

14.5.4 MSLS+Power

図 14.5.4 に実行例を示す．実行後，MFBANK14_0.spec というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 14 次元ベクトル列を格納している．うまく特徴抽出できないときは，../data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

```
> ./demo.sh 3
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network : MAIN
```

図 14.22: 実行例

本サンプルに含まれるモジュールは，12 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 9 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.23,14.24 に示す．処理の概要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である． MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，取り込んだ音声波形は， MultiFFT によって分析され， MatrixToMap と PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている．ここでは，MSLSExtraction の USE_POWER プロパティを true にして，パワー項も同時に出力している．MSLSExtraction は，MSLS 係数の他に δ 係数の格納領域をリザーブし，ベクトルを特徴量として出力する（δ 係数の格納領域には 0 が入れられている）．USE_POWER プロパティを true にしているので，δ 係数は， δ MSLS とデルタパワー項の格納領域が確保される．従って，MSLSExtraction の FBANK_COUNT プロパティで指定した値+1 次元の 2 倍の次元数のベクトルを特徴量として出力する．必要な MSLS 係数とパワー項以外は， FeatureRemover で削除している．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面方向の定位結果を与える．

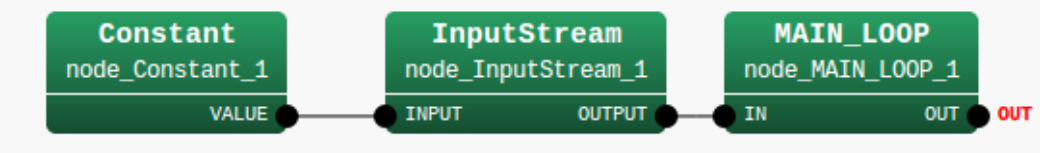


図 14.23: MAIN (subnet)

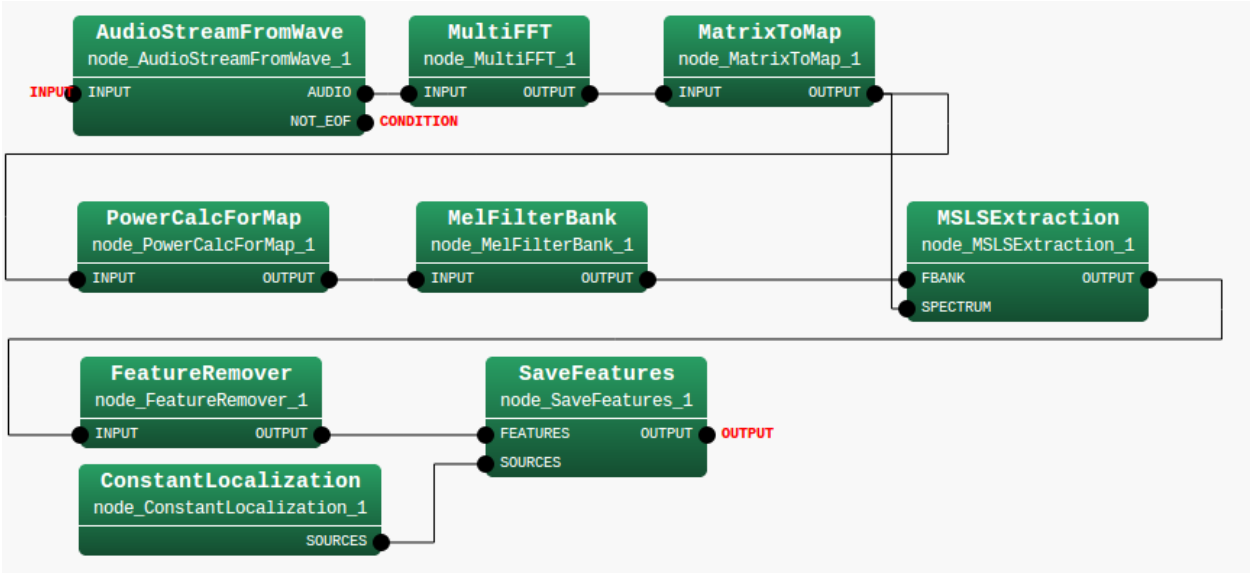


図 14.24: MAIN_LOOP (iterator)

表 14.17 に主要なパラメータを示す.

表 14.17: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	DOWHILE	bool	(empty)
MSLSExtraction	FBANK_COUNT	subnet_param	FBANK_COUNT
	NORMALIZE_MODE	string	Cepstral
	USE_POWER	bool	true

14.5.5 MSLS+ΔMSLS+Power+ΔPower

図 14.5.5 に実行例を示す．実行後，MFBANK28_0.spec というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 28 次元ベクトル列を格納している．うまく特徴抽出できないときは，data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

```
> ./demo.sh 4
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network : MAIN
```

図 14.25: 実行例

本サンプルに含まれるモジュールは，12 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 9 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.26,14.27 に示す．処理の概

要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である． MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，取り込んだ音声波形は， MultiFFT によって分析され， MatrixToMap と PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている．ここでは，MSLSExtraction の USE_POWER プロパティを true にして，パワー項も同時に出力している．MSLSExtraction は，MSLS 係数の他に δ 係数の格納領域をリザーブし，ベクトルを特徴量として出力する（δ 係数の格納領域には 0 が入れられている）．USE_POWER プロパティを true にしているので，δ 係数は， δ MSLS とデルタパワー項の格納領域が確保される．従って，MSLSExtraction の FBANK_COUNT プロパティで指定した値+1 次元の 2 倍の次元数のベクトルを特徴量として出力する．Delta により，δ MSLS 係数とデルタパワー項が計算され格納される．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面方向の定位結果を与える．

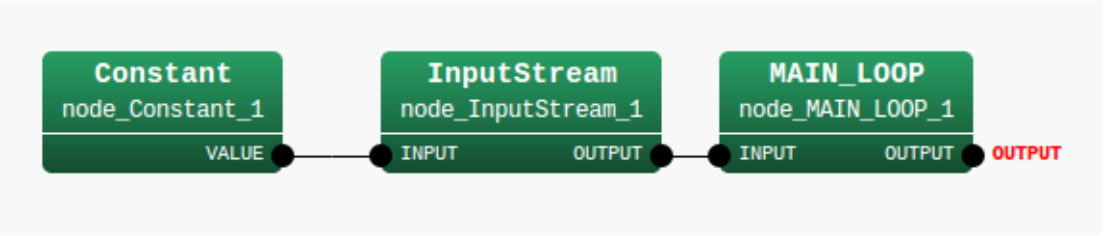


図 14.26: MAIN (subnet)

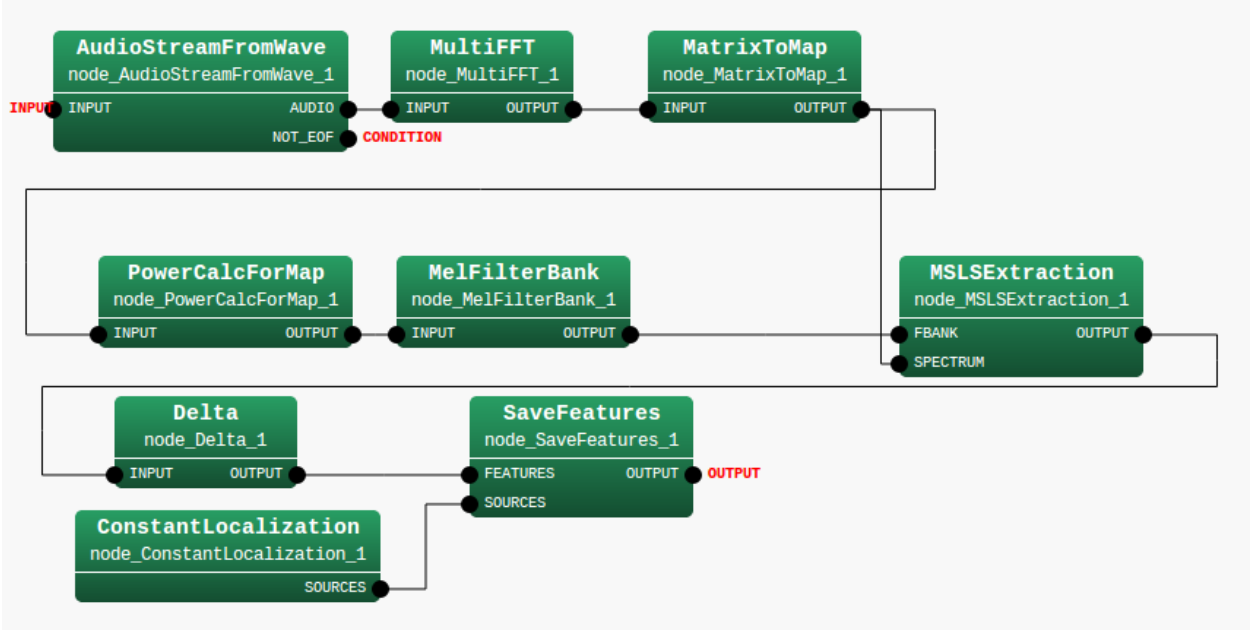


図 14.27: MAIN_LOOP (iterator)

表 14.18 が主要なパラメータである．

14.5.6 MSLS+ΔMSLS+ΔPower

図 14.5.6 に実行例を示す．実行後，MFBANK27_0.spec というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 27 次元ベクトル列を格納している．うまく特徴抽出できないときは，data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

表 14.18: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	FBANK_COUNT1	subnet_param	int :ARG6
	DOWHILE	bool	(空欄)
MSLSExtraction	FBANK_COUNT	subnet_param	FBANK_COUNT
	NORMALIZE_MODE	string	Cepstral
	USE_POWER	bool	true
Delta	FBANK_COUNT1	subnet_param	FBANK_COUNT1

```
> ./demo.sh 5
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network :MAIN
```

図 14.28: 実行例

本サンプルに含まれるモジュールは，13 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 10 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.29,14.30 に示す．処理の概要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である． MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，取り込んだ音声波形は， MultiFFT によって分析され， MatrixToMap と PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている． MSLSExtraction は，MSLS 係数の他に δ 係数の格納領域をリザーブし，ベクトルを特徴量として出力する（δ 係数の格納領域には 0 が入れられている）．ここでは MSLSExtraction の USE_POWER プロパティを true にしているので，δ 係数は， δ MSLS とデルタパワー項を含む．MSLSExtraction の FBANK_COUNT プロパティで指定した値+1 次元の 2 倍の次元数のベクトルを特徴量として出力する．Delta により，δ MSLS 係数とデルタパワー項が計算され格納される．必要な係数は MSLS 係数と δ MSLS 係数とデルタパワー項であるため，不要なパワー項を削除する必要がある．削除には FeatureRemover を用いている．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面方向の定位結果を与える．

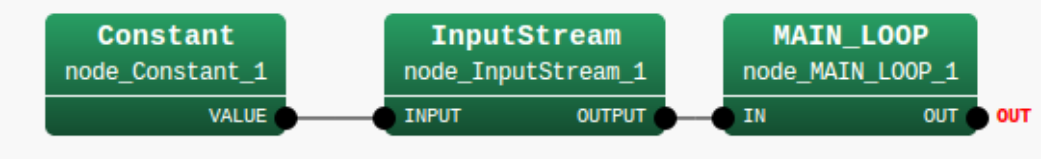


図 14.29: MAIN (subnet)

表 14.19 に主要なパラメータをまとめる．

14.5.7 MSLS+ΔMSLS+ΔPower+前処理

図 14.5.7 に実行例を示す．実行後，MFBANK27p_0.spec というファイルが生成される．このファイルは，リトルエンディアン，32 ビット浮動小数点数形式で表された 27 次元ベクトル列を格納している．うまく特徴抽

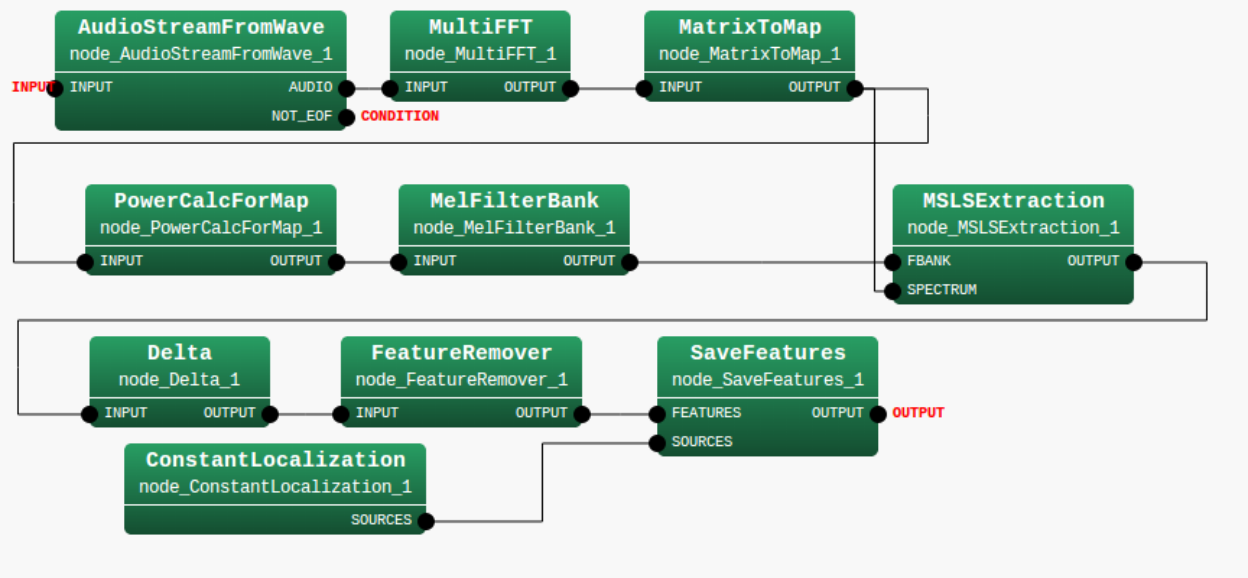


図 14.30: MAIN_LOOP (iterator)

表 14.19: パラメータ表

ノード名	パラメータ名	型	設定値
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	FBANK_COUNT1	subnet_param	int :ARG6
	DOWHILE	bool	(空欄)
MSLSExtraction	FBANK_COUNT	subnet_param	FBANK_COUNT
	NORMALIZE_MODE	string	Cepstral
	USE_POWER	bool	true
Delta	FBANK_COUNT1	subnet_param	FBANK_COUNT1
FeatureRemover	SELECTOR	Object	<Vector<float> 13>

出できないときは，data ディレクトリに f101b001.wav ファイルがあるかチェックしよう．

```
> ./demo.sh 6
 UINodeRepository::Scan()
 Scanning def /usr/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network : MAIN
```

図 14.31: 実行例

本サンプルに含まれるモジュールは，17 個である．MAIN (subnet) に 3 個 MAIN_LOOP (iterator) に 14 個のモジュールがある．MAIN (subnet) と MAIN_LOOP (iterator) を図 14.32,14.33 に示す．処理の概要は，AudioStreamFromWave モジュールで取り込んだ音声波形を MSLSExtraction で音響特徴量を計算し，SaveFeatures でファイルに書き出す単純なネットワーク構成である．プリエンファシスは，時間領域でかけるため，音声波形を MultiFFT で分析した後，MatrixToMap による型変換を経て一旦 Synthesize によって信号波形を合成して

いる．合成波形に PreEmphasis でプリエンファシスをかけ，改めて MultiFFT で分析し， PowerCalcForMap による型変換を経て，MSLSExtraction に送られている．MSLSExtraction は， MSLS の計算にメルフィルタバンクの出力とパワースペクトルを必要とするため，MultiFFT によって分析され，PowerCalcForMap によってデータ型を変換した後に MelFilterBank によりメルフィルタバンクの出力を求める処理が入っている．MSLSExtraction は，MSLS 係数の他に δ 係数の格納領域をリザーブし，ベクトルを特徴量として出力する（δ 係数の格納領域には 0 が入れられている）．ここでは MSLSExtraction の USE_POWER プロパティを `true` にしているので，δ 係数は， δ MSLS とデルタパワー項を含む．MSLSExtraction の FBANK_COUNT プロパティで指定した値+1 次元の 2 倍の次元数のベクトルを特徴量として出力する．平均除去処理を行う SpectralMeanNormalization を経て Delta により，δ MSLS 係数とデルタパワー項が計算され格納される．必要な係数は MSLS 係数と δ MSLS 係数とデルタパワー項であるため，不要なパワー項を削除する必要がある．削除には FeatureRemover を用いている．SaveFeatures は，入力 FEATURE を保存する．入力 SOURCES には ConstantLocalization で生成した正面方向の定位を与える．

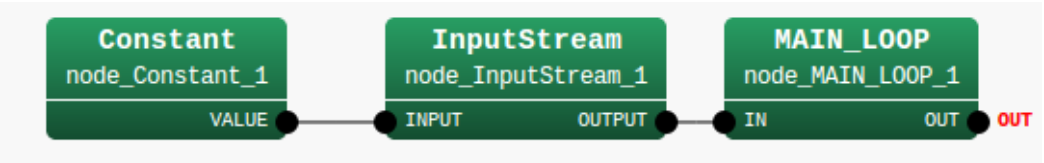


図 14.32: MAIN (subnet)

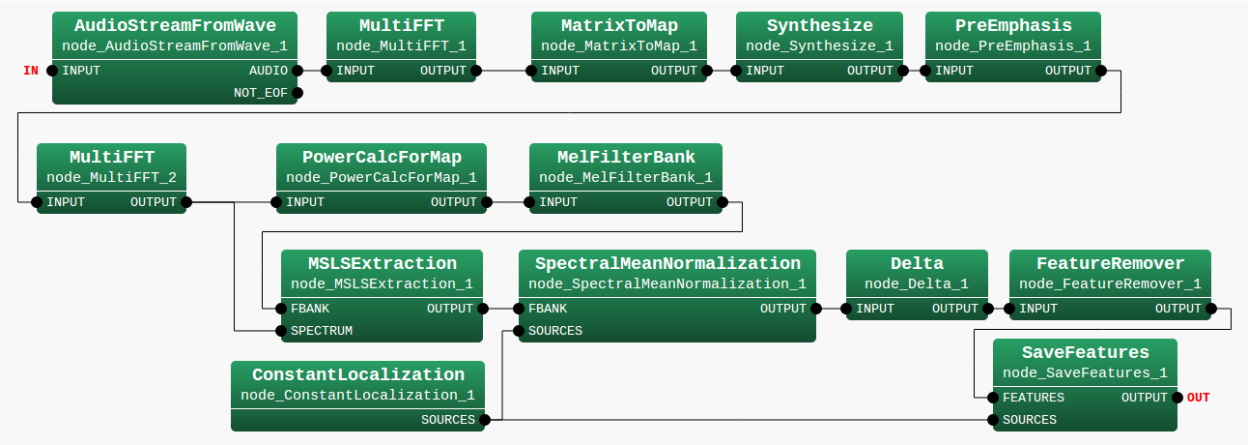


図 14.33: MAIN_LOOP (iterator)

表 14.20 にネットワークのパラメータの設定値をまとめる．重要なモジュールは，PreEmphasis と MSLSExtraction， SpectralMeanNormalization，Delta， FeatureRemover である．詳細は HARK ドキュメントを参照．

表 14.20: パラメータ表

ノード名	パラメータ名	型	値
Constant	VALUE	subnet_param	int :ARG1
MAIN_LOOP	LENGTH	subnet_param	int :ARG2
	ADVANCE	subnet_param	int :ARG3
	SAMPLING_RATE	subnet_param	int :ARG4
	FBANK_COUNT	subnet_param	int :ARG5
	FBANK_COUNT1	subnet_param	int :ARG6
	DOWHILE	bool	(空欄)
PreEmphasis	LENGTH	subnet_param	LENGTH
	SAMPLING_RATE	subnet_param	SAMPLING_RATE
	PREEMCOEFF	float	0.97
	INPUT_TYPE	string	WAV
MSLSExtraction	FBANK_COUNT	subnet_param	FBANK_COUNT
	NORMALIZE_MODE	string	Cepstral
	USE_POWER	bool	true
SpectralMeanNormalization	FBANK_COUNT1	subnet_param	FBANK_COUNT1
Delta	FBANK_COUNT1	subnet_param	FBANK_COUNT1
FeatureRemover	SELECTOR	Object	<Vector<float> 13>

14.6 音声認識ネットワークサンプル

同時発話が録音された音声ファイルを音源分離し，音声認識し，音声認識率を評価するサンプルを紹介する．サンプルファイルはオフライン処理であるが，AudioStreamFromWave を AudioStreamFromMic に入れ替えれば，オンラインの同時発話認識にも使える．すべてのファイルは Recognition ディレクトリにある．各ファイルの説明は表 14.21 を参照．以下では，サンプルの実行方法を音声認識の実行，認識率の評価の順に説明していく．

表 14.21: ファイルリスト

分類	ファイル名	説明
データ	MultiSpeech.wav	本サンプルで使用する同時発話ファイル
JuliusMFT	julius.jconf	JuliusMFT 設定ファイル
	hmmdefs.gz	音響モデル
	allTriphones	認識可能なトライフォン
	order.*	言語モデル
HARK	MultiSpeechRecog.n	音源分離・定位・特徴量抽出ネットワーク
	Recognition.sh	ネットワークを実行するスクリプト
	loc_tf.dat	定位用伝達関数ファイル
	sep_tf.tff	分離用伝達関数ファイル
	wav/	分離音保存用ディレクトリ
評価	score.py	評価スクリプト
	transcription_list*.txt	方向別正解データ

14.6.1 音声認識の実行

まずは，ターミナルを二つ用意して，片方で JuliusMFT による音声認識を実行し，もう一方で HARK による音源分離を実行しよう．実行するコマンドと実行例を，図 14.34 と図 14.35 に示す．音声認識の準備が終わっていないと音源分離結果が捨てられてしまうので，先に図 14.34 を実行し，その次に図 14.35 を実行する必要があることに注意．実行が終わると，wav ディレクトリ以下に大量の wav ファイルが生成され，result.txt に JuliusMFT のログが保存されているはずなので確認しよう．

```
> julius_mft -C julius.jconf 2>&1 | tee result.txt
STAT: include config: julius.jconf
STAT: loading plugins at "/usr/lib/julius_plugin":
STAT: file: calcmix_heu.jpi  #0 [Gaussian calculation plugin for Julius.
                                                              (ADD_MASK_TO_HEU)]
     (中略)
---------------------- System Information end -----------------------------
```

図 14.34: JuliusMFT 実行例

wav ファイル達は処理の中間ファイルである．ファイル名の最初の文字列は処理の段階を，続く数字は定位された通し番号を表す．GHDSS から始まるは音源分離結果，NR から始まるファイルは音源分離結果に含まれる漏れノイズを後処理で減らした結果，そして WN から始まるファイルはそれに白色雑音を付加したものである．WN から始まるファイルの MSLS 特徴量が抽出されて，JuliusMFT に送られる．標準的な wav ファイルなので，適当なオーディオプレイヤーで聞いてみるとよいだろう．

result.txt は文字コードが EUC-JP の 未加工の音声認識ログである．pass1_best: から始まる行が音声認識結果なので，次のコマンドを実行すると，認識結果の料理名がたくさん表示されるはずだ．

```
grep pass1_best: result.txt | nkf
```

トラブルシューティング

wav ファイルが生成されていないときは HARK が正しく動いていない．まずはファイルの確認をしよう．表 14.21 をもう一度確認して，ファイルがすべて揃っているかを確認しよう．wav ディレクトリに書き込み権

```
> Recognition.sh
 UINodeRepository::Scan()
 Scanning def /usr/local/lib/flowdesigner/toolbox
 done loading def files
 loading XML document from memory
 done!
 Building network  :MAIN
 TF = 1,INITW = 0,FixedNoise = 0
 SSMethod = 2LC_CONST = 0LC_Method = 1
 reading A matrix
 Try to read loc_tf.dat as HARK format.
 72 directions, 1 ranges, 8 microphones, 512 points
 done
 initialize
 Source 0 is created.
 Source 1 is created.
    (以下略)
```

図 14.35: HARK 実行例

限があるかも確認しよう．もしあれば，次は HARK のインストールが正しく行われているかを，レシピ: 3.1 うまくインストールできないを見ながら確認しよう．

result.txt に音声認識結果が無い場合は，JuliusMFT が正しく動いていない．まずは JuliusMFT がインストールされているかを確認しよう．`julius_mft` をコマンドラインで実行し， `command not found` と表示されたらインストールされていない．次に，表 14.21 を確認して，ファイルがすべて揃っているかを確認しよう．最後に，図 14.34 のコマンドを間違えずに実行しているかを再度確認しよう．

いずれの場合も，ログにエラーの原因が書かれているのでよく確認しよう．

14.6.2 音声認識率の評価

認識結果が出たら，次は評価用スクリプトは `score.py` で音声認識率を評価しよう．方向ごとに評価するために，次の 3 つのコマンドを順番に実行していこう．

```
> python score.py result.txt transcription_list1.txt 60 10
> python score.py result.txt transcription_list3.txt 0 10
> python score.py result.txt transcription_list2.txt -60 10
```

引数の意味は，前から順に音声認識結果のログ，正解データ，正解データの音声到来方向，音声到来方向の許容誤差である．たとえば 1 行目なら，50 度から 70 度から到来した音源を評価対象として，transcription_list1.txt にある正解データと比較する．

実行すると，図 14.36 のような表示が得られるはずだ．各行は左から順に認識が成功したかどうか，認識結果，正解データを表す．最後の行は，全発話中何発話が成功したかと，音声認識率を表す．この場合は， 20 発話中 17 発話の認識に成功し，認識率は 85% である．

いずれの認識率も 70% から 90% のはずなので，もし極端に低い場合は，正解データのファイル名と，方向の指定が正しいかを確かめよう．それも正しければ，音源分離・認識の失敗が考えられるので，wav/ ディレクトリのファイルを聞いてみたり，3 章のレシピを参照しよう．

Result	Recognition	Correct
Success	”からあげ定食”	”からあげ定食”
Success	”とんかつ定食”	”とんかつ定食”
Success	”焼き魚定食”	”焼き魚定食”
Success	”サイコロステーキ”	”サイコロステーキ”
Fail	”からあげ定食”	”松阪牛ステーキ”
	(中略)	
Success	”アメリカンコーヒー”	”アメリカンコーヒー”

17 / 20 (85.0 %)

図 14.36: 認識結果