pyFASST Documentation

Release 0.1

Jean-Louis Durrieu

August 01, 2013

CONTENTS

CHAPTER

ONE

CONTENTS:

1.1 pyFASST

contributors Jean-Louis Durrieu

web https://git.epfl.ch/repo/pyfasst/ https://github.com/wslihgt/pyfasst

A Python implementation to the Flexible Audio Source Separation Toolbox

1.1.1 Abstract

This toolbox is meant to allow to use the framework FASST and extend it within a python program. It is primarily a re-write in Python of the original Matlab (C) version. The object programming framework allows to extend and create

1.1.2 Using the Python package

Dependencies

Most of the code is written in Python, but occasionally, there may be some C source code, requiring either Cython or SWIG for compiling. In general, to run this code, the required components are:

- Matplotlib http://matplotlib.sourceforge.net
- Numpy http://numpy.scipy.org
- Scipy http://www.scipy.org
- setuptool https://pypi.python.org/pypi/setuptools

Install

In addition to the aforementioned packages, installing this package requires to compile the tracking part, in `pyfasst.SeparateLeadStereo.tracking`.

Examples

- Using the provided audio model classes
- Creating a new audio model class

1.1.3 Algorithms

The FASST framework is described in [Ozerov2012]. We have implemented this Python version mostly thanks to the provided Matlab (C) code available at http://bass-db.gforge.inria.fr/fasst/.

For initialization purposes, several side algorithms and systems have also been implemented:

- SIMM model (Smooth Instantaneous Mixture Model) from [Durrieu2010] and [Durrieu2011]: allows to analyze, detect and separate the lead instrument from a polyphonic audio (musical) mixture. Note: the original purpose of this implementation was to provide a sensible way of using information from the SIMM model into the more general multi-channel audio source separation model provided, for instance, by FASST.
- DEMIX algorithm (Direction Estimation of Mixing matrIX) [Arberet2010] for spatial mixing parameter initialization.

1.1.4 References

1.2 Reference

1.2.1 audioModel

AudioModel:

Description

FASST (Flexible Audio Source Separation Toolbox) class subclass it to obtain your own flavoured source separation model!

Usage

TBD

Reference

..[Ozerov2012] A. Ozerov, E. Vincent and F. Bimbot "A General Flexible Framework for the Handling of Prior Information in Audio Source Separation," IEEE Transactions on Audio, Speech and Signal Processing 20(4), pp. 1118-1133 (2012) Available: Archive on HAL

Adapted from the Matlab toolbox available at http://bass-db.gforge.inria.fr/fasst/

Copyright (TBD)

Jean-Louis Durrieu, EPFL-SSTI-IEL-LTS5

`jean DASH louis AT durrieu DOT ch`

2012-2013 http://www.durrieu.ch

Reference

class `pyfasst.audioModel.`**`FASST`**(*audio, transf='stft', wlen=2048, hopsize=512, iter_num=50, sim_ann_opt='ann', ann_PSD_lim=[None, None], verbose=0, nmfUpdateCoeff=1.0, tffmin=25, tffmax=18000, tfWinFunc=None, tfbpo=48, lambdaCorr=0.0*)

FASST audio model, from [Ozerov2012]

`GEM_iteration`()

GEM iteration

`comp_spat_cmps_powers`(*spat_comp_ind, spec_comp_ind=[ ], factor_ind=[ ]*)

Compute the sum of the spectral powers corresponding to the spatial components as provided in the list *spat_comp_ind*

NB: because this does not take into account the mixing process, the resulting power does not, in general, correspond to the the observed signal's parameterized spectral power.

`comp_spat_comp_power`(*spat_comp_ind, spec_comp_ind=[ ], factor_ind=[ ]*)

Matlab FASST Toolbox help:

```
% V = comp_spat_comp_power(mix_str, spat_comp_ind,
%                          spec_comp_ind, factor_ind);
%
% compute spatial component power
%
%
% input
% -----
%
% mix_str           : mixture structure
% spat_comp_ind     : spatial component index
% spec_comp_ind     : (opt) factor index (def = [], use all components)
% factor_ind         : (opt) factor index (def = [], use all factors)
%
%
% output
% ------
%
% V                 : (F x N) spatial component power
```

`comp_transf_Cx`()

Computes the signal representation, according to the provided signal_representation flag

`compute_Wiener_gain_2d`(*sigma_comp_diag, sigma_comp_off, inv_sigma_mix_diag, inv_sigma_mix_off, timeInvariant=False*)

Matlab FASST Toolbox help:

```
% WG = comp_WG_spat_comps(mix_str);
%
% compute Wiener gains for spatial components
%
%
% input
% -----
%
% mix_str           : input mix structure
%
%
% output
% ------
%
% WG                : Wiener gains [M x M x F x N x K_spat]
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Flexible Audio Source Separation Toolbox (FASST), Version 1.0
%
% Copyright 2011 Alexey Ozerov, Emmanuel Vincent and Frederic Bimbot
% (alexey.ozerov -at- inria.fr, emmanuel.vincent -at- inria.fr,
%  frederic.bimbot -at- irisa.fr)
%
% This software is distributed under the terms of the GNU Public
% License version 3 (http://www.gnu.org/licenses/gpl.txt)
%
% If you use this code please cite this research report
%
% A. Ozerov, E. Vincent and F. Bimbot
% "A General Flexible Framework for the Handling of Prior
% Information in Audio Source Separation,"
% IEEE Transactions on Audio, Speech and Signal Processing 20(4),
% pp. 1118-1133 (2012).
% Available: http://hal.inria.fr/hal-00626962/
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

`compute_inv_sigma_mix_2d`(*sigma_comps_diag*, *sigma_comps_off*)
only for nb channels = 2

sigma_comps_diag ncomp x nchan x nfreq x nframes

`compute_sigma_comp_2d`(*spat_ind*, *spec_comp_ind*)
only for stereo case self.audioObject.channels==2

`compute_suff_stat`(*spat_comp_powers*, *mix_matrix*)
Outputs:

hat_Rxx hat_Rxs hat_Rss hat_Ws loglik

`estim_param_a_post_model`()
Estimates the *a posteriori* model for the provided audio signal. In particular, this runs self.iter_num times the Generalized Expectation-Maximisation algorithm to update the various parameters of the model, so as to maximize the likelihood of the data given these parameters.

From these parameters, the posterior expectation of the "hidden" or latent variables (here the spatial and spectral components) can be computed, leading to the estimation of the separated underlying sources.

Consider using `self.separate_spat_comps` or `self.separate_spatial_filter_comp` to obtain the separated time series, once the parameters have been estimated.

`gcc_phat_tdoa_2d`()
Using the cross-spectrum in self.Cx[1] to estimate the time difference of arrival detection function (the Generalized Cross- Correllation GCC), with the phase transform (GCC-PHAT) weighing function for the cross-spectrum.

`initializeConvParams`(*initMethod='demix'*)
setting the spatial parameters

`initialize_all_spec_comps_with_NMF`(*sameInitAll=False*, ***kwargs*)
Computes an NMF on the one-channel mix (averaging diagonal of self.Cx, which are the power spectra of the corresponding channel)

$$C_x \approx WH$$

then, for all spec_comp in self.spec_comps, we set:

```
spec_comp['FB'] = W
spec_comp['TW'] = H
```

initialize_all_spec_comps_with_NMF_indiv(*niter=10*, *updateFreqBasis=True*, *updateTimeWeight=True*, ***kwargs*)

initialize the spectral components with an NMF decomposition, with individual decomposition of the monophonic signal TF representation.

TODO make keepFBind and keepTWind, in order to provide finer control on which indices are updated. Also requires a modified NMF decomposition function.

initialize_all_spec_comps_with_NMF_same(*niter=10*, ***kwargs*)

Initialize all the components with the same amplitude and spectral matrices W and H.

mvdr_2d(*theta*, *distanceInterMic=0.3*)

mvdr_2d(self, theta, # in radians distanceInterMic=.3, # in meters)

MVDR minimum variance distortion-less response spatial filter, for a given angle theta and given distance between the mics.

self.Cx is supposed to provide the necessary covariance matrix, for the "Capon" filter.

renormalize_parameters()

Re-normalize the components

retrieve_subsrc_params()

Computes the various quantities necessary for the estimation of the main parameters:

Outputs

- **spat_comp_powers** (`total_spat_rank x nbFreqsSigRepr x nbFramesSigRepr`) ndarray the spatial component power spectra. Note that total_spat_rank is the sum of all the spatial ranks for all the sources.
- **mix_matrix** (total_spat_rank x nchannels x nbFreqsSigRepr) ndarray the mixing matrices for each source
- **rank_part_ind** dictionary: each key is one source, and the values are the indices in `spat_comp_powers` and `mix_matrix` that correspond to that source. If the spatial rank of source `j` is 2, then its spectra will appear twice in `spat_comp_powers`, with mixing parameters (potentially different one from the other) appearing in two sub-matrices of `mix_matrix`.

separate_comps(*dir_results=None*, *spec_comp_ind=None*, *suffix=None*)

Separate the sources as defined by the spectral components provided in spec_comp_ind.

This function differs from separate_spat_comps in the way that it does not assume the sources are defined by their spatial positions.

Note: Trying to bring into one method ozerov's separate_spec_comps and separate_spat_comps

separate_spat_comps(*dir_results=None*, *suffix=None*)

This separates the sources for each spatial component.

separate_spatial_filter_comp(*dir_results=None*, *suffix=None*)

Separates the sources using only the estimated spatial filter (i.e. the mixing parameters in self.spat_comps[j]['params'])

In particular, we consider here the corresponding MVDR filter, as exposed in [Maazaoui2011].

per channel, the filter steering vector, source p:

$$b(f,p) = \frac{R_{aa,f}^{-1} a(f,p)}{a^H(f,p) R_{aa,f}^{-1} a(f,p)}$$

with

$$R_{aa,f} = \sum_q a(f,q) a^H(f,q)$$

It corresponds also to the given model in FASST, assuming that all the spectral powers are equal across all sources. Here, by computing the Wiener Gain WG to get the images, we actually have

$$b(f,p) a(f,p)^H$$

and the denominator therefore is the trace of the "numerator".

setComponentParameter(*newValue*, *spec_ind*, *fact_ind=0*, *partLabel='FB'*, *prior='free'*, *keepDimensions=True*)

A helper function to set a self.spec_comp[spec_ind]['factor'][fact_ind][partLabel] to the given value.

TODO 20130522 finish this function to make it general purpose...

update_mix_matrix(*hat_Rxs*, *hat_Rss*, *mix_matrix*, *rank_part_ind*)

update_spectral_components(*hat_W*)

Update the spectral components, with hat_W as the expected value of power

class pyfasst.audioModel.**MultiChanNMFConv**(*audio*, *nbComps=3*, *nbNMFComps=4*, *spatial_rank=2*, ***kwargs*)

Takes the multichannel NMF instantaneous class, and makes it convolutive!

class pyfasst.audioModel.**MultiChanNMFInst_FASST**(*audio*, *nbComps=3*, *nbNMFComps=4*, *spatial_rank=2*, ***kwargs*)

MultiChanNMFInst_FASST

sub-classes FASST

This class implements the Multi-channel Non-Negative Matrix Factorisation (NMF)

setSpecCompFB(*compNb*, *FB*, *FB_frdm_prior='fixed'*)

SetSpecCompFB

sets the spectral component's frequency basis.

class pyfasst.audioModel.**multiChanSourceF0Filter**(*audio, nbComps=3, nbNMFResComps=1, nbFilterComps=20, nbFilterWeigs=[4], minF0=39, maxF0=2000, minF0search=80, maxF0search=800, stepnoteF0=16, chirpPerF0=1, spatial_rank=1, sparsity=None, **kwargs*)

multi channel source/filter model nbcomps components, nbcomps-1 SF models, 1 residual component

estim_param_a_post_model()

Estimation of model parameters, using the sparsity constraints.

initSpecCompsWithLabelAndFiles(*instrus=*[], *instru2modelfile={}*, *freqBasisAdaptive='fixed'*)

Initialize the spectral components with the instrument labels as well as with the components stored in the provided dictionary in *instru2modelfile*

instrus **is a list with labels:**

***'SourceFilter'*:** keep the intialized source filter model

'Free_<nb_comp>': initialize the model with an adaptable spectral component using *nb_comp* elements in the NMF frequency basis

<key_in_instru2modelfile>: initialize with the :py:class:GSMM available and stored in the archive npz with filename *instru2modelfile[key_in_instru2modelfile]*

NB: needs the gmm-gsmm module to be installed and in the pythonpath

`initializeFreeMats`(*niter=10*)
initialize free matrices, with NMF decomposition

`makeItConvolutive`()
Takes the spatial parameters and sets them to a convolutive mixture, in case the parameter has not yet been changed to 'conv' mode.

`reweigh_sparsity_constraint`(*sigma*)

`setSpecCompFB`(*compNb*, *FB*, *FB_frdm_prior='fixed'*)
SetSpecCompFB

sets the spectral component's frequency basis.

1.2.2 demix

DEMIX Python/NumPy implementation

Description

DEMIX is an algorithm that counts the number of sources, based on their spatial cues, and returns the estimated parameters, which are related to the relative amplitudes between the channels, as well as the relative time shifts. The full description is given in:

```
Arberet, S.; Gribonval, R. & Bimbot, F.
    A Robust Method to Count and Locate Audio Sources in
    a Multichannel Underdetermined Mixture
IEEE Transactions on Signal Processing, 2010, 58, 121 - 133
```

This implementation is based on the MATLAB Toolbox provided by the authors of the above article.

Additionally, this implementation further allows time-frequency representations other than the short-term Fourier transform (STFT).

Copyright

Jean-Louis Durrieu, EPFL-STI-IEL-LTS5

jean DASH louis AT durrieu DOT ch

2012-2013

Reference

class `pyfasst.demixTF.`**`DEMIX`**(*audio*, *nsources=2*, *wlen=2048*, *hopsize=1024*, *neighbors=20*, *verbose=0*, *maxclusters=100*, *tfrepresentation='stft'*, *tffmin=25*, *tffmax=18000*, *tfbpo=48*, *winFunc=<function sqrt_blackmanharris at 0x10262b488>*)
DEMIX algorithm, for 2 channels.

adaptive_thresholding_clusters()
compute for each cluster in self.clusters a threshold depending on the other clusters, in order to keep only those points in cluster that are close to the actual centroid, but not close to centroids of other clusters.

The returned clusters are the original clusters thresholded.

comp_clusters(*threshold=0.8*)
Computes the time-frequency clusters, along with their centroids, which contain the parameters of the mixing process - namely *theta*, which parameterizes the relative amplitude, and *delta*, which is homogeneous to a delay in samples between the two channels.

comp_parameters()

comp_pcafeatures()
Compute the PCA features

comp_sig_repr()
Computes the signal representation, stft

compute_temporal(*ind_cluster_pts*, *zoom*)
This computes the inverse Fourier transform of the estimated Steering Vectors, weighed by their inverse variance

The result is a detection function that provides peaks at the most likely delta - the delay in samples.

create_exclusive_clusters()
reconfigures the cluster indices in self.clusters such that all the Time-Freq points that appear in more than one cluster are dismissed from all computations

estimDAOBound(*confidence*, *confidenceVal=None*)
computes the max distance between centroid and points

getTFPointsNearDelta(*centroid*)
returns a TF mask which is True if their corresponding value of *delta* is close enough to the delta from the *centroid*.

getTFpointsNearTheta_OneScale(*centroid_tfpoint*)
returns the TF points whose theta is close to that of the centroid, among the points considered in index_pts_to_classify

TODO: make the function for different scales, as in matlab toolbox

get_centroid_distance()
distance between the centroids

identify_deltaT(*ind_cluster_pts*, *centroid*, *threshold=0.8*)
returns the delay maxDelta in samples that corresponds to the largest peak of the cluster defined by the provided cluster index

reestimate_clusterCentroids()
reestimate cluster centroids

considering all the cluster masks, reestimate the centroids, discarding the clusters for which there was no well-defined delta.

refine_clusters()
Refining the clusters in order to verify that they are possible. Additionally, if self.nsources is defined, this method only keeps the required number. Otherwise, it is decided by choosing the most likely centroids.

remove_empty_clusters()
DJL: this did never happen in DEMIX Matlab version, have to contact authors for explanations...

spatial_filtering()

using optimal spatial filters to obtain separated signals

this is a beamformer implementation. MVDR or assuming the sources are normal, independent and with same variance (not sure whether this does not mean that we can't separate them...)

From:

```
Maazaoui, M.; Grenier, Y. & Abed-Meraim, K.
``Blind Source Separation for Robot Audition using
Fixed Beamforming with HRTFs'',
in proc. of INTERSPEECH, 2011.
```

per channel, the filter steering vector, source p:

$$b(f,p) = \frac{R_{aa,f}^{-1} a(f,p)}{a^H(f,p) R_{aa,f}^{-1} a(f,p)}$$

steeringVectorsFromCentroids()

Generates the steering vectors a(p,f,c) for source p, (reduced) freq f and channel c.

$$a[p,f,0] = \cos(\theta_p)$$
$$a[p,f,1] = \sin(\theta_p) \exp(-2j\pi f \delta_p)$$

pyfasst.demixTF.**confidenceFromVar**(*variance*, *neighbors*)

Computes the confidence, in dB, for a given number of neighbours and a variance.

pyfasst.demixTF.**get_indices_peak**(*sequence*, *ind_max*, *threshold=0.8*)

returns the indices of the peak around ind_max, with values down to `threshold * sequence[ind_max]`

1.2.3 SeparateLeadStereo

SeparateLeadStereo, with Time-Frequency choice

Provides a class (`SeparateLeadProcess`) within which several processings can be run on an audio file, in order to extract the lead instrument/main voice from a (stereophonic) audio mixture.

copyright (C) 2011 - 2013 Jean-Louis Durrieu

class `pyfasst.SeparateLeadStereo.SeparateLeadStereoTF.`**`SeparateLeadProcess`**(*inputAudioFilename*, *windowSize=0.0464*, *hopsize=None*, *NFT=None*, *nbIter=10*, *numCompAccomp=40*, *minF0=39*, *maxF0=2000*, *stepNotes=16*, *chirpPerF0=1*, *K_numFilters=4*, *P_numAtomFilters=30*, *imageCanvas=None*, *wavCanvas=None*, *progressBar=None*, *verbose=True*, *outputDirSuffix='/'*, *minF0search=None*, *maxF0search=None*, *tfrepresentation='stft'*, *cqtfmax=4000*, *cqtfmin=50*, *cqtbins=48*, *cqtWinFunc=<function sqrt_blackmanharris at 0x10264d578>*, *cqtAtomHopFactor=0.25*, *initHF00='random'*, *freeMemory=True*)

SeparateLeadProcess

class which implements the source separation algorithm, separating the 'lead' voice from the 'accompaniment'. It can deal automatically with the task (the 'lead' voice becomes the most energetic one), or can be manually told what the 'lead' is (through the melody line).

Attributes

dataType [dtype] this is the input data type (usually the same as the audio encoding)

displayEvolution [boolean] display the evolution of the arrays (notably HF0)

F, N [integer, integer]

F the number of frequency bins in the time-frequency representation (this is half the Fourier bins, + 1)

N the number of analysis input frames

files : dictionary containing the filenames of the output files for the separated signals, with the following keys (after initialization)

'inputAudioFilename' : input filename

'mus_output_file' : output filename for the estimated 'accompaniment', appending '_acc.wav' to the radical.

'outputDirSuffix' : the subfolder name to be appended to the path of the directory of the input file, the output files will be written in that subfolder

'outputDir' : the full path of the output files directory

'pathBaseName' : base name for the output files (full path + radical for all output files)

'pitch_output_file' : output filename for the estimated melody line appending '_pitches.txt' to the radical.

'voc_output_file' : output filename for the estimated 'lead instrument', appending '_voc.wav' to the radical.

Additionally, the estimated 'accompaniment' and 'lead' with unvoiced parts estimation are written to the corresponding filename without these unvoiced parts, to which '_VUIMM.wav' is appended.

imageCanvas [instance from MplCanvas or MplCanvas3Axes] canvas used to draw the image of HF0

scaleData [double] maximum value of the input data array. With scipy.io.wavfile, the data array type is integer, and does not fit well with the algorithm, so we need this scaleData parameter to navigate back and forth between the double and integer representation.

scopeAllowedHF0 [double] scope of allowed F0s around the estimated/given melody line

stftParams : dictionary with the parameters for the time-frequency representation (Short-Time Fourier Transform - STFT), with the keys:

'hopsize' : the step, in number of samples, between analysis frames for the STFT

'NFT' : the number of Fourier bins on which the Fourier transforms are computed.

'windowSizeInSamples' : analysis frame length, in samples

SIMMParams : dictionary with the parameters of the SIMM model (Smoothed Instantaneous Mixture Model [DRDF2010]), with following keys:

'alphaL', 'alphaR' [double] stereo model, panoramic parameters for the lead part

'betaL', 'betaR' [(R,) ndarray] stereo model, panoramic parameters for each of the component of the accompaniment part.

'chirpPerF0' [integer] number of F0s between two 'stable' F0s, modelled as chirps.

'F0Table' [(NF0,) ndarray] frequency in Hz for each of the F0s appearing in WF0

'HF0' [(NF0*chirpPerF0, N) ndarray, *estimated*] amplitude array corresponding to the different F0s (this is what you want if you want the visualisation representation of the pitch saliances).

'HF00' [(NF0*chirpPerF0, N) ndarray, *estimated*] amplitude array HF0, after being zeroed everywhere outside the given scope from the estimated melody

'HGAMMA' [(P, K) ndarray, *estimated*] amplitude array corresponding to the different smooth shapes, decomposition of the filters on the smooth shapes in WGAMMA

'HM' [(R, N) ndarray, *estimated*] amplitude array corresponding to the decomposition of the accompaniment on the spectral shapes in WM

'HPHI' [(K, N) ndarray, *estimated*] amplitude array corresponding to the decomposition of the filter part on the filter spectral shapes in WPHI, defined as np.dot(WGAMMA, HGAMMA)

'K' [integer] number of filters for the filter part decomposition

'maxF0' [double] the highest F0 candidate

'minF0' [double] the lowest F0 candidate

'NF0' [integer] number of F0s in total

'niter' [integer] number of iterations for the estimation algorithm

'P' [integer] number of smooth spectral shapes for the filter part (in WGAMMA)

'R' [integer] number of spectral shapes for the accompaniment part (in WM)

'stepNotes' [integer] number of F0s between two semitones

'WF0' [(F, NF0*chirpPerF0) ndarray, *fixed*] 'dictionary' of harmonic spectral shapes for the F0 candidates generated thanks to the KLGLOTT88 model [DRDF2010]

'WGAMMA' [(F, P) ndarray, *fixed*] 'dictionary' of smooth spectral shapes for the filter part

'WM' [(F, R) ndarray, *estimated*] array of spectral shapes that are directly *estimated* on the signal

verbose [boolean] if True, the program writes some information about what is happening

wavCanvas [instance from MplCanvas or MplCanvas3Axes] the canvas that is going to be used to draw the input audio waveform

XL, XR [(F, N) ndarray] resp. left and right channel STFT arrays

Methods

Constructor [reads the input audio file, computes the STFT,] generates the different dictionaries (for the source part, harmonic patterns WF0, and for the filter part, smooth patterns WGAMMA).

automaticMelodyAndSeparation : launches sequence of methods to estimate the parameters, estimate the melody, then re-estimate the parameters and at last separate the lead from the rest, considering the lead is the most energetic source of the mixture (with some continuity regularity)

estimSIMMParams : estimates the parameters of the SIMM, i.e. HF0, HPHI, HGAMMA, HM and WM

estimStereoSIMMParams : estimates the parameters of the stereo version of the SIMM, i.e. same parameters as estimSIMMParams, with the alphas and betas

estimStereoSUIMMParams : same as above, but first adds 'noise' components to the source part

initiateHF0WithIndexBestPath : computes the initial HF0, before the estimation, given the melody line (estimated or not)

runViterbi : estimates the melody line from HF0, the energies of each F0 candidates

setOutputFileNames : triggered when the text fields are changed, changing the output filenames

writeSeparatedSignals : computing and writing the adaptive Wiener filtered separated files

`writeSeparatedSignalsWithUnvoice()` **:** computing and writing the adaptive Wiener filtered separated files, unvoiced parts.

References

This is a class that encapsulates our work on source separation, published as:

and

As of 3/1/2012, available at http://www.durrieu.ch/research

`autoMelSepAndWrite`(*maxFrames=1000*)
Fully automated estimation of melody and separation of signals.

`automaticMelodyAndSeparation`()
Fully automated estimation of melody and separation of signals.

`checkChunkSize`(*maxFrames*)
Computes the number of chunks of size maxFrames, and changes maxFrames in case it does not provide long enough chunks (especially the last chunk).

`computeChroma`(*maxFrames=3000*)
Compute the chroma matrix.

`computeMonoX`(*start=0, stop=None*)
Computes and return SX, the mono channel or mean over the channels of the power spectrum of the signal

`computeNFrames`()
compute Nb Frames:

`computeWF0`()
Computes the frequency basis for the source part of SIMM, if tfrepresentation is a CQT, it also computes the cqt/hybridcqt transform object.

`determineTuning`()
Determine Tuning by checking the peaks corresponding to all possible patterns

`estimHF0`(*R=1, maxFrames=1000*)
estimating and storing only HF0 for the whole excerpt, with only

`estimStereoSIMMParamsWriteSeps`(*maxFrames=1000*)
Estimates the parameters little by little, by chunks, and sequentially writes the signals. In the end, concatenates all these separated signals into the desired output files

`estimStereoSUIMMParamsWriteSeps`(*maxFrames=1000*)
same as estimStereoSIMMParamsWriteSeps, but adds the unvoiced element in HF0

`setOutputFileNames`(*outputDirSuffix*)
If already loaded a wav file, at this point, we can redefine where we want the output files to be written.

Could be used, for instance, between the first estimation or the Viterbi smooth estimation of the melody, and the re-estimation of the parameters.

writeSeparatedSignals(*suffix='.wav'*)
Writes the separated signals to the files in self.files. If suffix contains 'VUIMM', then this method will take the WF0 and HF0 that contain the estimated unvoiced elements.

writeSeparatedSignalsWithUnvoice()
A wrapper to give a decent name to the function: simply calling self.writeSeparatedSignals with the '_VUIMM.wav' suffix.

1.2.4 separateLeadFunctions

separateLeadFunctions.py

Description

This module provides functions that are useful for the SeparateLeadStereo modules, essentially time-frequency transformations (and inverse), as well as generation of dictionary matrices.

Usage

See each function docstring for more information.

TODO: expend this? TODO: move all these functions in different modules, in ..tools for instance

License

Copyright (C) 2011-2013 Jean-Louis Durrieu

pyfasst.SeparateLeadStereo.separateLeadFunctions.**generateHannBasis**(*numberFrequencyBins*, *sizeOfFourier*, *Fs*, *frequencyScale='linear'*, *numberOfBasis=20*, *overlap=0.75*)
Generates a collection of Hann functions, spaced across the frequency axis, as desired by the user (and if implemented), targetting the given number of basis and adapting the extent (or bandwidth) of each function (over the frequencies) to that number and according to the desired overlap between these windows.

pyfasst.SeparateLeadStereo.separateLeadFunctions.**generate_ODGD_spec**(*F0*, *Fs*, *lengthOdgd=2048*, *Nfft=2048*, *Ot=0.5*, *t0=0.0*, *analysisWindowType='sinebell'*)
generateODGDspec:

generates a waveform ODGD and the corresponding spectrum, using as analysis window the -optional- window given as argument.

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_ODGD_spec_chirped`**(*F1*, *F2*, *Fs*, *lengthOdgd=2048*, *Nfft=2048*, *Ot=0.5*, *t0=0.0*, *analysisWindowType='sinebell'*)

generateODGDspecChirped:

generates a waveform ODGD and the corresponding spectrum, using as analysis window the -optional- window given as argument.

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_ODGD_spec_inharmo`**(*F0*, *Fs*, *lengthOdgd=2048*, *Nfft=2048*, *Ot=0.5*, *t0=0.0*, *analysisWindowType='sinebell'*, *inharmonicity=0.5*)

generateODGDspec:

generates a waveform ODGD and the corresponding spectrum, using as analysis window the -optional- window given as argument.

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_WF0_MinQT_chirped`**(*minF0*, *maxF0*, *cqtfmax*, *cqtfmin*, *cqtbins=48.0*, *Fs=44100.0*, *Nfft=2048*, *stepNotes=4*, *lengthWindow=2048*, *Ot=0.5*, *perF0=1*, *depthChirpInSemiTone=0.5*, *loadWF0=True*, *analysisWindow='hanning'*, *atomHopFactor=0.25*, *cqtWinFunc=<function hanning at 0x1027e7c08>*, *verbose=False*)

F0Table, WF0 = generate_WF0_MinCQT_chirped(minF0, maxF0, Fs, Nfft=2048, stepNotes=4, lengthWindow=2048, Ot=0.5, perF0=2, depthChirpInSemiTone=0.5)

Generates a 'basis' matrix for the source part WF0, using the source model KLGLOTT88, with the following I/O arguments: Inputs:

minF0 the minimum value for the fundamental frequency (F0)

maxF0 the maximum value for F0 cqtfmax... Fs the desired sampling rate Nfft the number of bins to compute the Fourier

transform

stepNotes the number of F0 per semitone lengthWindow the size of the window for the Fourier

transform

Ot the glottal opening coefficient for KLGLOTT88

perF0 the number of chirps considered per F0 value

depthChirpInSemiTone the maximum value, in semitone, of the allowed chirp per F0

Outputs:

F0Table the vector containing the values of the fundamental frequencies in Hertz (Hz) corresponding to the harmonic combs in WF0, i.e. the columns of WF0

WF0 the basis matrix, where each column is a harmonic comb generated by KLGLOTT88 (with a sinusoidal model, then transformed into the spectral domain)

20120828T2358 Horribly slow...

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_WF0_NSGTMinQT_chirped`**(*minF0*, *maxF0*, *cqtfmax*, *cqtfmin*, *cqtbins=48.0*, *Fs=44100.0*, *Nfft=2048*, *stepNotes=4*, *lengthWindow=2048*, *Ot=0.5*, *perF0=1*, *depthChirpInSemiTone=0.5*, *loadWF0=True* *analysisWindow='hanning'* *atomHopFactor=0.25*, *cqtWinFunc=<functio* *hanning at 0x1027e7c08>*, *verbose=False*)

F0Table, WF0 = generate_WF0_MinCQT_chirped(minF0, maxF0, Fs, Nfft=2048, stepNotes=4, lengthWindow=2048, Ot=0.5, perF0=2, depthChirpInSemiTone=0.5)

Generates a 'basis' matrix for the source part WF0, using the source model KLGLOTT88, with the following I/O arguments: Inputs:

minF0 the minimum value for the fundamental frequency (F0)

maxF0 the maximum value for F0 cqtfmax... Fs the desired sampling rate Nfft the number of bins to compute the Fourier transform

stepNotes the number of F0 per semitone lengthWindow the size of the window for the Fourier

transform

Ot the glottal opening coefficient for KLGLOTT88

perF0 the number of chirps considered per F0 value

depthChirpInSemiTone the maximum value, in semitone, of the allowed chirp per F0

Outputs:

F0Table the vector containing the values of the fundamental frequencies in Hertz (Hz) corresponding to the harmonic combs in WF0, i.e. the columns of WF0

WF0 the basis matrix, where each column is a harmonic comb generated by KLGLOTT88 (with a sinusoidal model, then transformed into the spectral domain)

20120828T2358 Horribly slow...

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_WF0_TR_chirped`**(*transform*, *minF0*, *maxF0*, *stepNotes=4*, *Ot=0.5*, *perF0=1*, *depthChirpInSemiTone=0.5*, *loadWF0=True*, *verbose=False*)

Generates a ‘basis’ matrix for the source part WF0, using the source model KLGLOTT88, with the following I/O arguments: Inputs:

minF0 the minimum value for the fundamental frequency (F0)

maxF0 the maximum value for F0 cqtfmax... Fs the desired sampling rate Nfft the number of bins to compute the Fourier

transform

stepNotes the number of F0 per semitone lengthWindow the size of the window for the Fourier

transform

Ot the glottal opening coefficient for KLGLOTT88

perF0 the number of chirps considered per F0 value

depthChirpInSemiTone the maximum value, in semitone, of the allowed chirp per F0

Outputs:

F0Table the vector containing the values of the fundamental frequencies in Hertz (Hz) corresponding to the harmonic combs in WF0, i.e. the columns of WF0

WF0 the basis matrix, where each column is a harmonic comb generated by KLGLOTT88 (with a sinusoidal model, then transformed into the spectral domain)

20120828T2358 Horribly slow...

`pyfasst.SeparateLeadStereo.separateLeadFunctions.`**`generate_WF0_chirped`**`(`*minF0*, *maxF0*, *Fs*, *Nfft=2048*, *stepNotes=4*, *lengthWindow=2048*, *Ot=0.5*, *perF0=1*, *depthChirpInSemiTone=0.5*, *loadWF0=True*, *analysisWindow='hanning'*`)`

F0Table, WF0 = generate_WF0_chirped(minF0, maxF0, Fs, Nfft=2048, stepNotes=4, lengthWindow=2048, Ot=0.5, perF0=2, depthChirpInSemiTone=0.5)

Generates a 'basis' matrix for the source part WF0, using the source model KLGLOTT88, with the following I/O arguments: Inputs:

minF0 the minimum value for the fundamental frequency (F0)

maxF0 the maximum value for F0 Fs the desired sampling rate Nfft the number of bins to compute the Fourier

transform

stepNotes the number of F0 per semitone lengthWindow the size of the window for the Fourier

transform

Ot the glottal opening coefficient for KLGLOTT88

perF0 the number of chirps considered per F0 value

depthChirpInSemiTone the maximum value, in semitone, of the allowed chirp per F0

Outputs:

F0Table the vector containing the values of the fundamental frequencies in Hertz (Hz) corresponding to the harmonic combs in WF0, i.e. the columns of WF0

WF0 the basis matrix, where each column is a harmonic comb generated by KLGLOTT88 (with a sinusoidal model, then transformed into the spectral domain)

`istft(X, analysisWindow=None, window=array([ 0.        ,  0.00153398,  0.00306796, ...,  0`
`0.00306796,  0.00153398]), hopsize=256.0, nfft=2048.0, originalDataLen=None, start=-1, sto`

data = istft(X, window=sinebell(2048), hopsize=256.0, nfft=2048.0)

Computes an inverse of the short time Fourier transform (STFT), here, the overlap-add procedure is implemented.

Inputs: X : STFT of the signal, to be "inverted" window=sinebell(2048) : synthesis window

(should be the "complementary" window for the analysis window)

hopsize=1024.0 : hopsize for the analysis nfft=2048.0 : number of points for the Fourier

computation (the user has to provide an even number)

Outputs:

data [time series corresponding to the given] STFT the first half-window is removed, complying with the STFT computation given in the function 'stft'

`stft(data, window=array([ 0.        ,  0.00153398,  0.00306796, ...,  0.00460193, 0.00306796,  0.00153398]), hopsize=256.0, nfft=2048.0, fs=44100.0, start=0, stop=None)`

X, F, N = stft(data, window=sinebell(2048), hopsize=1024.0, nfft=2048.0, fs=44100)

Computes the short time Fourier transform (STFT) of data.

Inputs:

data [one-dimensional time-series to be] analyzed

window=sinebell(2048) : analysis window hopsize=1024.0 : hopsize for the analysis nfft=2048.0 : number of points for the Fourier

computation (the user has to provide an even number)

fs=44100.0 : sampling rate of the signal

Outputs: X : STFT of data F : values of frequencies at each Fourier

bins

N [central time at the middle of each] analysis window

1.2.5 Spatial Signal Models

draws directivity diagrams for ULA assumption

Contents

dirdiag.py

Draw directivity diagrams

2013 Jean-Louis Durrieu http://www.durrieu.ch

`pyfasst.spatial.dirdiag.`**`directivity_filter_diagram_ULA`**(*n_sensors=2, dist_inter_sensor=0.15, w_filter=None, theta_filter=0, freqs=None, thetas=None, nfreqs=256, nthetas=30, samplerate=8000.0, doplot='2d', fig=None, subplot_number=(1, 1, 1), dyn_func=<function db at 0x10262b230>*)

Computes and displays the directivity diagram associated to the provided filter `w_filter` (optionally parameterized by the targetted direction `theta_filter`).

the diagram can be interpreted as the amplitude of the filter for a source at frequency `f`, located at an angle `theta` from the the ULA array axis.

`pyfasst.spatial.dirdiag.`**`generate_steer_vec_thetas`**(*n_sensors=2*, *dist_inter_sensors=0.15*, *freqs=None*, *n_freqs=256*, *thetas=None*, *n_thetas=30*, *samplerate=8000.0*, *computeRaa=False*)

Generates a collection of steering vectors with the provided array and signal parameters.

`pyfasst.spatial.dirdiag.`**`make_MVDR_filter_target`**(*steering_vec_target*, *steering_vec_interf*)

make MVDR spatial filter from estimated steering vectors

`pyfasst.spatial.dirdiag.`**`produceMVDRplots`**(*theta_filter=0.7853981633974483, freqs=None, n_freqs=256, thetas=None, n_thetas=30, samplerate=8000.0, n_sensors=2, dist_inter_sensors=0.15, dists=[0.15, 0.5, 1.0], doplot='2d', dyn_func=<function db at 0x10262b230>, theta_interf=None, n_theta_interf=4*)

MVDR gains for spatial filtering

Description:

Minimum Variance - Distortionless Response

Examples:

```
>>># importing the module
>>>import pyfasst.spatial.dirdiag as dd
>>># the MVDR plots, for 4 sensors and 4 interferers: visible rejection
>>>Raa, w, th, fr, thetas, diag = dd.produceMVDRplots(n_sensors=4,
       n_theta_interf=2, samplerate=8000., dists=[0.15,])
>>># plotting also the filter responses against the angles
>>>plt.figure();plt.plot(thetas,dd.db(diag))
>>>plt.xlabel('$\theta$ (rad)');plt.ylabel('Response (dB)')
>>># MVDR plots 2 sensors for 4 interferers: no rejection!
>>>Raa, w, th, fr, thetas, diag = dd.produceMVDRplots(n_sensors=2,
       n_theta_interf=2, samplerate=8000., dists=[0.15,])
>>># plotting also the filter responses against the angles
>>>plt.figure();plt.plot(thetas,dd.db(diag))
>>>plt.xlabel('$\theta$ (rad)');plt.ylabel('Response (dB)')
```

`pyfasst.spatial.dirdiag.`**`producePicDiagramAgainstDistNSensors`**(*w_filter=None*, *theta_filter=0.7853981633974483*, *sensors=None*, *dists=None*, *samplerate=8000.0*, *doplot='2d'*, *dyn_func=<function db at 0x10262b230>*, *thetas=None*, *nthetas=30*)

generate a drawing that shows the directivity diagrams for several values of distance between sensors and number of sensors.

STEERING VECTORS

generating steering vectors for arrays of sensors

Content

`pyfasst.spatial.steering_vectors.`**`dir_diag_stereo`**(*Cx*, *nft=2048*, *ntheta=512*, *samplerate=44100*, *distanceInterMic=0.3*)

Compute the diagram of directivity for the input short time Fourier transform second order statistics in Cx (this Cx is compatible with the attribute from an instantiation of `pyfasst.audioModel.FASST`)

$$C_x[0] = E[|x_0|^2]$$
$$C_x[2] = E[|x_1|^2]$$
$$C_x[1] = E[x_0 x_1^H]$$

Method:

We use the Capon method, on each of the Fourier channel k:

$$\phi_k(\theta) = a_k(\theta)^H R_{xx}^{-1} a_k(\theta)$$

The algorithm therefore returns one directivity graph for each frequency band.

Remarks:

One can compute a summary directivity by adding the directivity functions across all the frequency channels. The invert of the resulting array may also be of interest (looking at peaks and not valleys to find directions):

```
>>> directivity_diag = dir_diag_stereo(Cx)
>>> summary_dir_diag = 1./directivity_diag.sum(axis=1)
```

Some tests show that it is very important that the distance between the microphone is known. Otherwise, little can be infered from the resulting directivity measure...

`pyfasst.spatial.steering_vectors.`**`gen_steer_vec_acous`**(*freqs*, *dist_src_mic*)

generates a steering vector for the given frequencies and given distances between the microphones and the source.

To the difference with `gen_steer_vec_far_src_uniform_linear_array()`, this function also includes gains depending on the distance between the source and the mics.

`pyfasst.spatial.steering_vectors.`**`gen_steer_vec_far_src_uniform_linear_array`**(*freqs*, *nchannels*, *theta*, *distanceInterMic*)

generate steering vector with relative far source, uniformly spaced sensor array

Description:

assuming the source is far (compared to the dimensions of the array) The sensor array is also assumed to be a linear array, the direction of arrival (DOA) theta is defined as in the following incredible ASCII art drawing:

```
  theta
----->/              /
|    /              /
```

```
y   /              /
   /              /
^ /              /
|/              /
+---> x
o    o    o    o    o    o
M1   M2   M3  ...
<--->
  d = distanceInterMic
```

That is more likely valid for electro-magnetic fields, for acoustic wave fields, one should probably take into account the difference of gain between the microphones (see `gen_steer_vec_acous()`)

Output:

a (nc, nfreqs) ndarray contains the steering vectors, one for each channel, and

1.2.6 Time-Frequency Transforms

TFT module

Time-Frequency Transforms

TODO: turn this into something more self-contained (like defining a super class for all the possible time-freq transforms)

STFT module

`filter_conv_stft(data, W, analysisWindow=None, synthWindow=array([ 0.        ,  0.00153398,`
`0.00306796,  0.00153398]), hopsize=256.0, nfft=2048.0, fs=44100.0, verbose=0)`

Sequentially compute Fourier transfo, filter and overlap-add

INPUTS

W M x F x N (or M x F) filter for the data, which should be single channel

data T (number of samples, number of channels)

...

`filter_stft(data, W, analysisWindow=None, synthWindow=array([ 0.        ,  0.00153398,  0.`
`0.00306796,  0.00153398]), hopsize=256.0, nfft=2048.0, fs=44100.0)`

Sequentially compute Fourier transfo, filter and overlap-add

W is the M x M x F x N filter for the data, which should be T x M data T x M (number of samples, number of channels)

`istft(X, window=array([ 0.        ,  0.00153398,  0.00306796, ...,  0.00460193,`
`0.00306796,  0.00153398]), analysisWindow=None, hopsize=256.0, nfft=2048.0)`

data = istft(X,window=sinebell(2048),hopsize=1024.0,nfft=2048.0,fs=44100)

Computes an inverse of the short time Fourier transform (STFT), here, the overlap-add procedure is implemented.

Inputs:

X : STFT of the signal, to be "inverted"

window=sinebell(2048) : synthesis window (should be the "complementary" window for the analysis window)

hopsize=1024.0 : hopsize for the analysis

nfft=2048.0 : number of points for the Fourier computation (the user has to provide an even number)

Outputs:

data : time series corresponding to the given STFT the first half-window is removed, complying with the STFT computation given in the function stft

`stft(data, window=array([ 0.        ,  0.00153398,  0.00306796, ...,  0.00460193, 0.00306796,  0.00153398]), hopsize=256.0, nfft=2048.0, fs=44100.0)`

X, F, N = stft(data,window=sinebell(2048),hopsize=1024.0, nfft=2048.0,fs=44100)

Computes the short time Fourier transform (STFT) of data.

Inputs:

data : one-dimensional time-series to be analyzed

window=sinebell(2048) : analysis window

hopsize=1024.0 : hopsize for the analysis

nfft=2048.0 : number of points for the Fourier computation (the user has to provide an even number)

fs=44100.0 : sampling rate of the signal

Outputs:

X : STFT of data

F : values of frequencies at each Fourier bins

N : central time at the middle of each analysis window

MinQTmodule

Constant-Q transform after the work by C. Scholkhuber and A. Klapuri 2010 [SK2010]

Adaptation of the Constant Q transform as presented in

Comments beginning with '%' and '%%' are retained from the original Matlab code.

Python/Numpy/Scipy by Jean-Louis Durrieu, EPFL, 2012 - 2013

class `pyfasst.tftransforms.minqt.`**`CQTKernel`**(*fmax*, *bins*, *fs*, *q=1*, *atomHopFactor=0.25*, *thresh=0.0005*, *winFunc=<function sqrt_blackmanharris at 0x10267bc08>*, *perfRast=0*)

The CQT Kernel contains everything that can be precomputed for Constant-Q transforms. This relies on [SK2010], and therefore computes a Kernel for a single octave. It is then efficiently used to compute the decomposition on the different octaves by downsampling the signal.

Parameters:

fmax The maximum desired central frequency

bins The number of bins per octave

fs Sampling rate of the audio files

q parameter that controls the quality

atomHopFactor hopsize rate (0.25 is a hopsize of 25% the size of the windows) between successive analysis windows

thresh threshold value for sparsifying the kernel (Note: in this implementation, we do not use the sparsity, more efficiency could be achieved by considering it)

winFunc (python function that outputs an array) the analysis window function

perfRast whether computing rasterized version or not (if so, the decompositions at all scales will have the same number of frames, otherwise, each lower analysis octave will have half as many frames as the direct upper analysis octave.)

Attributes:

sparKernel weight atomHOP FFTLen fftOLP fftHOP bins winNr Nk_max Q fmin fmax frequencies perfRast first_center fs winFunc thresh q

class `pyfasst.tftransforms.minqt.`**`CQTransfo`**(*fmin*, *fmax*, *bins*, *fs*, *q=1*, *atomHopFactor=0.25*, *thresh=0.0005*, *winFunc=<function sqrt_blackmanharris at 0x10267bc08>*, *perfRast=0*, *cqtkernel=None*, *lowPassCoeffs=None*, *data=None*, *verbose=0*, ***kwargs*)

Constant Q Transform

`cellCQT2spCQT`()
compute the full cqt from self.cellCQT

`computeTransform`(*data*)
Computes the desired transform

`freq_stamps`
frequency stamps for spCQT

`invertFromCellCQT`()
inverting the Cell CQT

`invertFromSpCQT`()
Assuming we have self.spCQT, and not self.cellCQT, we recompute self.cellCQT from self.spCQT, and then invert as usual.

NB: here, self.cellCQT is written over, if it existed.

`invertFromSpCQTRast`()
this inverts the transform, if perfRast, then this means we can invert each hop of the different octaves.

`invertTransform`()
Invert the desired transform, here invert CQT from the cell CQT: like the original from [Schorkhuber2010]

`qValues`
Q values, approximated

`spCQT`
spCQT: the constant Q transform, in a readable format.

`spCQT2CellCQT`()
generates self.cellCQT from self.spCQT

NB: after transformation of spCQT (by filtering, for instance), this method only keeps downsampled versions of each CQT representation for each octave. More elaborated computations may be necessary to take into account more precise time variations at low frequency octaves.

`time_stamps`
time stamps for spCQT

`transfo`
returns the computed transform

class `pyfasst.tftransforms.minqt.`**`HybridCQTKernel`**(***kwargs*)

Hybrid CQT/Linear kernel

`computeMissingLinearFreqKernel`()

Compute the missing (high) frequency components, and make a similar Kernel for them.

We can use this for the first octave (the highest frequency octave) to extend the high frequencies. Actually, this can be used to compute a hybrid CQT transform on the low frequencies, while keeping linear freqs in the high spectrum, and still benefiting from the invertibility of the CQT transform by Schoerkhuber and Klapuri

class `pyfasst.tftransforms.minqt.`**`HybridCQTransfo`**(***kwargs*)

Hybrid Constant Q Transform

`cellCQT2spCQT`()

the spCQT is computed from self.cellCQT

`computeHybrid`(*data*)

calculates a hybrid CQT/FT representation of a sound stored in data

`computeLinearPart`(*data*)

Same as computeCQT, except it uses the linear frequency components in cqtkernel.linearSparKernel

NB: since this should be equivalent to computing an FFT after windowing each frame, there may be a faster way of implementing this function. For now, keeping the same rules as the original CQT implementation, for consistency and also for avoiding problems with window synchrony

`computeTransform`(*data*)

Computes the desired transform

`invertHybridCQT`()

Invert the hybrid transform.

Linearity allows to perform the cqt inverse first, and add the inverse of the linear freqs part thereafter (or the other way around).

`invertLinearPart`()

This inverts the linear part of the hybrid transform

NB: as for the computation of this part in transform, a windowed version of a plain FFT should do the same job, and faster.

`invertTransform`()

invert the desired transform

class `pyfasst.tftransforms.minqt.`**`MinQTKernel`**(*bins*, *fmax*, *fs*, *linFTLen=2048*, ***kwargs*)

Min Q Transform Kernel

`computeMissingLinearFreqKernel`()

Compute the missing (high) frequency components, and make a similar Kernel for them.

We can use this for the first octave (the highest frequency octave) to extend the high frequencies. Actually, this can be used to compute a hybrid CQT transform on the low frequencies, while keeping linear freqs in the high spectrum, and still benefiting from the invertibility of the CQT transform by Schoerkhuber and Klapuri

class `pyfasst.tftransforms.minqt.`**`MinQTransfo`**(*fmax*, *bins*, *linFTLen*, *fs*, *fmin=70*, ***kwargs*)

Minimum Q Transform

`cellCQT2spCQT`()

converts the cellCQT into a spCQT (matrix form)

computeLinearPart (*data*)
Compute the linear frequency part with an STFT, and taking only the desired frequencies.

computeTransform (*data*)
Computes the desired transform

invertFromSpCQTRast ()

invertLinearPart ()
This inverts the linear part of the hybrid transform

NB: as for the computation of this part in transform, a windowed version of a plain FFT should do the same job, and faster.

invertTransform ()
invert the desired transform

linCellCQT2LinSpCQT ()
converts cellCQT['linear'] into the corresponding bins in self._spCQT

spCQT2CellCQT ()
Reverting spCQT (matrix form of the transform) back into the (original?) cellCQT (one matrix per octave, one for the linear part of the transform).

1.2.7 TOOLS

The `tools` package contains various modules with different uses.

DISTANCES

distances.py

distance/divergence functions

2013 Jean-Louis Durrieu

`pyfasst.tools.distances.`**ISDistortion** (*X*, *Y*)
value = ISDistortion(X, Y)

Returns the value of the Itakura-Saito (IS) divergence between matrix X and matrix Y. X and Y should be two NumPy arrays with same dimension.

NONNEGATIVE MATRIX FACTORIZATION

Simple Nonnegative Matrix Factorization (NMF) routines to be used to estimate initial parameters in FASST.

Relevant references:

`pyfasst.tools.nmf.`**NMF_decomp_init** (*SX*, *nbComps=10*, *niter=10*, *verbose=0*, *Winit=None*, *Hinit=None*, *updateW=True*, *updateH=True*)
NMF multiplicative gradient, for Itakura Saito divergence measure between `SX` and `np.dot(W,H)`

See for instance [Fevotte2009].

Note: for (probably marginal) efficiency, the amplitude matrix `H` is "transposed", such that its use in the `np.dot` operations uses a C-ordered contiguous array. The output is however in the "correct" form.

`pyfasst.tools.nmf.`**`NMF_decomposition`**`(`*SX*, *nbComps=10*, *niter=10*, *verbose=0*`)`

NMF multiplicative gradient, for Itakura Saito divergence measure between SX and `np.dot(W,H)`

See for instance [Fevotte2009].

`pyfasst.tools.nmf.`**`SFNMF_decomp_init`**`(`*SX*, *nbComps=10*, *nbFiltComps=10*, *niter=10*, *verbose=0*, *Winit=None*, *Hinit=None*, *WFiltInit=None*, *HFiltInit=None*, *updateW=True*, *updateH=True*, *updateWFilt=True*, *updateHFilt=True*, *nbResComps=2*`)`

Implements a simple source/filter NMF algorithm, similar to that introduced in [Durrieu2010]

UTILS

`utils.py`

Useful functions for (audio) signal processing

2013 Jean-Louis Durrieu

http://www.durrieu.ch

Content

`pyfasst.tools.utils.`**`db`**`(`*val*`)`

`db()` db(positiveValue)

Returns the decibel value of the input positiveValue

`pyfasst.tools.utils.`**`hann`**`(`*args*`)`

window = hann(args)

Computes a Hann window, with NumPy's function hanning(args).

`pyfasst.tools.utils.`**`ident`**`(`*energy*`)`

`ident()` : identity function, return the inputs unchanged

`pyfasst.tools.utils.`**`nextpow2`**`(`*i*`)`

Find 2^n that is equal to or greater than.

code taken from the website:

http://www.phys.uu.nl/~haque/computing/WPark_recipes_in_python.html

`pyfasst.tools.utils.`**`sinebell`**`(`*lengthWindow*`)`

window = sinebell(lengthWindow)

Computes a "sinebell" window function of length L=lengthWindow

The formula is:

$$window(t) = sin(\pi \frac{t}{L}), t = 0..L-1$$

`pyfasst.tools.utils.`**`sqrt_blackmanharris`**`(`*M*`)`

A root-squared Blackman-Harris window function.

For use in scholkhuber and klapuri's framework.

SIGNALTOOLS

signalTools.py

gathers signal processing tools

`pyfasst.tools.signalTools.`**`f0detectionFunction`**(*TFmatrix*, *freqs=None*, *axis=None*, *samplingrate=44100*, *fouriersize=2048*, *f0min=80*, *f0max=3000*, *stepnote=16*, *numberHarmonics=20*, *threshold=0.5*, *detectFunc=<function sum at 0x10277a758>*, *weightFreqs=None*, *debug=False*)

Computes the Harmonic Sum

detectFunc should be a function taking an array as argument, and

threshold is homogenous to a tone on the western musical scale

`pyfasst.tools.signalTools.`**`harmonicProd`**(*TFmatrix*, ***kwargs*)

Computes the harmonic sum

`pyfasst.tools.signalTools.`**`harmonicSum`**(*TFmatrix*, ***kwargs*)

Computes the harmonic sum

`pyfasst.tools.signalTools.`**`invHermMat2D`**(*a_00*, *a_01*, *a_11*)

This inverts a set of 2x2 Hermitian matrices

better check `inv_herm_mat_2d()` instead, and replace all reference to this by the former.

`pyfasst.tools.signalTools.`**`inv_herm_mat_2d`**(*sigma_x_diag*, *sigma_x_off*, *verbose=False*)

Computes the inverse of 2D hermitian matrices.

Inputs

sigma_x_diag ndarray, with (dim of axis=0) = 2

The diagonal elements of the matrices to invert. sigma_x_diag[0] are the (0,0) elements and sigma_x_diag[1] are the (1,1) ones.

sigma_x_off ndarray, with the same dimensions as sigma_x_diag[0]

The off-diagonal elements of the matrices, more precisely the (0,1) element (since the matrices are assumed Hermitian, the (1,0) element is the complex conjugate)

Outputs

inv_sigma_x_diag ndarray, 2 x shape(sigma_x_off)

Diagonal elements of the inverse matrices. [0] <-> (0,0) [1] <-> (1,1)

inv_sigma_x_off ndarray, shape(sigma_x_off)

Off-diagonal (0,1) elements of the inverse matrices

det_sigma_x ndarray, shape(sigma_x_off)

For each inversion, the determinant of the matrix.

Remarks

The inversion is done explicitly, by computing the determinant (explicit formula for 2D matrices), then the elements of the inverse with the corresponding formulas.

To deal with ill-conditioned matrices, a minimum (absolute) value of the determinant is guaranteed.

`pyfasst.tools.signalTools.`**`medianFilter`**`(`*inputArray*, *length=10*`)`
median filter

`pyfasst.tools.signalTools.`**`prinComp2D`**`(`*X0*, *X1*, *neighborNb=10*, *verbose=0*`)`
Computes the eigen values and eigen vectors for a matrix X of shape 2 x F x N, computing the 2 x 2 covariance matrices for the F x N over the temporal neighborhood of size neighborNb.

`pyfasst.tools.signalTools.`**`sortSpectrum`**`(`*spectrum*, *numberHarmonicsHS=50*, *numberHarmonicsHP=1*, ***kwargs*`)`
Sort the spectra in `spectrum` with respect to their F0 values, as estimated by HS * HP function.

20130521 DJL sort of works, but periodicity detection should be reworked according to YIN and the like, in order to obtain better estimates.

1.3 Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[Arberet2010] Arberet, S.; Gribonval, R. and Bimbot, F., *A Robust Method to Count and Locate Audio Sources in a Multichannel Underdetermined Mixture*, IEEE Transactions on Signal Processing, 2010, 58, 121 - 133. [web]

[Durrieu2010] J.-L. Durrieu, G. Richard, B. David and C. F\'{e}votte, *Source/Filter Model for Main Melody Extraction From Polyphonic Audio Signals*, IEEE Transactions on Audio, Speech and Language Processing, special issue on Signal Models and Representations of Musical and Environmental Sounds, March 2010, Vol. 18 (3), pp. 564 – 575.

[Durrieu2011] J.-L. Durrieu, G. Richard and B. David, A Musically Motivated Representation For Pitch Estimation And Musical Source Separation, IEEE Journal of Selected Topics on Signal Processing, October 2011, Vol. 5 (6), pp. 1180 - 1191.

[Ozerov2012] A. Ozerov, E. Vincent, and F. Bimbot, A general flexible framework for the handling of prior information in audio source separation, IEEE Transactions on Audio, Speech and Signal Processing, Vol. 20 (4), pp. 1118-1133 (2012).

[Maazaoui2011] Maazaoui, M.; Grenier, Y. and Abed-Meraim, K. Blind Source Separation for Robot Audition using Fixed Beamforming with HRTFs, in proc. of INTERSPEECH, 2011.

[DDR2011] J.-L. Durrieu, B. David and G. Richard, A Musically Motivated Mid-Level Representation For Pitch Estimation And Musical Audio Source Separation, IEEE Journal of Selected Topics on Signal Processing, October 2011, Vol. 5 (6), pp. 1180 - 1191.

[DRDF2010] J.-L. Durrieu, G. Richard, B. David and C. F'evotte, Source/Filter Model for Main Melody Extraction From Polyphonic Audio Signals, IEEE Transactions on Audio, Speech and Language Processing, special issue on Signal Models and Representations of Musical and Environmental Sounds, March 2010, vol. 18 (3), pp. 564 – 575.

[SK2010] Schoerkhuber, C. and Klapuri, A., "Constant-Q transform toolbox for music processing," submitted to the 7th Sound and Music Computing Conference, Barcelona, Spain.

[Durrieu2010] J.-L. Durrieu, G. Richard, B. David and C. Fevotte, Source/Filter Model for Main Melody Extraction From Polyphonic Audio Signals, IEEE Transactions on Audio, Speech and Language Processing, special issue on Signal Models and Representations of Musical and Environmental Sounds, March 2010, Vol. 18 (3), pp. 564 – 575.

[Fevotte2009] C. Fevotte and N. Bertin and J.-L. Durrieu, Nonnegative matrix factorization with the Itakura-Saito divergence. With application to music analysis, Neural Computation, vol. 21 (3), pp. 793-830, March 2009. [pdf]

PYTHON MODULE INDEX

INDEX