talkbox Documentation

Release 0.2.3

David Cournapeau

March 26, 2009

CONTENTS

CHAPTER

ONE

INTRODUCTION

The following document describes how to use the talkbox scikits for signal processing. The document assumes basic knowledge of signal processing (Fourier Transform, Linear Time Invariant systems).

Talkbox is set of python modules for speech/signal processing. The following features are planned before a 1.0 release:

- Spectrum estimation related functions: both parametic (lpc, high resolution methods like music and co), and non-parametric (Welch, periodogram)
- Fourier-like transforms (DCT, DST, MDCT, etc...)
- Basic signal processing tasks such as resampling
- Speech related functionalities: mfcc, mel spectrum, etc..
- More as it comes

The goal of this toolbox is to be a sandbox for features which may end up in scipy at some point. I also want talkbox to be useful for both research and educational purpose. As such, a requirement is to have a pure python implementation for everything, with optional C/C++/Lisp for speed: reading signal processing in C is no fun, and neither is waiting for your mfcc computation one hour before ICASSP submission deadline :).

1.1 Prerequisites

Talkbox needs at least **Python 2.4** to run. It also needs Numpy and Scipy, as well as setuptools. Any recent version of numpy (1.0 and higher), scipy (0.6 and higher) should do.

Other useful packages are matplotlib, and audiolab scikits. They are not mandatory, but will be assumed for the examples in this documentation.

1.2 Installing talkbox

Unfortunately, at this point, your only option is to install talkbox from sources:

```
svn co http://svn.scipy.org/svn/scikits/trunk/talkbox
```

You can install talkbox with the usual setup.py method in the talkbox source tree:

```
python setup.py install
```

If you don't want to install it as a python egg (for stow, etc...), you can use:

```
python setup.py install --single-version-externally-managed --record=/dev/null
```

1.3 Starting with talkbox

1.3.1 main namespace

The main functions are available through the main scikits.talkbox namespace:

```
import scikits.talkbox as talk
```

1.3.2 Getting help

All high level functions are duely documented, and are available through the usual python facilities:

```
import scikits.talkbox as talk
help(talk)
```

Will give you online-help for the main talkbox functions.

CHAPTER

TWO

SPECTRAL ANALYSIS

Spectral signal analysis is the field concerned with the estimation of the distribution of a signal (more exactly its power) in the frequency domain.

The Power Spectrum Density (PSD) S_X of X_n is defined as the squared discrete time Fourier transform of X_n

$$\forall f \in \mathbb{R}, \qquad S_X = \left| \sum_n X_n e^{-2\pi j f n} \right|^2$$

Under suitable technical conditions, this is equal to the discrete time Fourier transform of the autocorrelation function for stationary signals in the wide sense:

$$\forall f \in \mathbb{R}, \qquad S_X = \sum_n \gamma(n) e^{-2\pi j f n}$$

This is called the power spectrum density because integrating it over the frequency domain gives you the average power of X and because it can be proved that S_X is always positive for any f.

2.1 Spectral density estimation

Since in practice, we only have finite signals, we need method to estimate the PSD. talkbox implements various methods for PSD estimation, which are classified in two broad classes:

- non-parametric (general methods, estimate the PSD directly from the signal)
- parametric (use an underlying signal model, for example AR; is less general, but generally more efficient in some sense if applicable).

2.2 Non-parametric estimation

2.2.1 Periodogram

The raw periodogram is a relatively straightforward estimator of the PSD. The raw periodogram I of a signal of length N is defined as:

$$I(f) \triangleq \frac{\left| \sum_n x[n] e^{-2\pi j k f / f_s} \right|^2}{f_s N}$$

where f_s is the sampling rate. In practice, the periodogram can only be computed on a frequency grid; the most commonly used grid is k/N (normalized frequencies) for k in [0, ..., N-1]. With this grid, the sum becomes a simple DFT which can be computed using the FFT.

Examples

As a first example, let's generate a simple sinusoid signal embedded into white noise:

```
import numpy as np
import matplotlib.pyplot as plt
from scikits.talkbox.spectral.basic import periodogram
fs = 1000
x = np.sin(2 * np.pi * 0.15 * fs * np.linspace(0., 0.3, 0.3 * fs))
x += 0.1 * np.random.randn(x.size)
px, fx = periodogram(x, nfft=16384, fs=fs)
plt.plot(fx, 10 * np.log10(px))
```

Plotting the log periodogram then gives:

The number of points used for the FFT has been set high to highlight the lobe, artefact of the rectangular window.

periodogram(*x, nfft=None, fs=1*)

Compute the periodogram of the given signal, with the given fft size.

Parameters **x** : array-like
input signal

nfft : int
size of the fft to compute the periodogram. If None (default), the length of the signal is used. if nfft > n, the signal is 0 padded.

fs : float
Sampling rate. By default, is 1 (normalized frequency. e.g. 0.5 is the Nyquist limit).

Returns **pxx** : array-like
The psd estimate.

fgrid : array-like
Frequency grid over which the periodogram was estimated.

Notes

Only real signals supported for now.

Returns the one-sided version of the periodogram.

Discrepency with matlab: matlab compute the psd in unit of power / radian / sample, and we compute the psd in unit of power / sample: to get the same result as matlab, just multiply the result from talkbox by 2pi

Examples

Generate a signal with two sinusoids, and compute its periodogram:

```
>>> fs = 1000
>>> x = np.sin(2 * np.pi  * 0.1 * fs * np.linspace(0, 0.5, 0.5*fs))
>>> x += np.sin(2 * np.pi  * 0.2 * fs * np.linspace(0, 0.5, 0.5*fs))
>>> px, fx = periodogram(x, 512, fs)
```

2.3 Parametric estimation

2.3.1 ar method

TODO: To be implemented

2.4 Useful complements

A random signal X is said to be (strictly) stationary if its distribution does not depend on the time. For time-discrete signals X_n of distribution F_{X_n} , this is written:

$$\forall n, k, \qquad F_{X_n} = F_{X_{n+k}}$$

Other stationary concepts can be defined, when only some moments of the signal do not depend on time. A widely used concept is weakly stationary signals. A signal is said to be weakly stationary (or stationary in the wide sense) if its means and covariance do not depend on time, and its covariance function $\gamma(n, k)$ depends only on the time difference n-k:

$$\forall n, k, \qquad \gamma(n, k) = \mathbb{E}[(X[n] - m)(X[k] - m)] = \gamma(n - k)$$

CHAPTER

THREE

LINEAR PREDICTION

The goal of linear-prediction is to model a signal as a linear combination of its past/future.

3.1 Yule-Walker equations

For a discrete-time signal x, we want to approximate the sample x[n] as a linear combination xp[n] of the k preceding samples:

xp[n] = -c[1] * x[n-2] - ... - c[k-1] * x[n-k-1]

The best approximation in the mean-square sense is a tuple(c[1], ..., c[k]) such as the squared error:

e = xp - x

Is minimal. Noting p(x) = xp, and x^{-k} the signal x^{-k}[n] = x[n-k], since p is a linear combination of the (x^-1, ..., x^-k), we know that the error p(x) - x is minimal for p the orthogonal project of x on the vector space V spanned by the x^-1, ..., x^-k. In particular, the error e is then orthogonal to any vector in V:

TODO: decent blob for above

$$\begin{pmatrix} -R[1] \\ -R[2] \\ \vdots \\ -R[p] \end{pmatrix} = \begin{pmatrix} R[0] & \overline{R}[1] & \dots & \overline{R}[p-1] \\ R[1] & R[0] & \ddots & \vdots \\ \vdots & \ddots & \ddots & \overline{R}[1] \\ R[p-1] & \dots & R[1] & R[0] \end{pmatrix} \begin{pmatrix} a[1] \\ \vdots \\ \vdots \\ a[p] \end{pmatrix}$$

3.2 Levinson-Durbin recursion

Levinson-Durbin recursion is a recursive algorithm to solve the Yule-Walker equations in O(p^2) instead of O(p^3) usually necessary to inverse a matrix. It uses the Hermitian-Toeplitz structure of the correlation matrix.

levinson(*r, order, axis=-1*)

Levinson-Durbin recursion, to efficiently solve symmetric linear systems with toeplitz structure.

Parameters **r** : array-like

input array to invert (since the matrix is symmetric Toeplitz, the corresponding pxp matrix is defined by p items only). Generally the autocorrelation of the signal for linear prediction coefficients estimation. The first item must be a non zero real, and corresponds to the autocorelation at lag 0 for linear prediction.

order : int
order of the recursion. For order p, you will get p+1 coefficients.
axis : int, optional
axis over which the algorithm is applied. -1 by default.

Returns **a** : array-like
the solution of the inversion (see notes).
e : array-like
the prediction error.
k : array-like
reflection coefficients.

Notes

Levinson is a well-known algorithm to solve the Hermitian toeplitz equation:

```
                    _          _
-R[1] = R[0]   R[1]   ... R[p-1]    a[1]
 :       :      :           :          :
 :       :      :           _      *   :
-R[p] = R[p-1] R[p-2] ... R[0]      a[p]
```

with respect to a. Using the special symmetry in the matrix, the inversion can be done in O(p^2) instead of O(p^3).

Only double argument are supported: float and long double are internally converted to double, and complex input are not supported at all.

3.3 Linear prediction coding

Solve the Yule-Walker equation for a signal x, using the autocorelation method and Levinson-Durbin for the Yule-Walker inversion.

lpc(*signal, order, axis=-1*)
Compute the Linear Prediction Coefficients.

Return the order + 1 LPC coefficients for the signal. c = lpc(x, k) will find the k+1 coefficients of a k order linear filter:

xp[n] = -c[1] * x[n-2] - ... - c[k-1] * x[n-k-1]

Such as the sum of the squared-error e[i] = xp[i] - x[i] is minimized.

Parameters **signal: array_like** :
input signal
order : int
LPC order (the output will have order + 1 items)

Returns **a** : array-like
the solution of the inversion.
e : array-like
the prediction error.
k : array-like
reflection coefficients.

Notes

This uses Levinson-Durbin recursion for the autocorrelation matrix inversion, and fft for the autocorrelation computation.

For small order, particularly if order << signal size, direct computation of the autocorrelation is faster: use levinson and correlate in this case.

INDEX