Methods for Segmentation in Music Structure Analysis (MSA)

by Andrea De Carlo, mat. 249518

General Goal

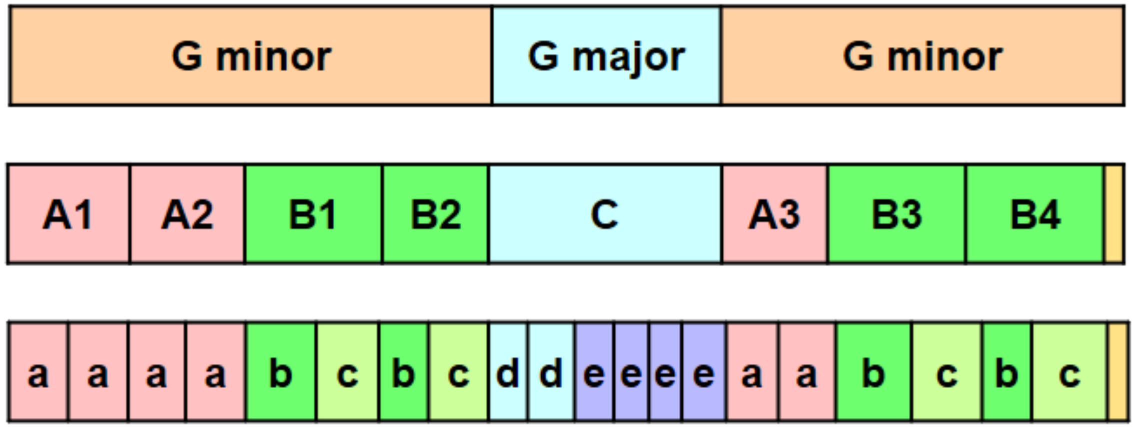


Figure 4.28 from [Müller, FMP, Springer 2015]

Hungarian Dance No. 5
by Johannes Brahms

Problem Definition

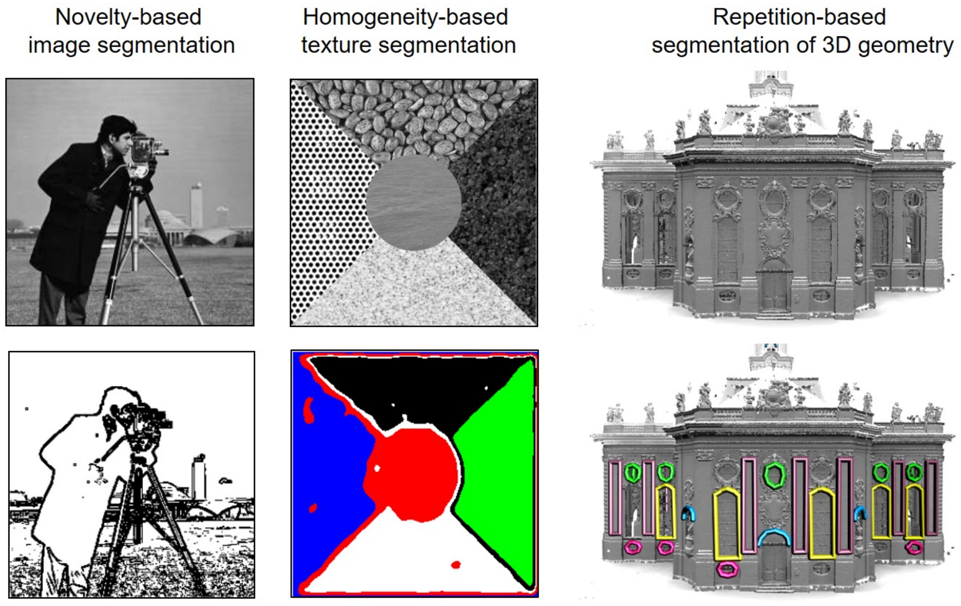
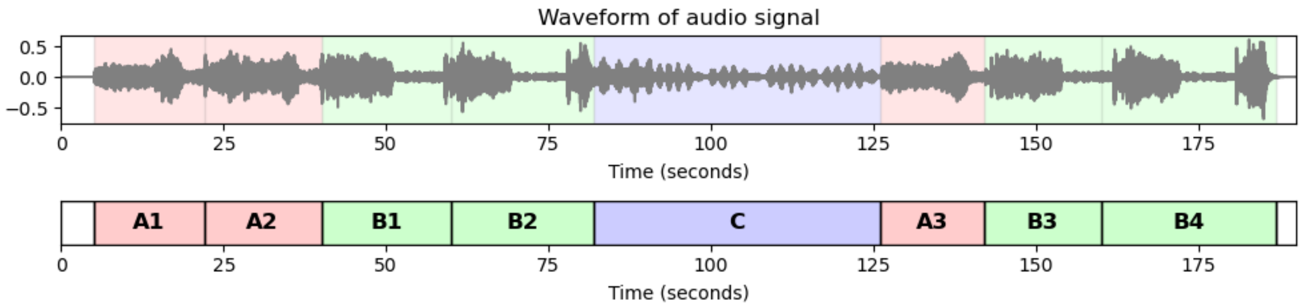


Figure 4.3 from [Müller, FMP, Springer 2015], 3D model by kind permission of [Sunkel et al., CGF, 2011]



Aspects to Consider

Time scale	Dimension	Content
Short term	Timbre	Quality of the produced sound
	Orchestration	Sources of sound production
	Acoustics	Quality of the recorded sound
Middle term	Rhythm	Patterns of sound onsets
	Melody	Sequences of notes
	Harmony	Sequences of chords
Long term	Structure	Organization of the musical work

Different Approaches

- ***Homogeneity***
- ***Repetition***
- ***Novelty***

What we will see

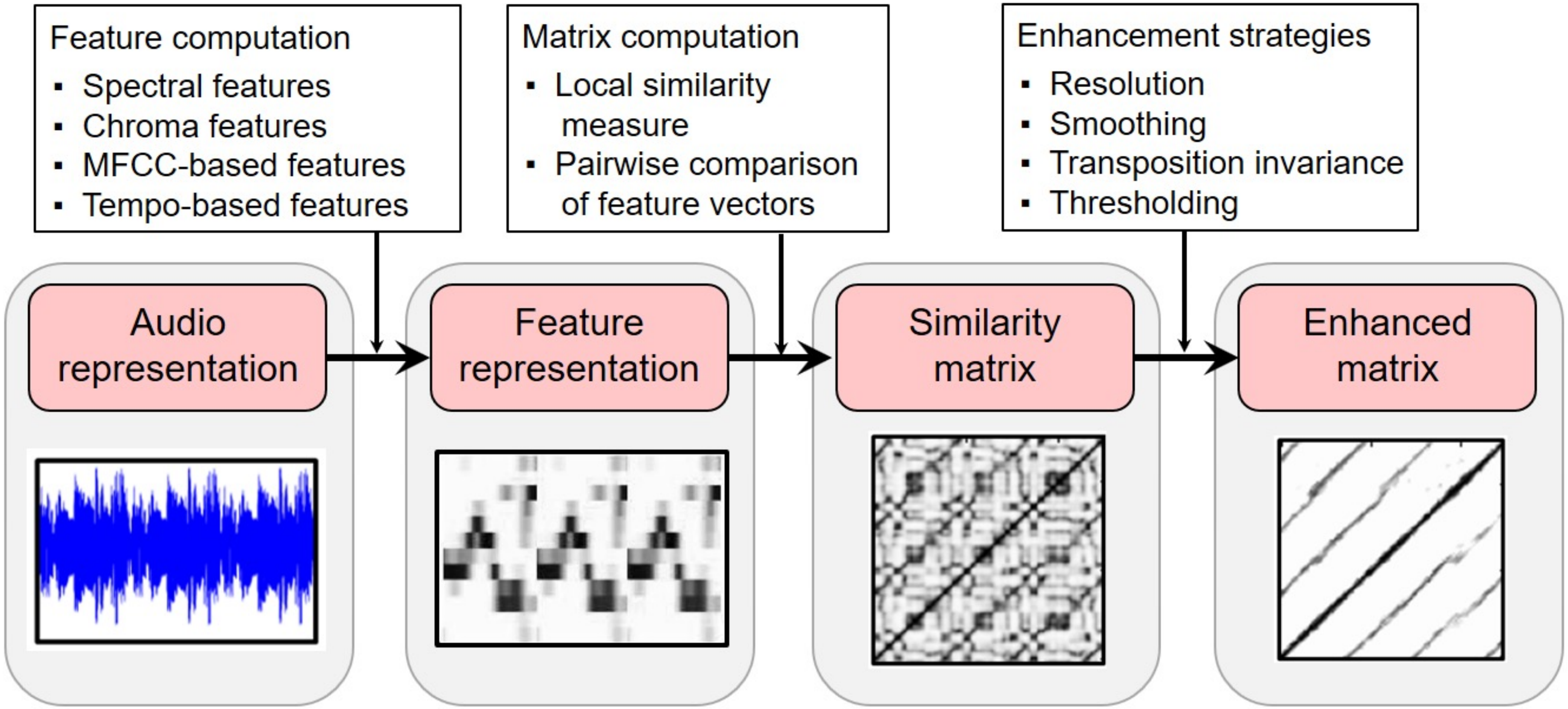
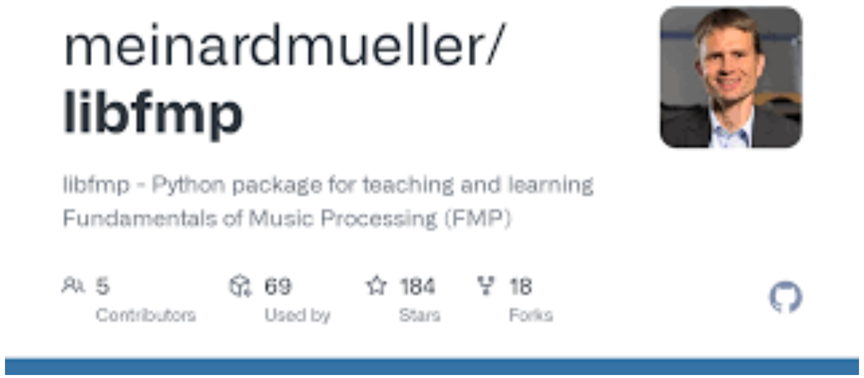


Figure 4.9 from [Müller, FMP, Springer 2015]

Libraries Needed

```python
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

from IPython.display import Audio

import numpy as np

import librosa

from libfmp import *
import libfmp.b
import libfmp.c3
import libfmp.c4
import libfmp.c6
```

librosa

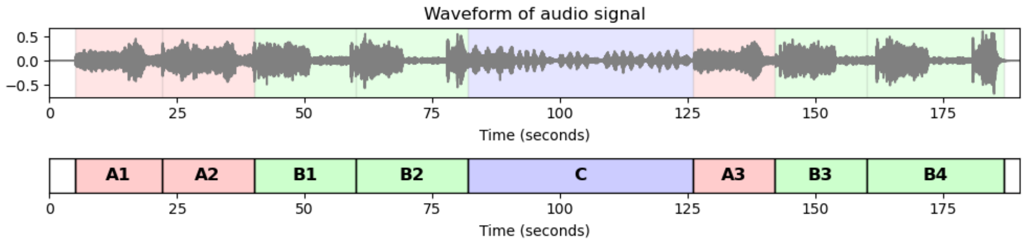


Importing the Audio File with Annotations

```
path = 'Hungarian Dance No. 5.mp3'

x, Fs = librosa.load(path)

x_duration = x.shape[0] // Fs

print(f'x.shape: {x.shape}')
print(f'samplerate: {Fs}')
print(f'x.shape / sr: {x_duration} seconds')

Audio(x, rate=Fs)
```

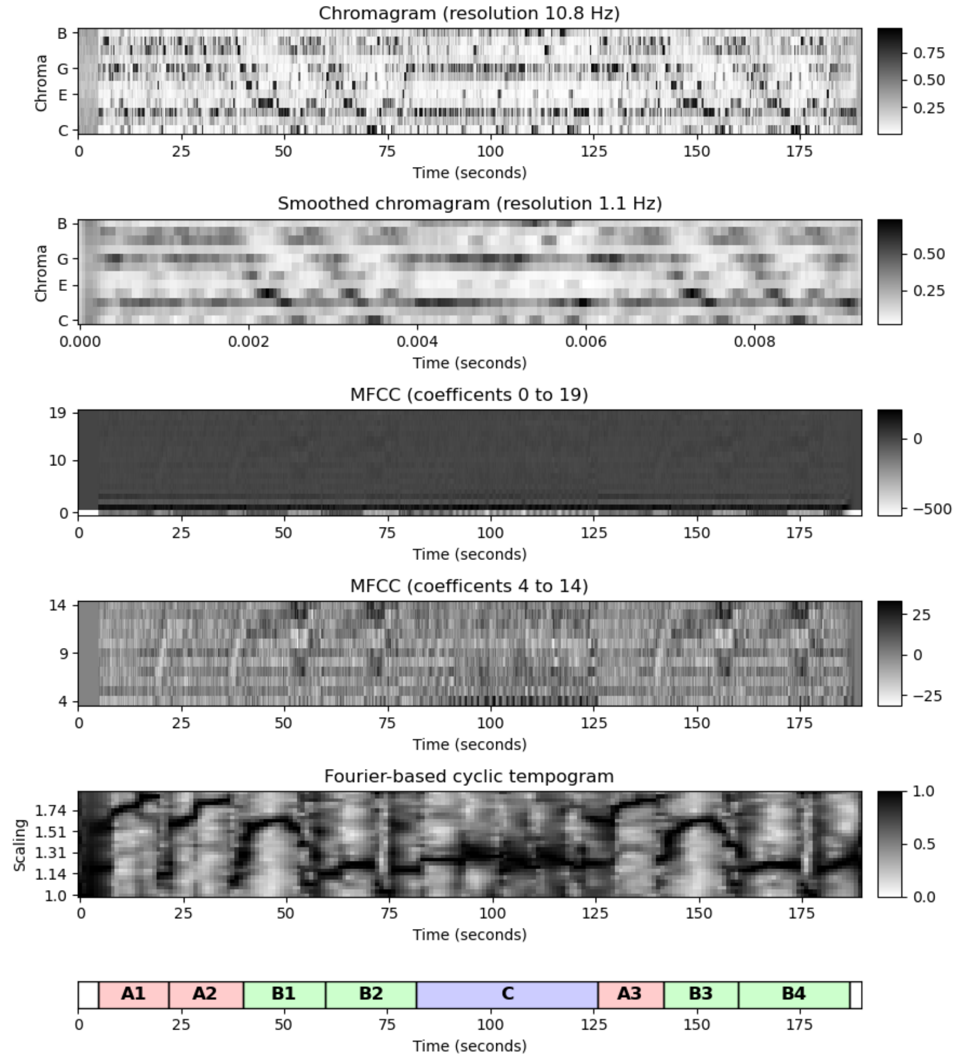
```
x.shape: (4189500,)
samplerate: 22050
x.shape / sr: 190 seconds
```


```
## Plot waveform with segmentation overlay
libfmp.b.plot_signal(x, Fs, ax=ax[0], title='Waveform of audio
signal')

libfmp.b.plot_segments_overlay(ann, ax=ax[0],
time_max=x_duration,
print_labels=False, label_ticks=False, edgecolor='gray',
colors = color_ann, fontsize=10, alpha=0.1)

## Plot segmentation
libfmp.b.plot_segments(ann, ax=ax[1], time_max=x_duration,
colors=color_ann, time_label='Time (seconds)')
```



Feature Representations

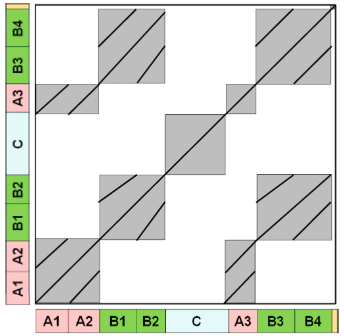
```
# ------------ chromagram ---------------------------------------------

chroma = librosa.feature.chroma_stft(y=x, sr=Fs,
tuning=0, # A440 tuning
norm=2, # L2 normalization
hop_length=H, # temporal resolution
n_fft=N # size of FFT window
)

filt_len = 41
down_sampling = 10
filt_kernel = np.ones([1, filt_len])

chroma_smoothed = signal.convolve(chroma, filt_kernel, mode='same') / filt_len
chroma_smoothed = chroma_smoothed[:, ::down_sampling]

# ------------ mfcc ---------------------------------------------------

mfcc = librosa.feature.mfcc(y=x, sr=Fs, hop_length=H, n_fft=N)
coef = np.arange(4, 15)
mfcc_upper = mfcc[coef, :]


# ------------ tempogram ----------------------------------------------

nov, sr_nov = libfmp.c6.compute_novelty_spectrum(x, Fs=Fs, N=2048, H=512,
gamma=100, M=10, norm=True)
nov, sr_nov = libfmp.c6.resample_signal(nov, Fs_in=sr_nov, Fs_out=100)

X, T_coef, F_coef_BPM = libfmp.c6.compute_tempogram_fourier(nov, sr_nov,
N=1000, H=100, Theta=np.arange(30, 601))
```

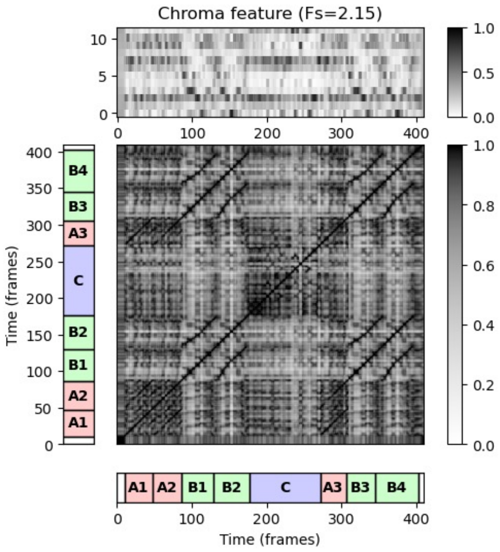


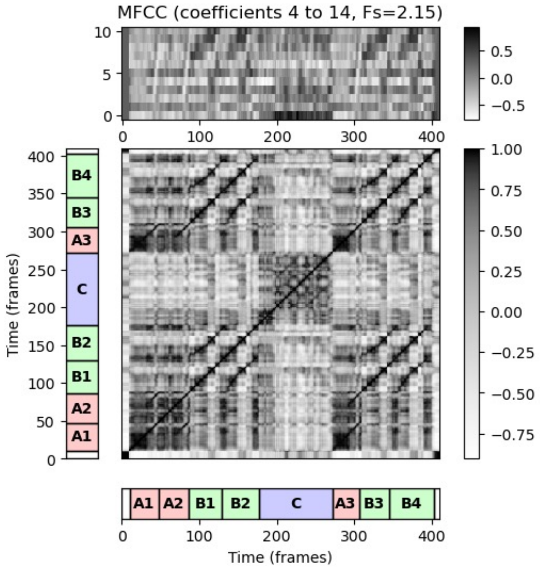
Segmentation Methods

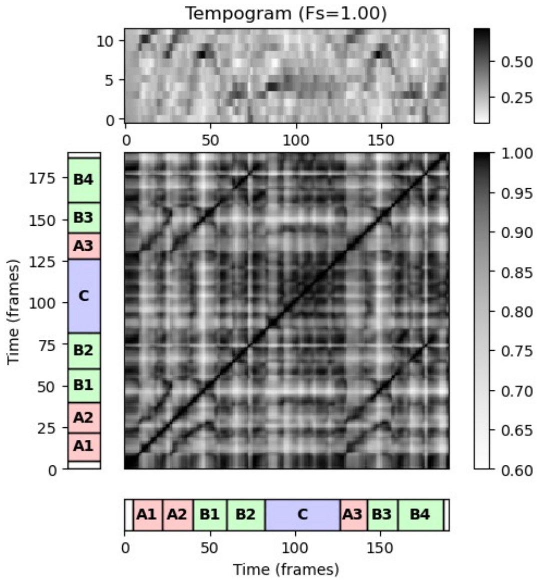
Self-Similarity Matrix

```
# Chroma Feature Sequence
N, H = 4096, 1024
chromagram = librosa.feature.chroma_stft(y=x, sr=Fs, tuning=0, norm=2, hop_length=H, n_fft=N)
X, Fs_X = libfmp.c3.smooth_downsample_feature_sequence(chromagram, Fs/H, filt_len=41, down_sampling=10)

# Annotation
ann_frames = libfmp.c4.convert_structure_annotation(ann, Fs=Fs_X)

# SSM
X = libfmp.c3.normalize_feature_sequence(X, norm='2', threshold=0.001)
S_chroma = compute_sm_dot(X,X)
fig, ax = plot_feature_ssm(X, 1, S_chroma, 1, ann_frames, x_duration*Fs_X, color_ann=color_ann,
clim_X=[0,1], clim=[0,1], label='Time (frames)',
title='Chroma feature (Fs=%0.2f)'%Fs_X)
```





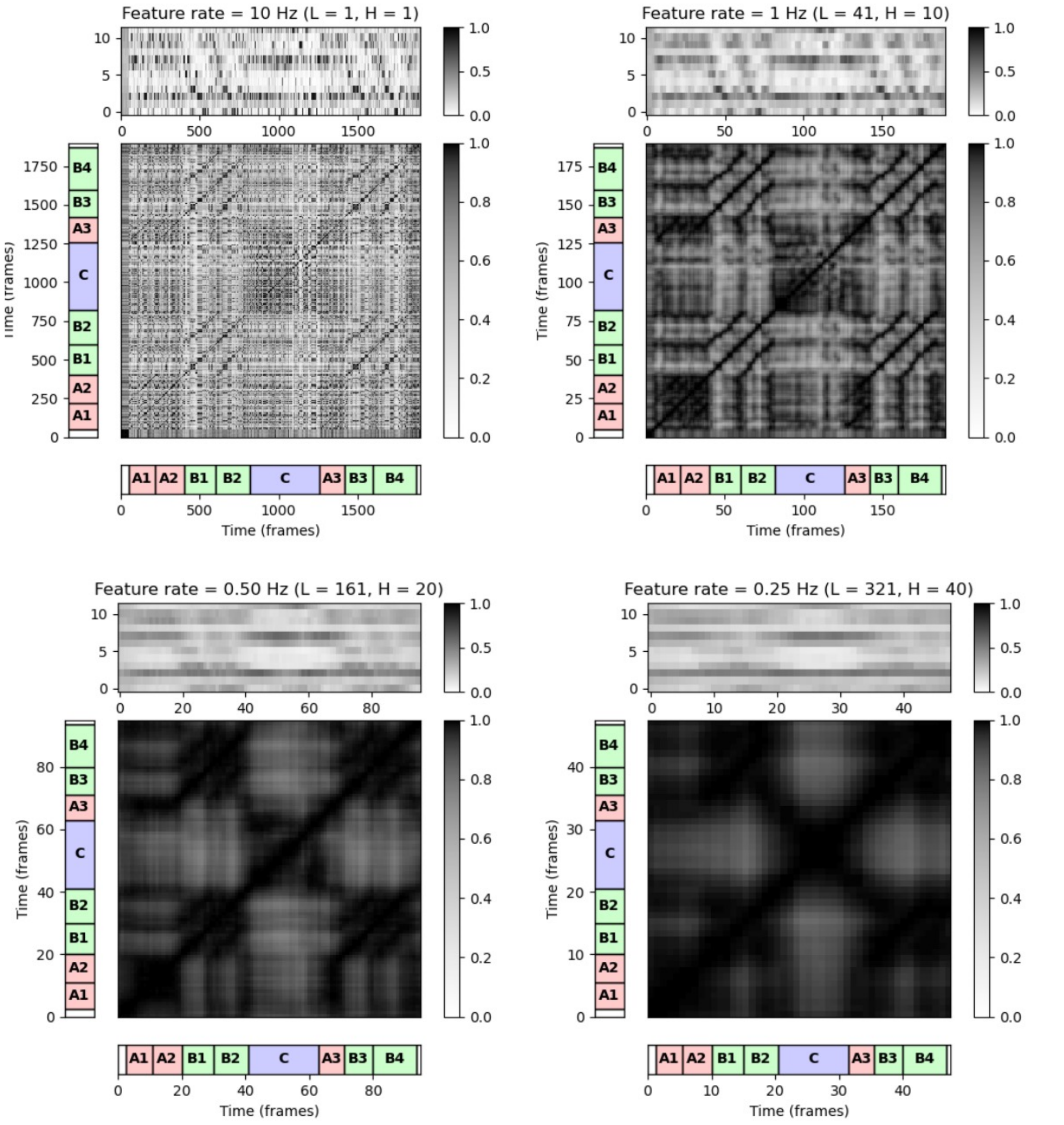




Path Enhancements

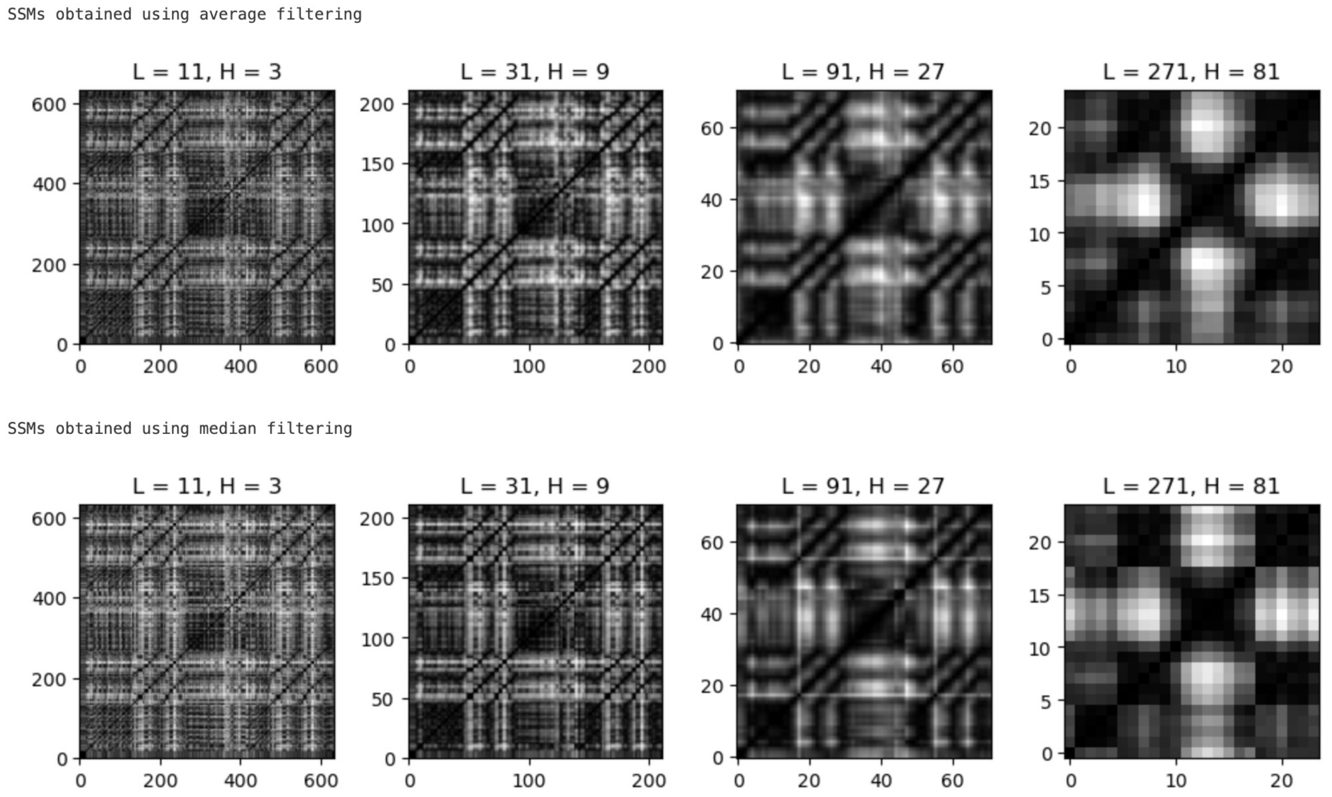
Smoothing

```
C = librosa.feature.chroma_stft(y=x, sr=Fs, tuning=0,
norm=2, hop_length=2205, n_fft=4410)
Fs_C = Fs/2205

# Chroma Feature Sequence and SSM (10 Hz)
L, H = 1, 1
X, Fs_feature =
libfmp.c3.smooth_downsample_feature_sequence(C, Fs_C,
filt_len=L, down_sampling=H)
X = libfmp.c3.normalize_feature_sequence(X, norm='2',
threshold=0.001)

S = libfmp.c4.compute_sm_dot(X,X)

# Chroma Feature Sequence and SSM (1 Hz)
L, H = 41, 10
...
# Chroma Feature Sequence and SSM (0.5 Hz)
L, H = 161, 20
...
# Chroma Feature Sequence and SSM (0.25 Hz)
L, H = 321, 40
...
```



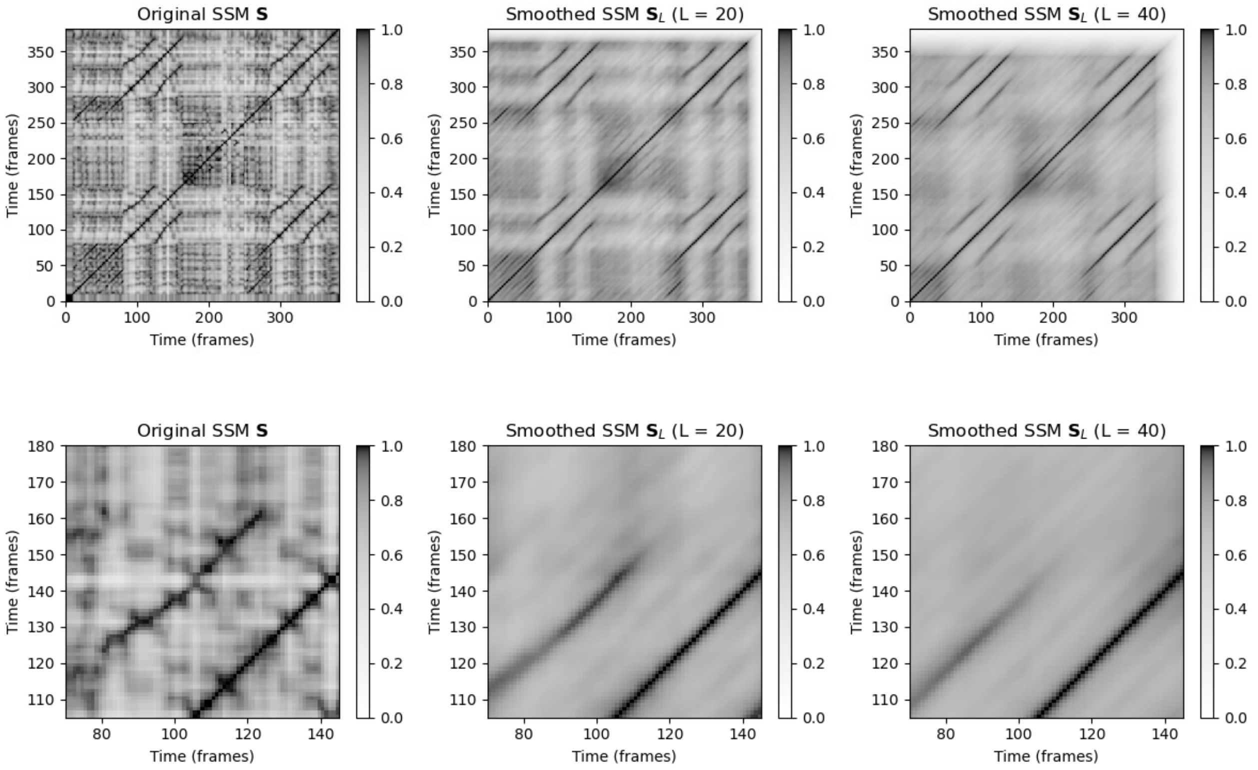
Median Filtering

```
# Chroma Feature Sequence and SSM (0.5 Hz)
L_iter = [11, 31, 91, 271]
H_iter = [ 3, 9, 27, 81]
num_iter = len(L_iter)

print('SSMs obtained using median filtering')
fig = plt.figure(figsize=(10,3))
for i in range(num_iter):
    L = L_iter[i]
    H = H_iter[i]
    X, Fs_feature =
    libfmp.c3.median_downsample_feature_sequence(C,
    Fs_C,
    filt_len=L, down_sampling=H)
    X = libfmp.c3.normalize_feature_sequence(X,
    norm='2', threshold=0.001)
    S = libfmp.c4.compute_sm_dot(X,X)
    ...
```



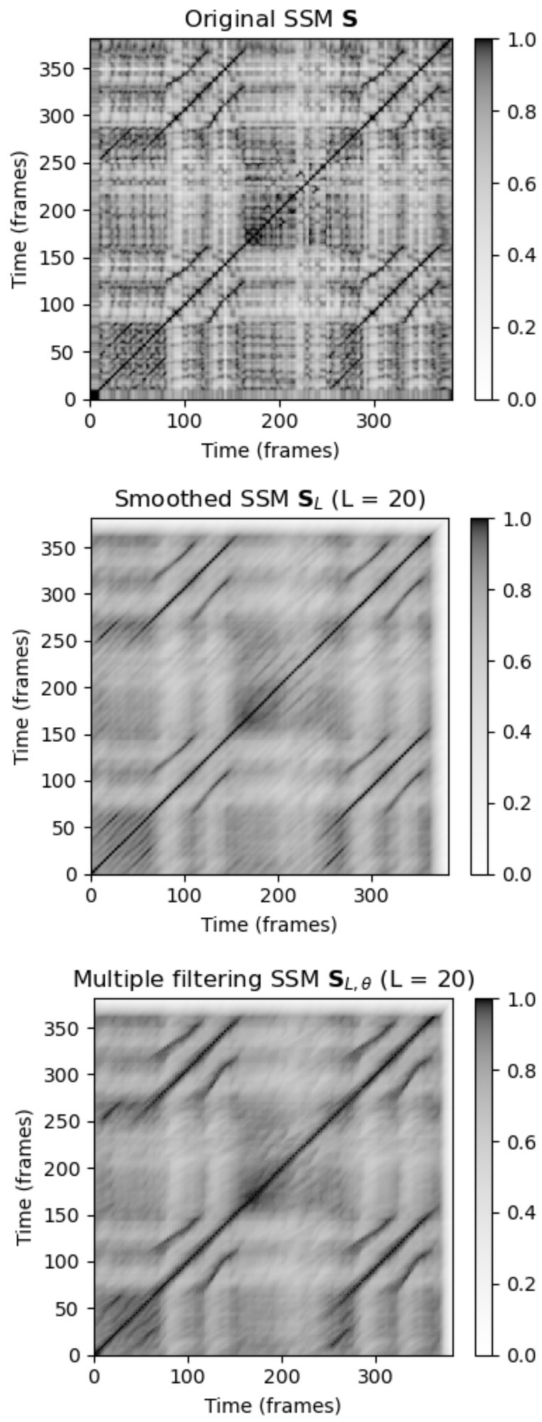
Diagonal Smoothing

```
def filter_diag_sm(S, L):
    """Path smoothing of similarity matrix by forward filtering
    along main diagonal
    """
    N = S.shape[0] # Number of rows
    M = S.shape[1] # Number of columns
    S_L = np.zeros((N, M))
    S_extend_L = np.zeros((N + L, M + L))
    S_extend_L[0:N, 0:M] = S # copy original matrix
    for pos in range(0, L):
          # add portion of matrix
         S_L = S_L + S_extend_L[pos:(N + pos), pos:(M + pos)]
    S_L = S_L / L # average
    return S_L
...
L = 20
S_L = filter_diag_sm(S, L)
subplot_matrix_colorbar(S_L, fig, ax[1], clim=[0,1], ylabel='Time
(frames)', xlabel='Time (frames)',
title=r'Smoothed SSM $\mathbf{S}_{L}$ (L = %d)'%L)
...
```



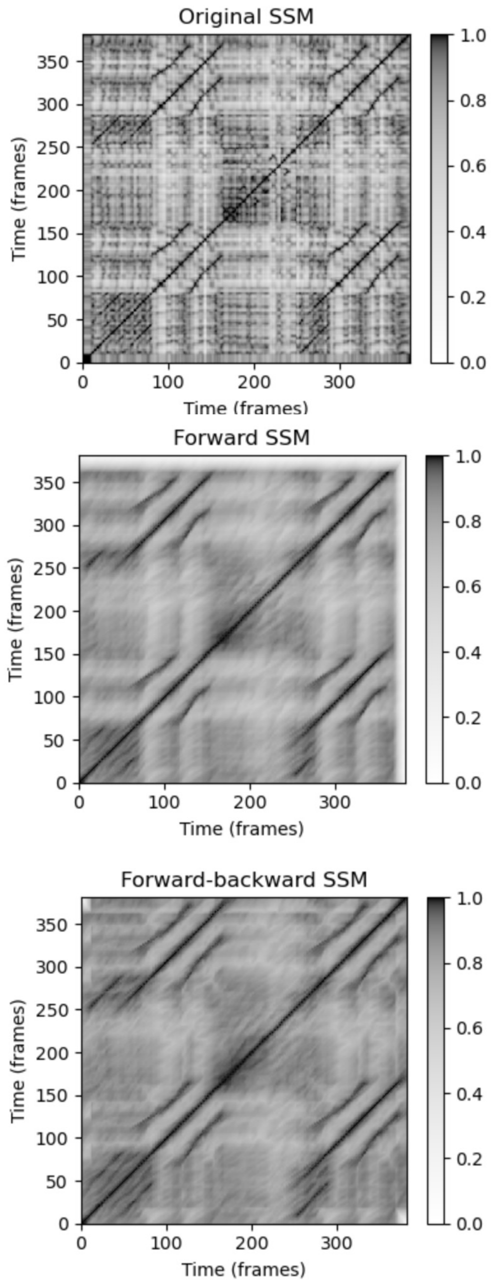
Multiple Filtering

```python
def filter_diag_mult_sm(S, L=1, tempo_rel_set=np.asarray([1])):
    N = S.shape[0]
    M = S.shape[1]
    num = len(tempo_rel_set)
    S_L_final = np.zeros((N, M))

    for s in range(0, num):
        M_ceil = int(np.ceil(M / tempo_rel_set[s]))

        resample = np.multiply(np.divide(np.arange(1, M_ceil+1), M_ceil), M)
        np.around(resample, 0, resample)
        resample = resample - 1
        index_resample = np.maximum(resample, np.zeros(len(resample))).astype(np.int64)
        S_resample = S[:, index_resample]

        S_L = np.zeros((N, M_ceil))
        S_extend_L = np.zeros((N + L, M_ceil + L))
        S_extend_L[0:N, 0:M_ceil] = S_resample
        for pos in range(0, L):
            S_L = S_L + S_extend_L[pos:(N + pos), pos:(M_ceil + pos)]
        S_L = S_L / L

        resample = np.multiply(np.divide(np.arange(1, M+1), M), M_ceil)
        np.around(resample, 0, resample)
        resample = resample - 1
        index_resample = np.maximum(resample, np.zeros(len(resample))).astype(np.int64)
        S_resample_inv = S_L[:, index_resample]

        S_L_final = np.maximum(S_L_final, S_resample_inv)

    return S_L_final
```



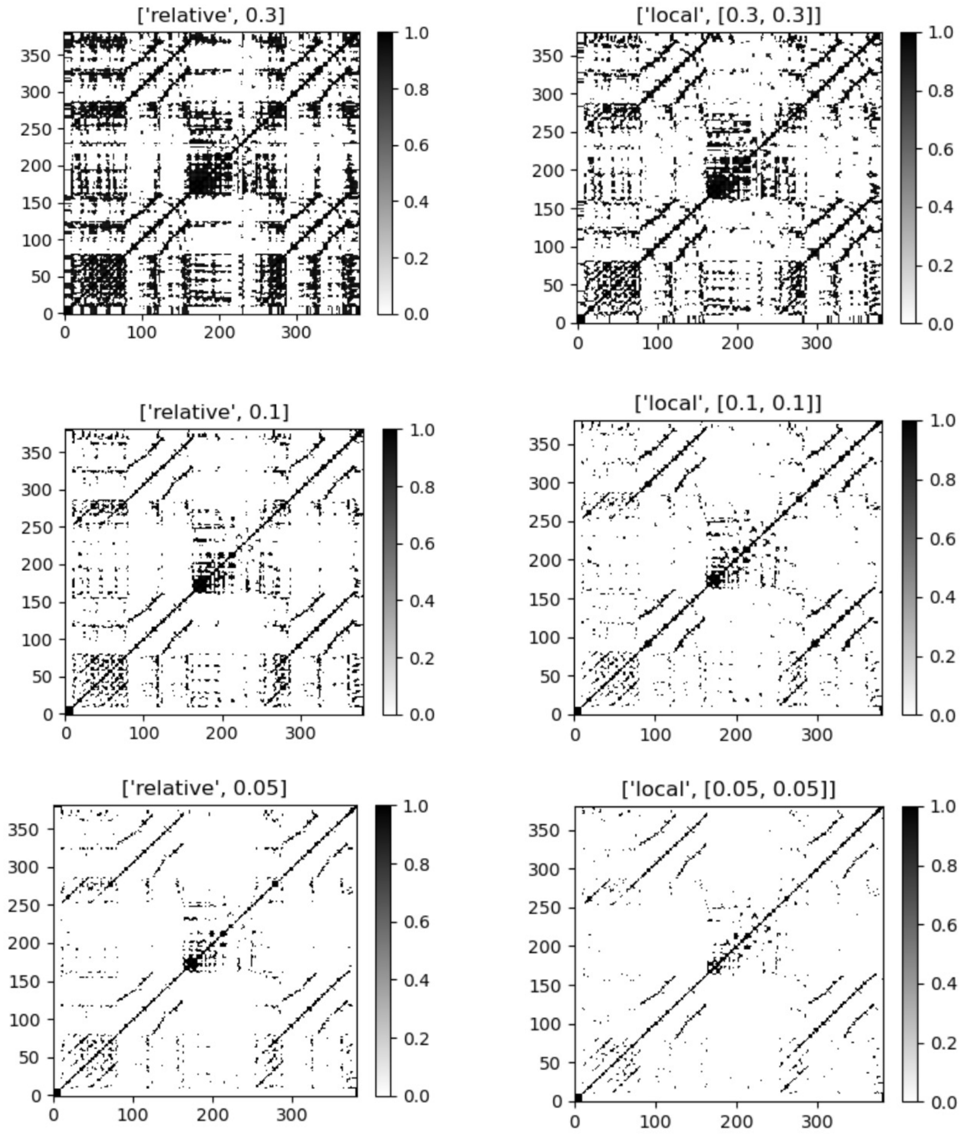
Forward-Backward Smoothing

```
def filter_diag_mult_sm(S, L=1, tempo_rel_set=np.asarray([1]), direction=0):
    N = S.shape[0]
    M = S.shape[1]
    num = len(tempo_rel_set)
    S_L_final = np.zeros((N, M))

    for s in range(0, num):
        M_ceil = int(np.ceil(M / tempo_rel_set[s]))
        resample = np.multiply(np.divide(np.arange(1, M_ceil+1), M_ceil), M)
        np.around(resample, 0, resample)
        resample = resample - 1
        index_resample = np.maximum(resample, np.zeros(len(resample))).astype(np.int64)
        S_resample = S[:, index_resample]

        S_L = np.zeros((N, M_ceil))
        S_extend_L = np.zeros((N + L, M_ceil + L))

        # Forward direction
        if direction == 0:
                S_extend_L[0:N, 0:M_ceil] = S_resample
                for pos in range(0, L):
                        S_L = S_L + S_extend_L[pos:(N + pos), pos:(M_ceil + pos)]

        # Backward direction
        if direction == 1:
                S_extend_L[L:(N+L), L:(M_ceil+L)] = S_resample
                for pos in range(0, L):
                        S_L = S_L + S_extend_L[(L-pos):(N + L - pos),
                                               (L-pos):(M_ceil + L - pos)]

        ...
```



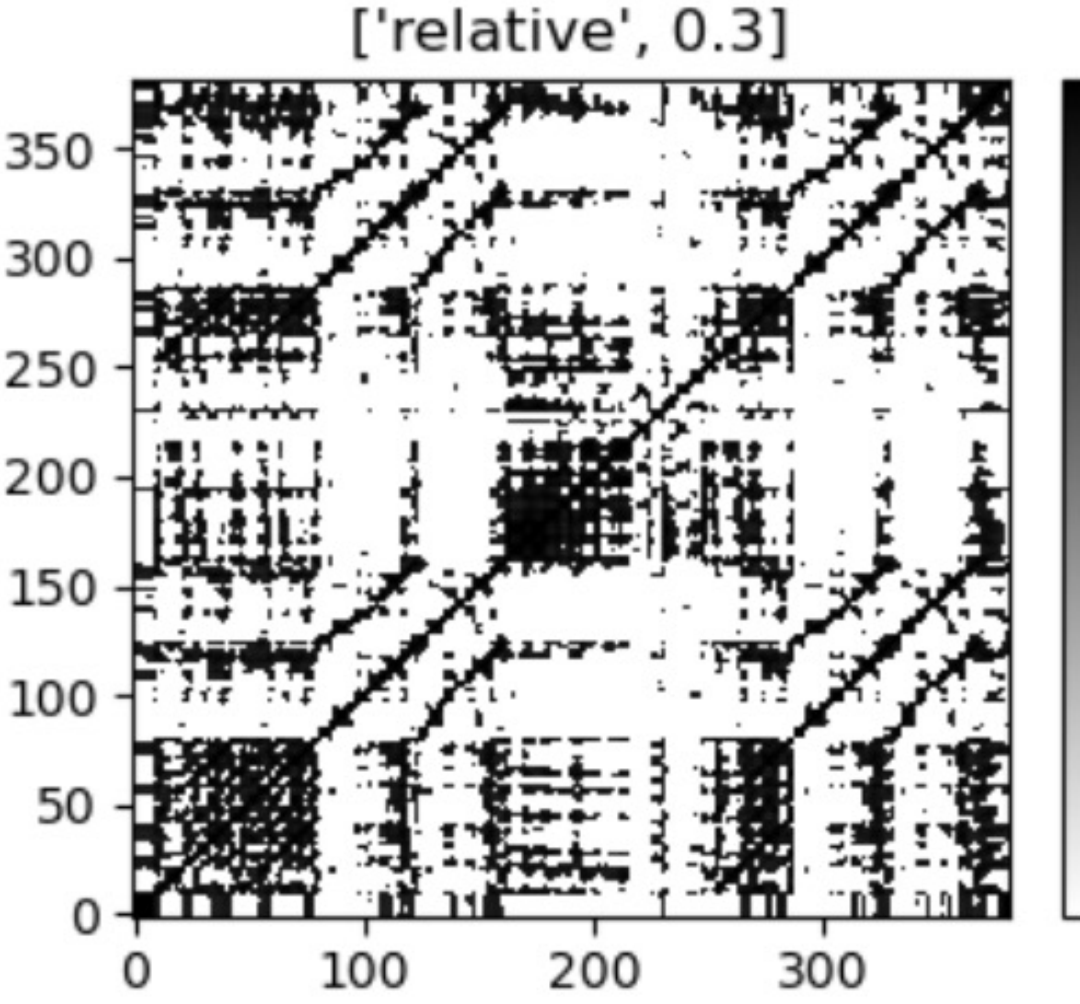
Thresholding

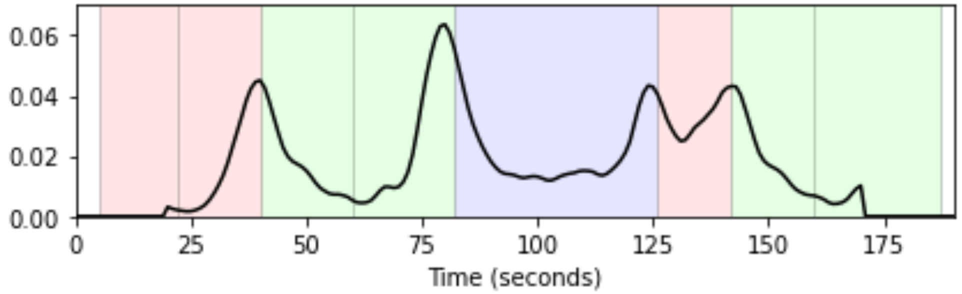
```
if strategy == 'absolute':
    thresh_abs = thresh
    S_thresh[S_thresh < thresh] = 0

if strategy == 'relative':
    thresh_rel = thresh
    num_cells_below_thresh = int(np.round(S_thresh.size*(1-thresh_rel)))
    if num_cells_below_thresh < num_cells:
        values_sorted = np.sort(S_thresh.flatten('F'))
        thresh_abs = values_sorted[num_cells_below_thresh]
        S_thresh[S_thresh < thresh_abs] = 0
    else:
        S_thresh = np.zeros([N, M])

if strategy == 'local':
    thresh_rel_row = thresh[0]
    thresh_rel_col = thresh[1]
    S_binary_row = np.zeros([N, M])
    num_cells_row_below_thresh = int(np.round(M * (1-thresh_rel_row)))
    for n in range(N):
        row = S[n, :]
        values_sorted = np.sort(row)
        if num_cells_row_below_thresh < M:
                thresh_abs = values_sorted[num_cells_row_below_thresh]
                S_binary_row[n, :] = (row >= thresh_abs)
    S_binary_col = np.zeros([N, M])
    num_cells_col_below_thresh = int(np.round(N * (1-thresh_rel_col)))
    for m in range(M):
        col = S[:, m]
        values_sorted = np.sort(col)
        if num_cells_col_below_thresh < N:
            thresh_abs = values_sorted[num_cells_col_below_thresh]
            S_binary_col[:, m] = (col >= thresh_abs)
    S_thresh = S * S_binary_row * S_binary_col
return S_thresh
```



Novelty-Based approach

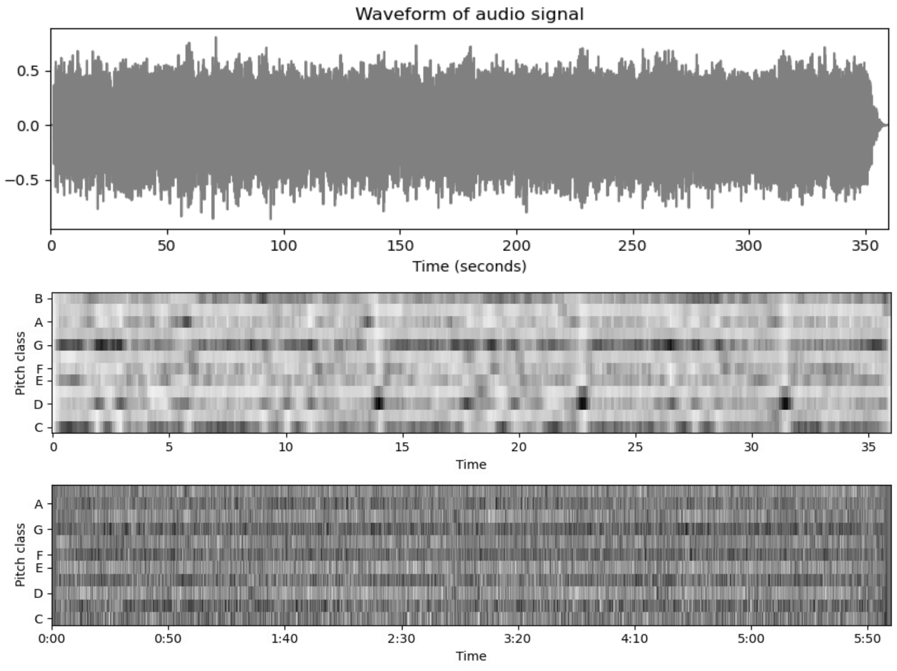
```
def compute_kernel_checkerboard_gaussian(L, var=1, normalize=True):

    taper = np.sqrt(1/2) / (L * var)
    axis = np.arange(-L, L+1)
    gaussian1D = np.exp(-taper**2 * (axis**2))
    gaussian2D = np.outer(gaussian1D, gaussian1D)
    kernel_box = np.outer(np.sign(axis), np.sign(axis))
    kernel = kernel_box * gaussian2D
    if normalize:
        kernel = kernel / np.sum(np.abs(kernel))
    return kernel

def compute_novelty_ssm(S, kernel=None, L=10, var=0.5, exclude=False):

    if kernel is None:
        kernel = compute_kernel_checkerboard_gaussian(L=L, var=var)
    N = S.shape[0]
    M = 2*L + 1
    nov = np.zeros(N)
    S_padded = np.pad(S, L, mode='constant')

    for n in range(N):
        nov[n] = np.sum(S_padded[n:n+M, n:n+M] * kernel)
    if exclude:
        right = np.min([L, N])
        left = np.max([0, N-L])
        nov[0:right] = 0
        nov[left:N] = 0

    return nov
```

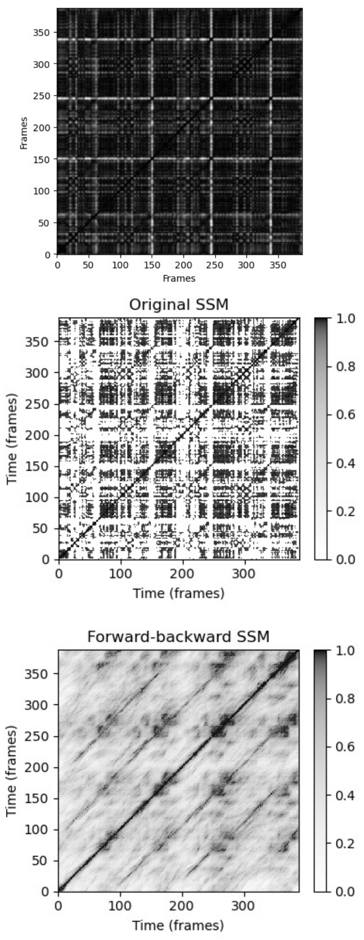


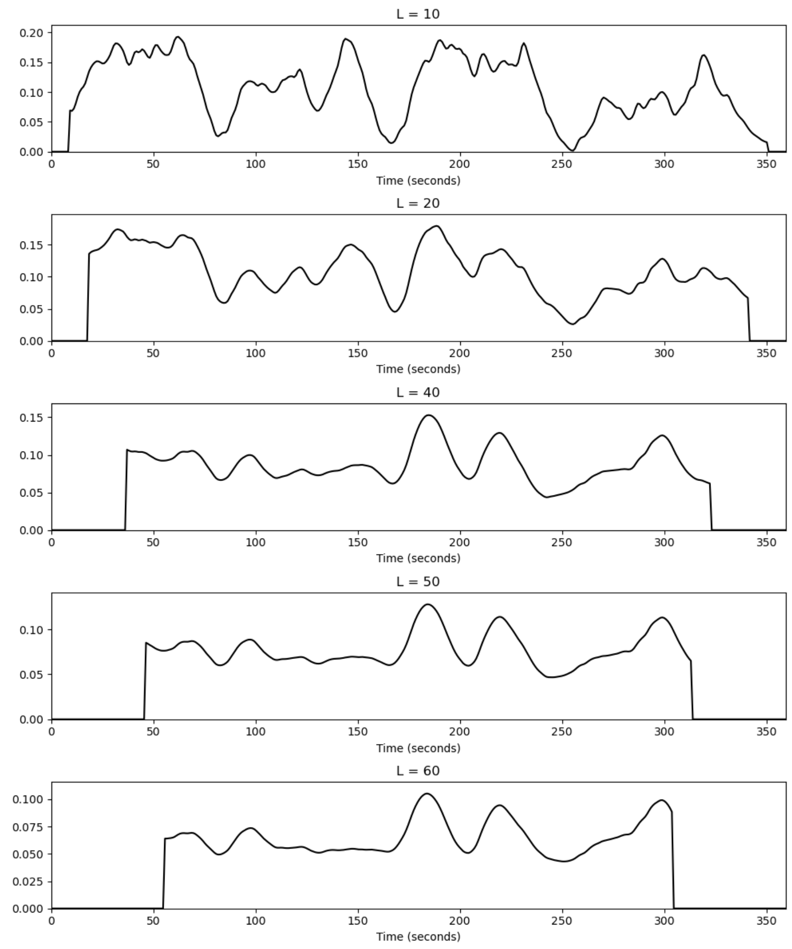


Summary

Like a Rolling Stone - *Bob Dylan*







Thank you for your attention.