An Original Implementation of Entropy and Active Information Storage based on the Fast Fourier Transform

Kilian Liss

December 5, 2020

1 Introduction

In order to do any information theoretic analysis from a time series waveform such as that of songs, a probability distribution must first be obtained. While there are various methods out there for calculating probability distributions from time series data, such as Kernel, KSG estimators [5] and the permutation entropy [3], we provide here an original method based on the Fast Fourier Transform (FFT).

In the signal processing and audio engineering scene, it is a common practice to use the FFT to decompose signals and sounds into their corresponding frequencies. When analysing songs and their various components, it would make sense to provide a probability distribution based on the notes that are played. Since musical notes are just vibrations of differing frequencies, we can analyse notes by means of their frequency distributions calculated by the FFT.

We introduce here an original method to analyse sounds and signals by information theoretic means based on the FFT, and provide an example of its usage using song waveforms obtained from a demo project of the Digital Audio Workstation named FL Studio. We then compare our approach to Bandt-Pompe's method for permutation entropy [4], which seems to be a well established method in the literature. The song under analysis is *Tevlo: Release Me (feat. Veela)* [1].

The question that we are addressing here is: *How suitable is the Fourier transform for information theoretical analysis of song components*?

1.1 The Waveform Data

The data is read from .WAV sound files that were obtained by exporting from free demo projects from the free demo version of the Digital Audio Workstation (DAW) called FL Studio. Fl Studio has a built in Mixer board with various channels where the artists can link various song components on, these may include Vocals, Percussions, Synthesisers, Bass or sound effects.

To create a dataset:

1. Download and install the free demo version of FL Studio.
2. After opening FL Studio, in the built in browser, navigate to *Demo Project* $\rightarrow$ *Demo Songs* $\rightarrow$ *Tevlo - Release Me (feat. Veela)* or any other demo project.
3. Click on *File* $\rightarrow$ *Export* $\rightarrow$ *Wave File*.
4. In the pop-up menu, navigate to where the code is located, create a new folder called *Songs*, and create a new subfolder with the song name, and navigate to it. Make sure the File Name is left as just *.wav*, and click *Save*.
5. In the new pop-up window, make sure *Split Mixer Tracks* is selected and the mode is in *Full Song*. Click *Start*, and FL Studio will begin exporting the song files to be used in our code.

A new .WAV file will be created for each channel in the mixer board, and the filenames will correspond to the channel name that the artist gave to the corresponding channel. Each file will contain the sound output of its channel for the entirety of the song. Figure 2a plots the waveforms of each channel.

2 Methods

2.1 Fast Fourier Transform (FFT)

For a discrete variable $X = \{x_1, ..., x_N\}$, the Fourier transform may be calculated by:

$$fft(k) = \sum_{n=0}^{N-1} x_n e^{-2\pi j \frac{kn}{N}} \qquad k \in [0, N-1] \tag{1}$$

where j is the imaginary unit.

By applying the FFT to a sliding window of the channel waveforms and then normalizing them, we can create a probability distribution of frequencies as a function of time for the various instruments in the song. In our implementation, the FFT will be calculated by simply calling the python function from the Scipy library *scipy.fft.fft()*, which returns the real component of the FFT.

2.2 Normalizing the Fourier Transform

Since the frequencies of the waveforms under analysis correspond to musical notes, we would like to treat each musical note equally rather than treating each frequency equally. To do so, we scale the x-axis of the FFT by a scale of $\log_{10}$

seems to be a standard in the music audio engineering industry[1]. However, this introduces a slight complication in normalizing our FFT, which is demonstrated in figure 1.

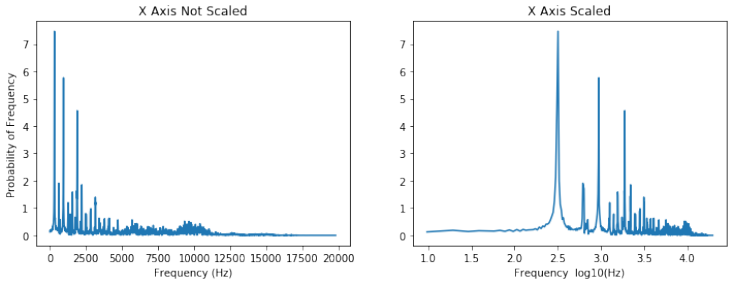


Figure 1: Comparing the scaled and unscaled Fourier transform. Obtained from the *Vox* (vocals) channel at 20 seconds into the song. These plots here have been normalized using the trapezoidal rule.

In the scaled FFT plot in figure 1, the FFT is discrete with values on the right hand side are spaced much closer to each other compared to those in the left hand side. Therefore we must use a numerical integration scheme such as the trapezoidal rule in order to find our normalization constant C, given by:

$$C = \sum_{k=1}^{N} \frac{fft(x_{k-1}) + fft(x_k)}{2} \Delta x_k \tag{2}$$

and our probability distributions need to be scaled similarly when calculating information theoretic measures.

2.3 Fourier Entropy

Entropy is a measure of the amount of disorder, which in our context of analysing song components, is a measure of how evenly spread out the frequency spectrum of a song channel is.

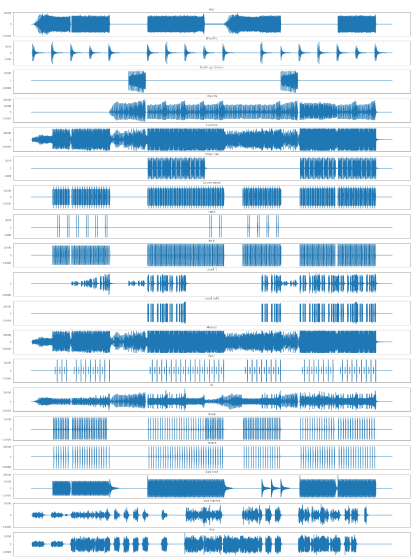
One we have the probabilities based on the normalized Fourier transforms, we can use them to calculate the entropy as a function of time using a sliding window approach. The waveforms are split into snipplets of the size of the predefined sliding window, and for each snipplet the entropy is calculated as:

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i) \tag{3}$$

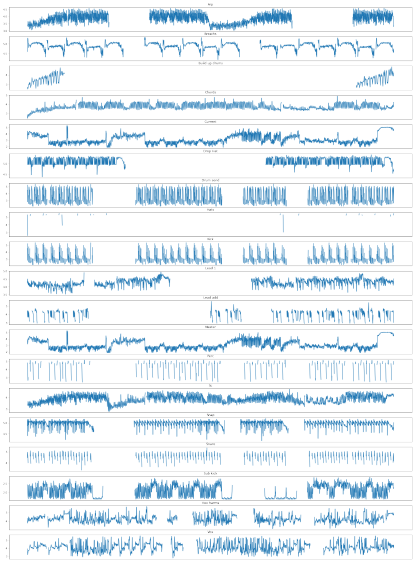
where $p(x)$ is the fourier transform but weighted based on the trapezoidal rule:

$$p(x) = \frac{fft(x_{k-1}) + fft(x_k)}{2} \Delta x_i \tag{4}$$

[1]Based on the fact that all *Parametric Equilizers* in a quick google search tend to use a log_{10} scale

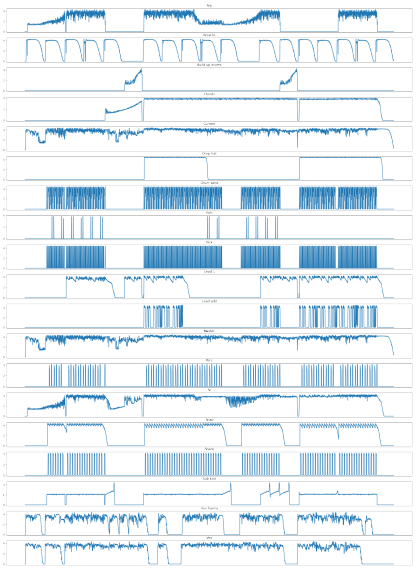


(a) The raw waveform data after exporting from FL Studio.



(b) The fourier entropy calculated from all channel waveforms.

(c) The active information calculated from all channel waveforms.



(d) The permutation entropies calculated from all channel waveforms.

Figure 2: Various time series plots. Larger versions are provided at the bottom of this report

2.4 Active Information Storage

The active information storage is given by:

$$A_x(k) = \langle a_x(k) \rangle = \left\langle \log_2 \frac{p(x_{n+1}|\mathbf{x}_n^{(k)})}{p(x_{n+1})} \right\rangle \tag{5}$$

where once again, our probabilities were normalized based on the trapezoidal rule. To calculate the probability term, we used:

$$p(x_{n+1}|\mathbf{x}_n^{(k)}) = \frac{p(x_{n+1}, \mathbf{x}_n^{(k)})}{p(\mathbf{x}_n^{(k)})} \tag{6}$$

To evaluate $p(x_{n+1}, \mathbf{x}_n^{(k)})$, the waveform was cut such that all the past windows were included along with the current window, and the Fourier transform was applied as usual. Similarly, to evaluate $p(\mathbf{x}_n^{(k)})$, the waveform was cut so that only the past windows were included. $p(x_{n+1})$ was calculated similarly, and to make sure that all probabilities are stored in arrays of the same size, a cubic interpolation was used. Again, all probabilities were weighted based on the trapezoidal rule.

2.5 Permutation Entropy

To provide a comparison of how the Fourier entropy performs, we provide the permutation entropy, which is a well established measure in the literature [3]. To calculate the permutation entropy, we take a time series $\chi = \{x_t : t = 1, ..., N\}$, then $\forall x \in \chi$ and a given embedding dimension D, a set of D dimensional vectors are constructed as follows:

$$s \to (x_s, x_{s+1}, ..., x_{s+(D-1)}) \tag{7}$$

and a set of permutations $\pi = (r_0, r_1, ..., r_{D-1})$ which fulfil:

$$x_{s+r_0} \leq x_{s+r_1} \leq ..... \leq x_{s+r_{D-1}} \tag{8}$$

is formed. The occurrences of the permutations π are then counted and are used to form a probability distribution of permutations $p(\pi)$ occuring in our sliding window. The permuation entropy may then be calculated as:

$$\mathcal{S}[P] = -\sum_{i=1}^{M} p_i \ln p_i \tag{9}$$

3 Results

3.1 Fourier Entropy

All our results were calculated using a sliding window of 2^{12}. Figure 2b shows the entropy as a function of time for all the channel waveforms in our song. We

note that the ends of the plots are clipped if the channel has a flat waveform at either beginning or end of the song.

Figure 3: Analysing the *Sub kick* channel and the *Vox* (Vocals) channel for comparisongs of entropies. First plot shows their raw waveforms, the second plot shows their entropy, the third shows their fourier transforms at 50 seconds into the song, and the fourth plot is a zoom into the waveform.

Figure 3 compares the entropy of the *Sub kick* (Blue) and *Vox* (Orange) waveforms, and how their corresponding Fourier transforms produce different levels of entropy.

The *Sub kick* is a low bass instrument, and at 50 seconds into the song its waveform resembles that of a single frequency sine wave. Its Fourier transform confirms that its frequency is concentrated around a single value, and since it's expected frequency has a low level of surprise, we measure it as low entropy.

Meanwhile the *Vox* channel is a female singer with a high pitch voice. From her Fourier transform we can see that her voice has a much higher range of frequencies, making her frequency distribution much more chaotic, giving her a higher entropy compared to that of the *Sub kick*.

The *Sub kick* can generally be split into three types of regions which are marked in figure 3 at times 50, 100 and 170 seconds into the song by red lines. At 100 seconds into the song, the *Sub kick* is beating frequently (6 times a bar) at a lower pitch, since the sound of the *Sub kick* has an initial punch to it with treble apparent in the punch, its entropy time series appears to be more chaotic. At 50 seconds into the song, the *Sub kick* is at the same pitch, but beating significantly less frequent (once every 4 bars), this produces the sine wave looking waveform as shown in figure 3 which has a consistently low entropy. At 170 seconds into the song, the *Sub kick* is up in pitch, which widens it's frequency distribution and therefore increases the entropy.

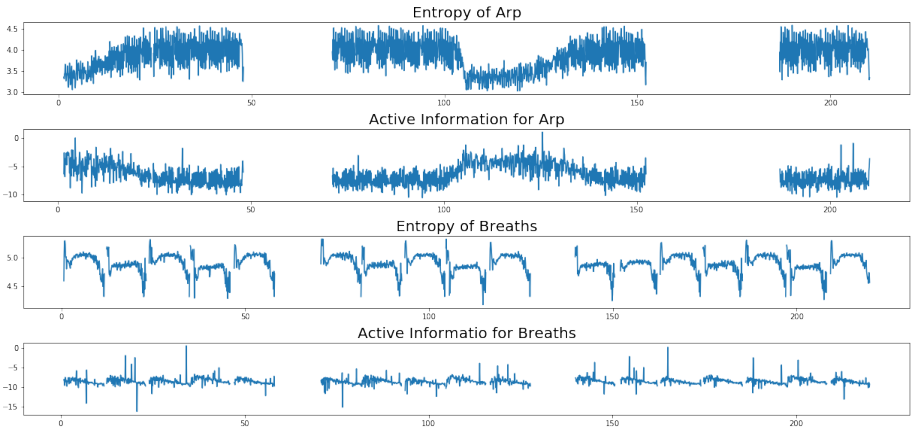


Figure 4: Comparing the entropy and active information of the *Arp* and *Breaths* channels

3.2 Active Information Storage

Figure 2c shows the calculated active information storage for all channels in the song. A history length of 1 was chosen for the blue coloured lines, and a history length of 6 was chosen for the orange coloured lines. Generally, the active information storage for all channels seem to be negative for the most part. This seems to suggest that the frequency distribution is constantly and rapidly changing, which is to be expected for songs.

We can see that the *Vox* channel has a significant lower information storage compared to that of the *Sub kick*, which seems confirm the chaotic nature of the frequency distribution that vocals have over a bass generated by sine waves, which we have already discussed when talking about the entropy of these two channels.

Figure 4 provides a comparison of the entropy and active information storage with history of 6 for the *Arp* and *Breaths* channels. What is interesting here is that as the entropy for the *Arp* increases, the information storage decreases, which tends to be the case for most channels in the song. However, the *Breaths* seems to break that rule, which suggests that even though the breaths channel is chaotic in it's frequency distribution, its distribution seems to be somewhat constant in time.

3.3 Comparing Fourier Entropy with the Permutation Entropy

Figure 2d shows the permutation entropy of all channels, and is provided as a comparison for our Fourier entropy. We note that for the majority of the waveforms, our fourier entropy seems resemble that of the permutation entropy, but with significantly more noise, even the amplitudes of the entropies seem to

be in the same scale, which is at around 4 bits.

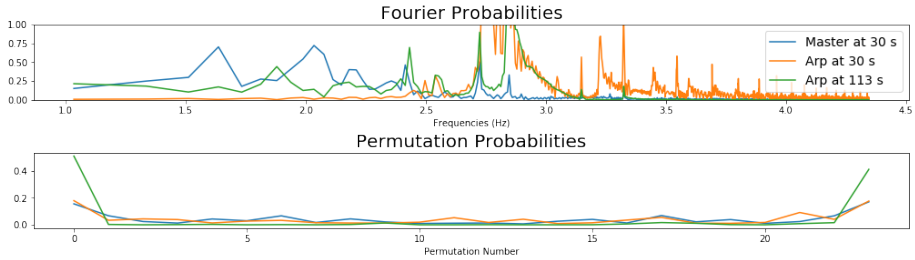


Figure 5: Comparing Fourier probability and Permutation probabilities

Figure 5 shows the Fourier and Permutation probabilities for the *Arp* channel at 30 seconds (High Entropy and Loud Volume) and 113 seconds (Low Entropy and Low Volume), which is a channel where the two entropy measures resemble each other well. At low volume, less frequencies of the *Arp* are audible, hence giving a lower entropy compared to a loud *Arp*.

However in the *Master* channel, the Fourier entropy fails to replicate the results obtained from the permutation entropy. The *Master* channel is the combination of all other channels and is the waveform of the final song. Figure 5 shows the permutation and fourier probabilities of the *Master* channel. We can see that even though the treble is visible in the *Master* channel, it is dominated by the bass frequencies, so the fourier entropy is low because the frequency distribution is so skewed to the left. Meanwhile the permutation probability is not affected by the amplitude of frequencies, and therefore produces a higher entropy due to the more uniform probability distribution.

Since the entropy is meant to be a measure of chaos, our Fourier entropy fails to represent that in the case of the *Master* channel, where all the instruments have been combined. In future work, one could attempt to weigh the frequencies such that the entropy is less dependent on the frequency amplitudes.

It is also noteworthy that the code for the Fourier entropy runs significantly faster than that of the permutation entropy, especially when a high embedding dimension is chosen. This is because the Fourier entropy is based on the Fast Fourier Transform, and since Fast is in the name of the algorithm, it's gotta be **FAST**!!!!!

4 Discussion

We have successfully managed to design and implement a method for calculating the entropy and active information storage based on the FFT. Although its results were found to be more noisy compared to that obtained by permutations, it seems to run significantly faster. While it doesn't quite perfectly capture the intent of a entropy measure (the *Master* channel's entropy shouldn't be so low), there are some potential ideas that could be used to improve the measure.

Firstly, when we normalize our Fourier entropy, we are losing information about the volume of the song. Secondly, after writing the majority of this report, I later discovered that the Fourier Entropy has already been used in the literature, and in their implementation [6] they don't scale the x-axis like we did. Figure 10 shows the fourier entropy with the normalization and scaling of the x-axis removed from the calculation. At first glance the plots appear to be a better measure, however the values obtained seem to be too much dominated by the volume of the channel.

Our implementation of the active information storage could have been improved in much the same way the Fourier entropy could be improved. Although we have not compared it to other implementations, we can still see it being somewhat noisy. Additionally, it would have been nice to implement the Transfer Entropy based on the Fourier probabilities, which was my original goal, but I am running low on word count and time for this report, and it is more challenging than I thought.

References

[1] *Tevlo — Release Me (feat. Veela)* https://soundcloud.com/fl-studio/tevlo-release-me-feat-veela

[2] FL Studio *https://www.image-line.com/* Visited on: 2020/11/26

[3] Massimiliano Zanin, Luciano Zunino, Osvaldo A. Rosso and David Papo *Permutation Entropy and Its Main Biomedical and Econophysics Applications: A Review* Entropy 2012, 14, 1553-1577; doi: 10.3390/e14081553

[4] Traversaro,Francisco and Redelico,Francisco O. and Risk,Marcelo R. and Frery,Alejandro C. and Rosso,Osvaldo A. *Bandt-Pompe symbolization dynamics for time series with tied values: A data-driven approach* Chaos: An Interdisciplinary Journal of Nonlinear Science, 28, 7

[5] J.T. Lizier, *JIDT: An information-theoretic toolkit for studying the dynamics of complex systems*, Frontiers in Robotics and AI, 1:11, 2014

[6] Sourav Chakraborty and Raghav Kulkarni and Satyanarayana V. Lokam and Nitin Saurabh *Upper Bounds on Fourier Entropy* Theoretical Computer Science, 654, 92 - 112

Final Code

November 30, 2020

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pathlib
from scipy.fft import fft
from scipy.io.wavfile import read
from itertools import permutations
from scipy.interpolate import interp1d

import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)
```

Load the .wav files and store them into a dictionary as numpy arrays.

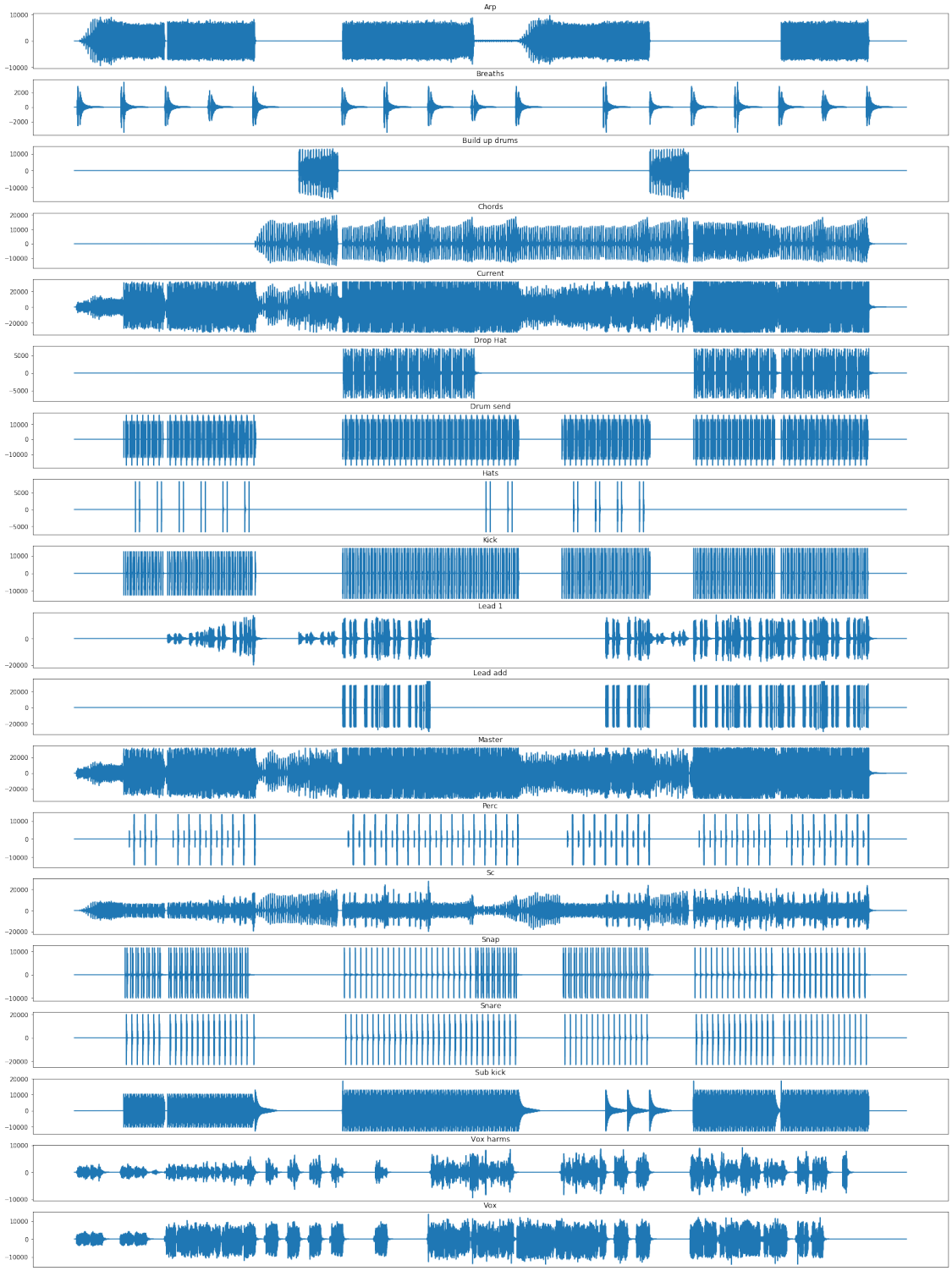
[2]:
```
#Load the wave files
#songPathString = './Songs/Lollivox - Optimum Momentum'
songPathString = './Songs/Tevlo - Release Me (feat. Veela)'
songPath = pathlib.Path(songPathString)
channels = {}
for channel in list(songPath.glob('./*.wav')):
    channelName = str(channel)[len(songPathString):-4]
    channel = read(channel)
    channel = np.array(channel[1])
    channel = channel[:, 0]
    channels[channelName] = channel
channelNames = list(channels.keys())

#Some parameters that will be used throughout the code
songLength = 3 * 60 + 40
windowSize = 2**12
samplingFrequency = len(channels['Master']) / songLength
time = np.arange(0, songLength, 1/samplingFrequency)

#Print the channel names as a reference
print(channelNames)
```

```
C:\Users\BGH360\AppData\Roaming\Python\Python37\site-
packages\ipykernel_launcher.py:8: WavFileWarning: Chunk (non-data) not
understood, skipping it.
```

```
['Arp', 'Breaths', 'Build up drums', 'Chords', 'Current', 'Drop Hat', 'Drum
send', 'Hats', 'Kick', 'Lead 1', 'Lead add', 'Master', 'Perc', 'Sc', 'Snap',
'Snare', 'Sub kick', 'Vox harms', 'Vox']
```

We plot the raw song files to help visualize the original data

[3]:
```
fig, axs = plt.subplots(len(channelNames))
fig.set_figheight(2 * len(channelNames))
fig.set_figwidth(27)
for i, name in enumerate(channelNames):
    axs[i].plot(channels[name])
    axs[i].set_title(name)
    axs[i].set_xticks([])
```



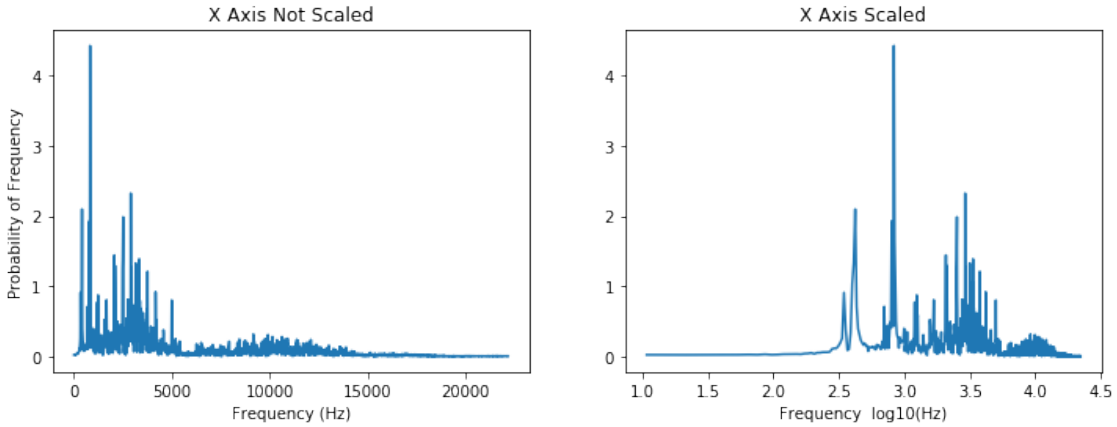
Get the fourier transform given a time series

[4]:
```
def getFourierFromTimeSeries(series):

    #Calculate the fourier transform for a snipplet of the song
    fourierTrans = fft(series)[0:len(series)//2]
    fourierTrans = np.abs(fourierTrans)

    #Get the frequencies corresponding to the fourier transform that will be␣
 ↪used for the x-axis
    frequency = np.linspace(0, samplingFrequency/2, len(fourierTrans))
    frequency = np.log10(frequency)
    frequencyIsInfinity = ~ np.isinf(frequency)
    frequency = frequency[frequencyIsInfinity]

    #Normalize the fourier transform and return values
    fourierTrans = fourierTrans[frequencyIsInfinity]
    trapezoidal = 2 * (frequency[1:] - frequency[:-1]) * ((fourierTrans[1:] +␣
 ↪fourierTrans[:-1]) / 2)
    fourierTrans = fourierTrans / np.sum(trapezoidal)
    return frequency, fourierTrans

#An old function that was in the code and I can't be bothered fixing them
def getFourierTransformAtIndex(series, idx):
    snipplet = series[idx:idx+windowSize]
    frequency, fourierTrans = getFourierFromTimeSeries(snipplet)
    return frequency, fourierTrans
```

We provide a comparison of the x axis being scaled and not scaled.

[5]:
```
#Get the data to plot
sampleAt = int(20 * samplingFrequency)
snipplet = channel[sampleAt : sampleAt+windowSize]
x, transform = getFourierFromTimeSeries(snipplet)
xNotScaled = np.linspace(0, samplingFrequency/2, windowSize//2)

#Plot the data and format plots
fig, axs = plt.subplots(1, 2)
fig.set_figheight(4)
fig.set_figwidth(12)
axs[0].plot(xNotScaled[1:], transform)
axs[0].set_title('X Axis Not Scaled')
axs[0].set_xlabel('Frequency (Hz)')
axs[0].set_ylabel('Probability of Frequency')
axs[1].plot(x, transform)
axs[1].set_title('X Axis Scaled')
axs[1].set_xlabel('Frequency  log10(Hz)')
```

[5]:
```
Text(0.5, 0, 'Frequency  log10(Hz)')
```

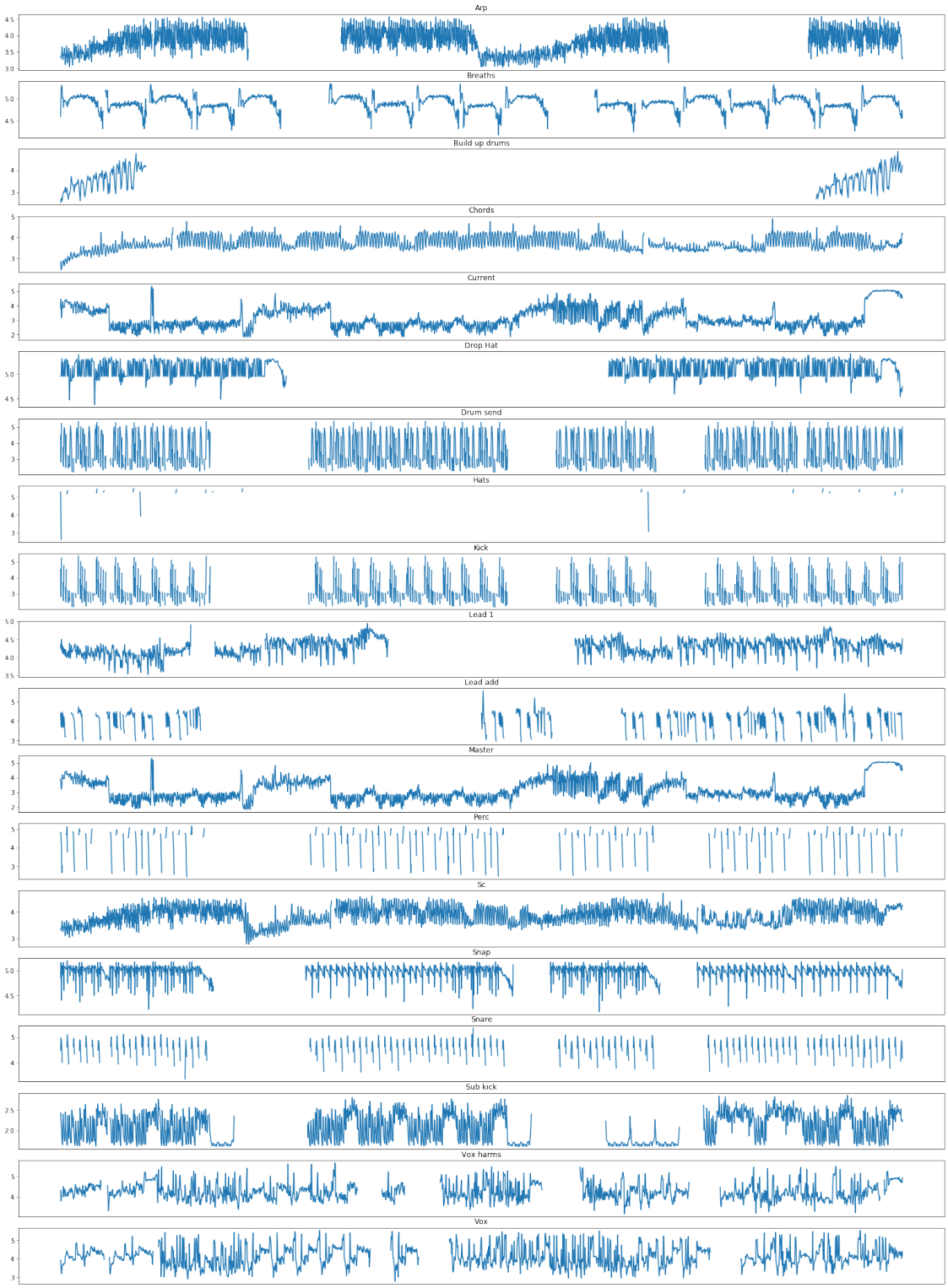


[6]:
```
def getFourierEntropy(channel):

    #We iterate over the whole channel waveform and store results from sliding␣
↪window as lists
    time, entropies = [], []
    for i in range(len(channel) // windowSize):
        t = i * windowSize / samplingFrequency
        time.append(t)

        #Get the fourier transform for this sliding window
        idx = i * windowSize
        snipplet = channel[idx:idx+windowSize]
        freq, transform = getFourierFromTimeSeries(snipplet)

        #We weigh the transform based on the trapezoidal rule and calculate the␣
↪entropy for this window
        weightedTransform =  (freq[1:] - freq[:-1]) * (transform[1:] +␣
↪transform[:-1]) / 2
        localEntropy = weightedTransform * np.log2(weightedTransform)
        entropy = - np.sum(localEntropy)
        entropies.append(entropy)
    return time, entropies
```

We plot the entropy for all channels in the song

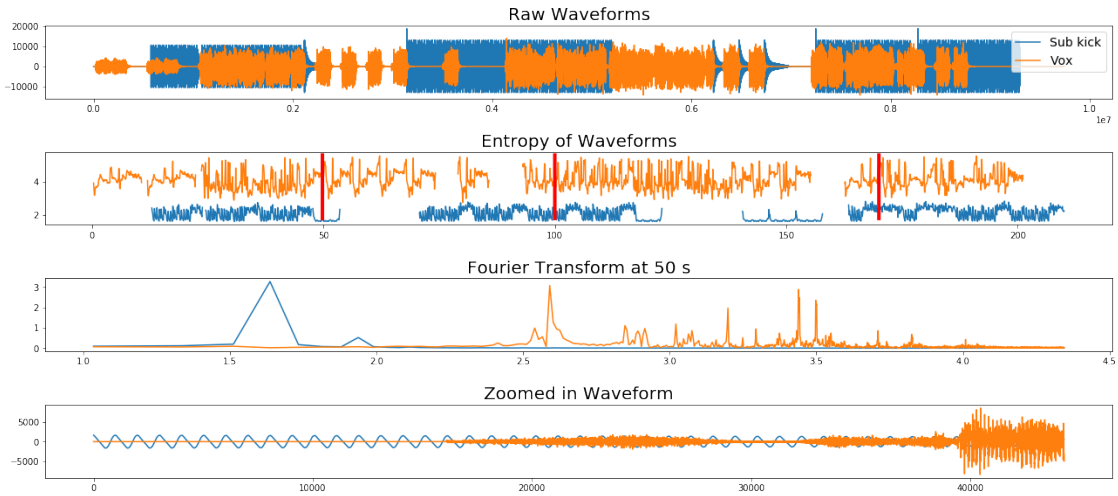
[7]:
```
fig, axs = plt.subplots(len(channelNames))
fig.set_figheight(2 * len(channelNames))
fig.set_figwidth(27)
for i, name in enumerate(channelNames):
    t, ents = getFourierEntropy(channels[name])
    axs[i].plot(t, ents)
    axs[i].set_title(name)
```

```
    axs[i].set_xticks([])
```



More plots that are used in our report

[8]:
```
chan1 = 'Sub kick'
chan2 = 'Vox'
time = 49.8

tMaster, entMaster = getFourierEntropy(channels[chan1])
tVocals, entVocals = getFourierEntropy(channels[chan2])
freq, fourierMaster = getFourierTransformAtIndex(channels[chan1], int(time *␣
 ↪samplingFrequency))
freq, fourierVocals = getFourierTransformAtIndex(channels[chan2], int(time *␣
 ↪samplingFrequency))

fig, axs = plt.subplots(4)
fig.set_figheight(8)
fig.set_figwidth(18)
axs[0].plot(channels[chan1])
axs[0].plot(channels[chan2])
axs[1].plot(tMaster, entMaster)
axs[1].plot(tVocals, entVocals)
axs[2].plot(freq, fourierMaster)
axs[2].plot(freq, fourierVocals)
axs[3].plot(channels[chan1][int(time * samplingFrequency) : int((time+1) *␣
 ↪samplingFrequency)])
axs[3].plot(channels[chan2][int(time * samplingFrequency) : int((time+1) *␣
 ↪samplingFrequency)])
#axs[2].set_ylim((0, 1))

axs[0].set_title('Raw Waveforms', fontsize=20)
axs[1].set_title('Entropy of Waveforms', fontsize=20)
axs[2].set_title('Fourier Transform at 50 s', fontsize=20)
axs[3].set_title('Zoomed in Waveform', fontsize=20)
axs[0].legend([chan1, chan2], prop={'size': 14}, loc=1)
axs[1].plot([time, time + 0.001], [1.8, 5.6], color='r', linewidth=4)
axs[1].plot([100, 100 + 0.001], [1.8, 5.6], color='r', linewidth=4)
axs[1].plot([170, 170 + 0.001], [1.8, 5.6], color='r', linewidth=4)
fig.tight_layout(pad=1.0)
```



We write a function for calculating the active information storage of a channel.

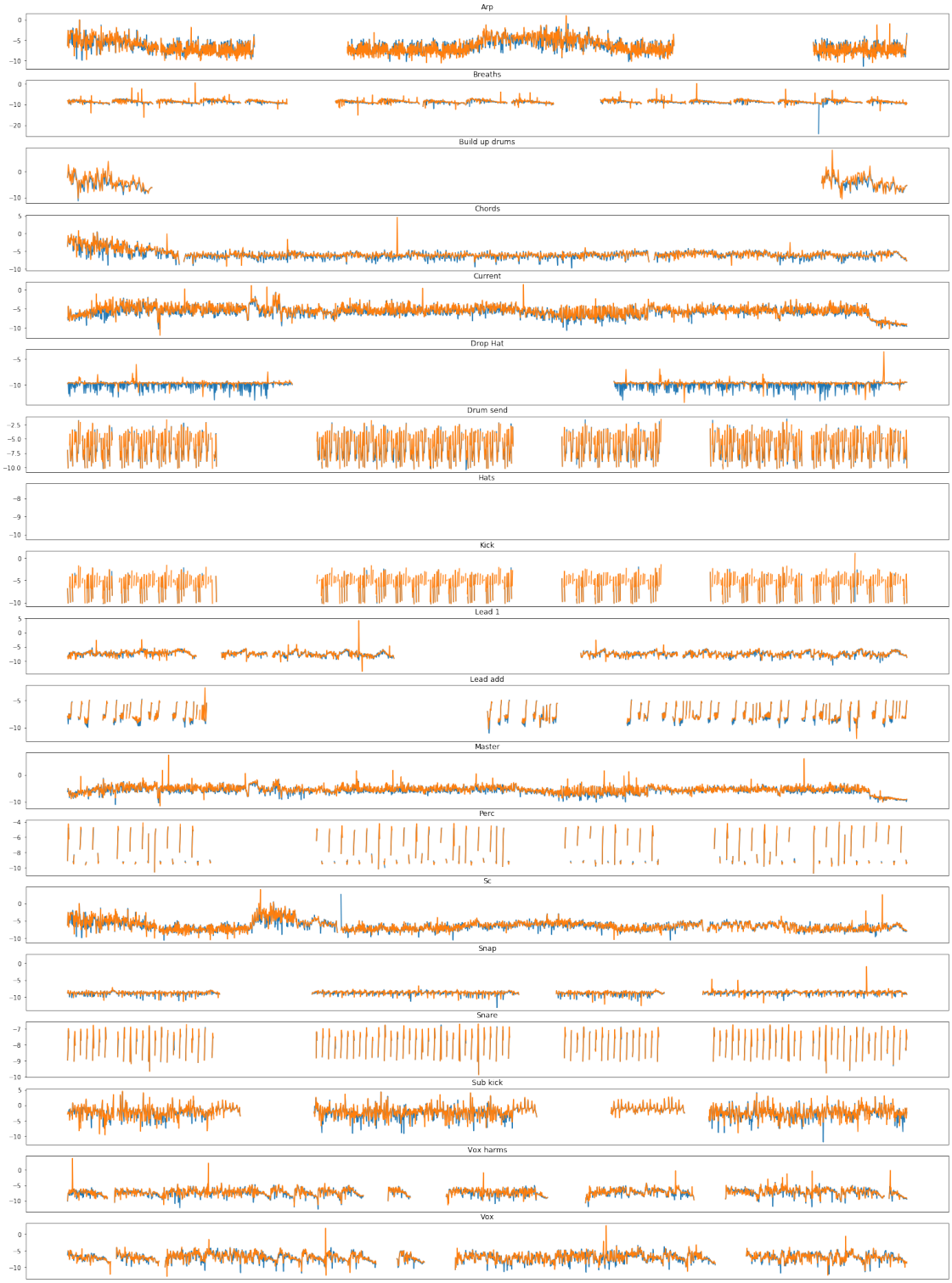
[9]:
```
def getInformationStorage(channel, history):

    #We iterate over the whole channel waveform and store results from sliding␣
↪window as lists
    time, infoStorage = [], []
    for i in range(history+1, len(channel) // windowSize):
        t = i * windowSize / samplingFrequency
        time.append(t)

        #Get the snipplets that we will use to calculate probabilities from
        currentAndPast = channel[(i - history - 1) * windowSize : i *␣
↪windowSize]
        current = channel[(i - 1) * windowSize : i * windowSize]
        past = channel[(i - history - 1) * windowSize : (i - 1) * windowSize]

        #Get the probabilities
        freq, probsCurrentAndPast = getFourierFromTimeSeries(currentAndPast)
        freqCurrent, probsCurrent = getFourierFromTimeSeries(current)
        freqPast, probsPast = getFourierFromTimeSeries(past)

        #Interpolate the arrays to make sure they all have the same length
        interpCandP = interp1d(freq, probsCurrentAndPast, kind='cubic')
        interpP = interp1d(freqPast, probsPast, kind='cubic')
        pastAndCurInterpolated = interpCandP(freqCurrent)
        pastInterpolated = interpP(freqCurrent)

        #Calculate the fraction term that is in the local information storage
        probsNowGivenPast = pastAndCurInterpolated / pastInterpolated
```

```
        normalizationConstant = (freqCurrent[1:] - freqCurrent[:-1]) *␣
↪(probsNowGivenPast[1:] + probsNowGivenPast[:-1]) / 2
        probsNowGivenPast = probsNowGivenPast / np.sum(normalizationConstant)
        fractionTerm = probsNowGivenPast / probsCurrent

        #Weigh the fraction term and remove nan values
        weightedFract = (freqCurrent[1:] - freqCurrent[:-1]) * (fractionTerm[1:
↪] + fractionTerm[:-1]) / 2
        localActiveInfo = np.log2(weightedFract)
        localActiveInfo = localActiveInfo[~np.isnan(localActiveInfo)]

        #Append to our results
        infoStorage.append(np.mean(localActiveInfo))
    return time, infoStorage
```

We calculate the information storage for all song and plot them.

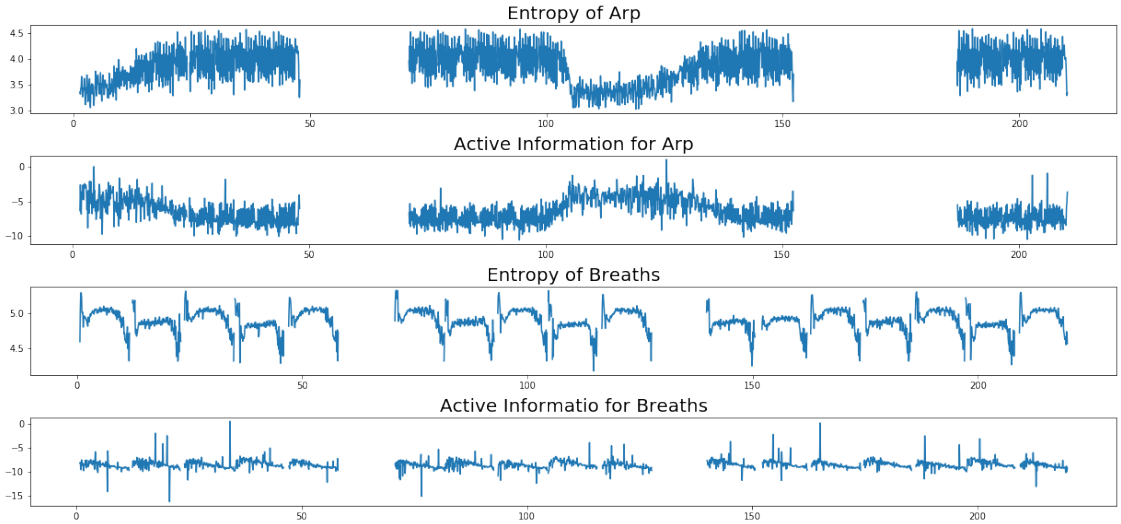
[10]:
```
infoStoresTwo, infoStoresSix = [], []
for i, name in enumerate(channelNames):
    tTwo, info = getInformationStorage(channels[name], 1)
    infoStoresTwo.append(info)
    tSix, info = getInformationStorage(channels[name], 6)
    infoStoresSix.append(info)
```

[11]:
```
fig, axs = plt.subplots(len(channelNames))
fig.set_figheight(2 * len(channelNames))
fig.set_figwidth(27)
for i, info in enumerate(infoStoresTwo):
    axs[i].plot(tTwo, info)
    axs[i].plot(tSix, infoStoresSix[i])
    axs[i].set_title(channelNames[i])
    axs[i].set_xticks([])
```



[12]:
```
chan1 = 'Arp'
chan2 = 'Breaths'
```

```
tArp, entArp = getFourierEntropy(channels[chan1])
tBreaths, entBreaths = getFourierEntropy(channels[chan2])
tA, inforArp = getInformationStorage(channels[chan1], 6)
tB, inforBreaths = getInformationStorage(channels[chan2], 6)

fig, axs = plt.subplots(4)
fig.set_figheight(8)
fig.set_figwidth(17)
axs[0].plot(tArp, entArp)
axs[1].plot(tA, inforArp)
axs[2].plot(tBreaths, entBreaths)
axs[3].plot(tB, inforBreaths)

axs[0].set_title('Entropy of Arp', fontsize=20)
axs[1].set_title('Active Information for Arp', fontsize=20)
axs[2].set_title('Entropy of Breaths', fontsize=20)
axs[3].set_title('Active Informatio for Breaths', fontsize=20)
fig.tight_layout(pad=0.8)
```



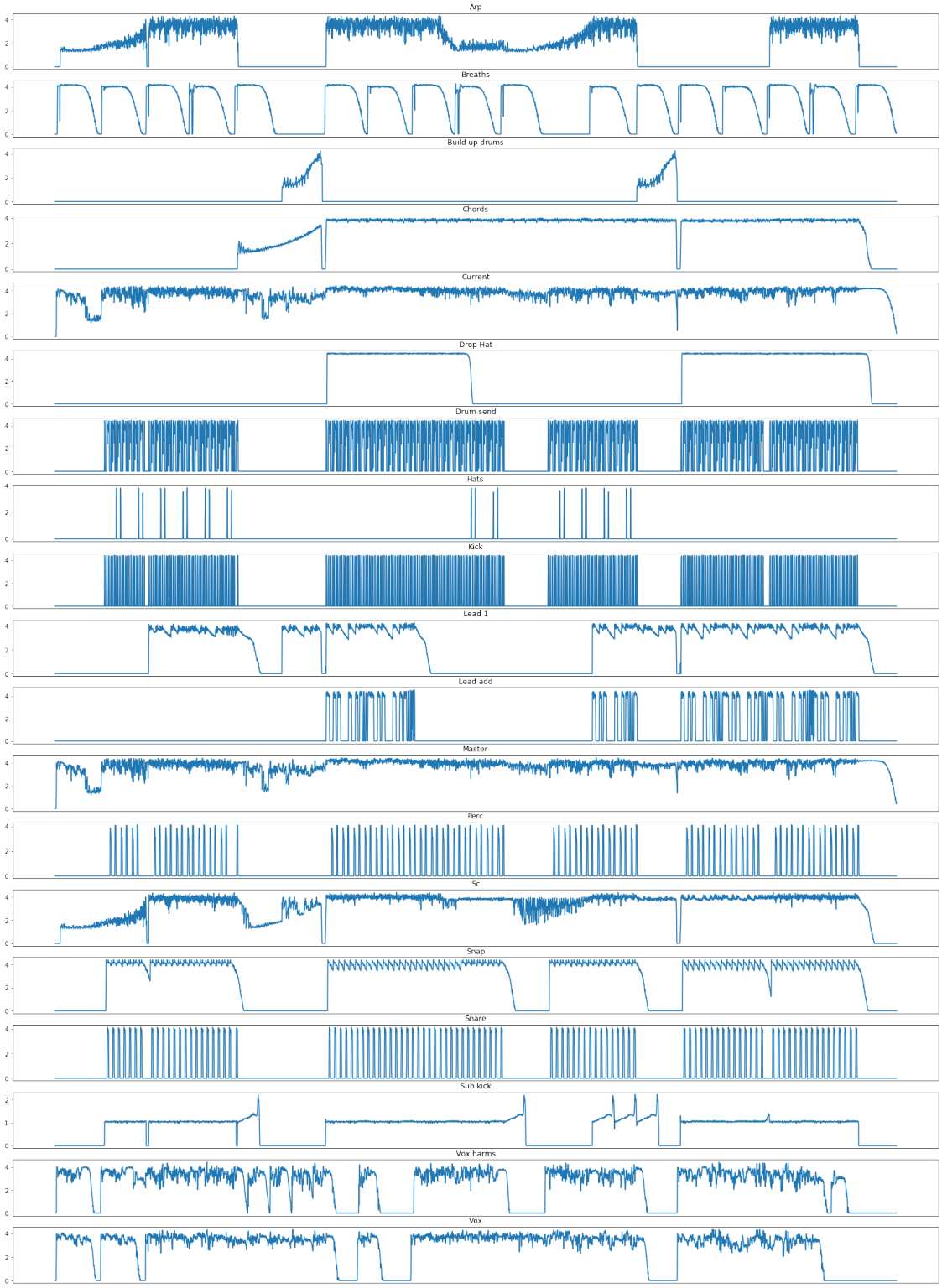
We define functions for calculating the permutation entropy.

[13]:
```
#Get the permutation entropy of a single window
def permutationEntropy(X, embedingDimension):

    #Create a list of all possible permutations, and a list to store the counts␣
↪of each permutation in our time series
    perms = list(permutations(list(range(embedingDimension))))
    permCounts = np.zeros(len(perms))
```

```
    #Slide over the snipplet to count the permutations
    iterations = X.shape[0] - embedingDimension + 1
    for i in range(iterations):
        subsequents = X[i : i+embedingDimension]
        permutation = tuple(np.argsort(subsequents))
        idx = perms.index(permutation)
        permCounts[idx] += 1

    #Remove zeroes and calculate the permutation entropy
    isZero = (permCounts == 0)
    permCounts += 0.0001 * isZero
    permProb = permCounts / sum(permCounts)
    permEntropy = - np.sum(permProb * np.log2(permProb) * (1-isZero))
    return permEntropy, permProb

#Gets a list of normalized permutation entropies of a time series X using a␣
 ↪sliding window approach
def getPermuationEntropies(X, embeddingDimension, window, delay):

    #Filter out values in X with accordance to the embedding delay
    X = X[(np.array(range(len(X))) % delay == 0)]

    #Use a sliding window to cut the time series into snipplets and calculate␣
 ↪their permutation entropies
    permutationEntropies = []
    iterations = (len(X)//window) + 1
    for i in range(iterations):
        x = X[i*window : (i+1)*window]
        permEnt, permProb = permutationEntropy(x, embeddingDimension)
        permutationEntropies.append(permEnt)

    #Normalize the permutation entropy and return results
    permutationEntropies = np.array(permutationEntropies)
    return permutationEntropies
```

[14]:
```
embeddingDimension, window, windowStepSize, tao = 4, 2048, 2048, 1

fig, axs = plt.subplots(len(channelNames))
fig.set_figheight(2 * len(channelNames))
fig.set_figwidth(27)

permutationEnts = []
for i, name in enumerate(channelNames):
    ents = getPermuationEntropies(channels[name], embeddingDimension, window,␣
 ↪tao)
    permutationEnts.append(ents)
```

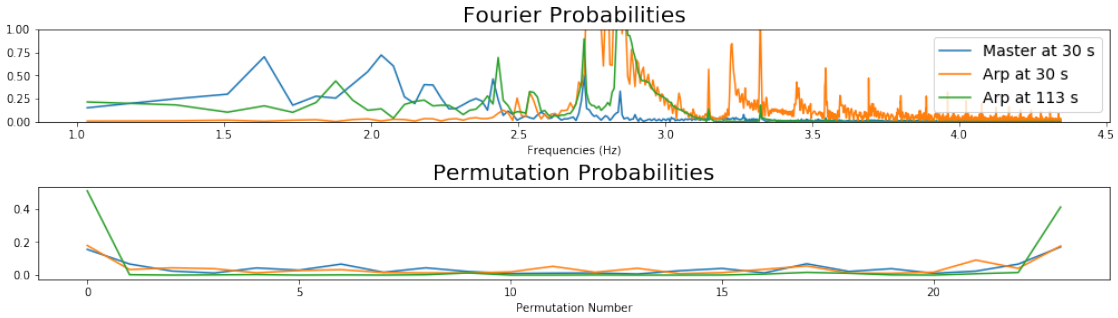
```
axs[i].plot(ents)
axs[i].set_title(name)
axs[i].set_xticks([])
```



Provide plots comparing the fourier and permutation probabilities.

[15]:
```
time = 30
t2 = 113
freq, fourierMaster = getFourierTransformAtIndex(channels['Master'], int(time *␣
 ↪samplingFrequency))
freqArp, fourierArp = getFourierTransformAtIndex(channels['Arp'], int(time *␣
 ↪samplingFrequency))
freqArp2, fourierArp2 = getFourierTransformAtIndex(channels['Arp'], int(t2 *␣
 ↪samplingFrequency))


embeddingDimension, window, windowStepSize, tao = 4, 2048, 2048, 1
idx = int(time * samplingFrequency)

mast = channels['Master']
mastSnipplet = mast[idx : idx + windowSize]
mastPermEnt, mastPermProbs = permutationEntropy(mastSnipplet,␣
 ↪embeddingDimension)

arp = channels['Arp']
arpSnipplet = arp[idx : idx + windowSize]
arpPermEnt, arpPermProbs = permutationEntropy(arpSnipplet, embeddingDimension)

idx = int(t2 * samplingFrequency)
arpSnipplet2 = arp[idx : idx + windowSize]
arpPermEnt2, arpPermProbs2 = permutationEntropy(arpSnipplet2,␣
 ↪embeddingDimension)

fig, axs = plt.subplots(2)
fig.set_figheight(4)
fig.set_figwidth(14)

axs[0].plot(freq, fourierMaster)
axs[0].plot(freqArp, fourierArp)
axs[0].plot(freqArp2, fourierArp2)
axs[1].plot(mastPermProbs)
axs[1].plot(arpPermProbs)
axs[1].plot(arpPermProbs2)

axs[0].set_title('Fourier Probabilities', fontsize=20)
axs[1].set_title('Permutation Probabilities', fontsize=20)

axs[0].set_xlabel('Frequencies (Hz)')
axs[1].set_xlabel('Permutation Number')
```

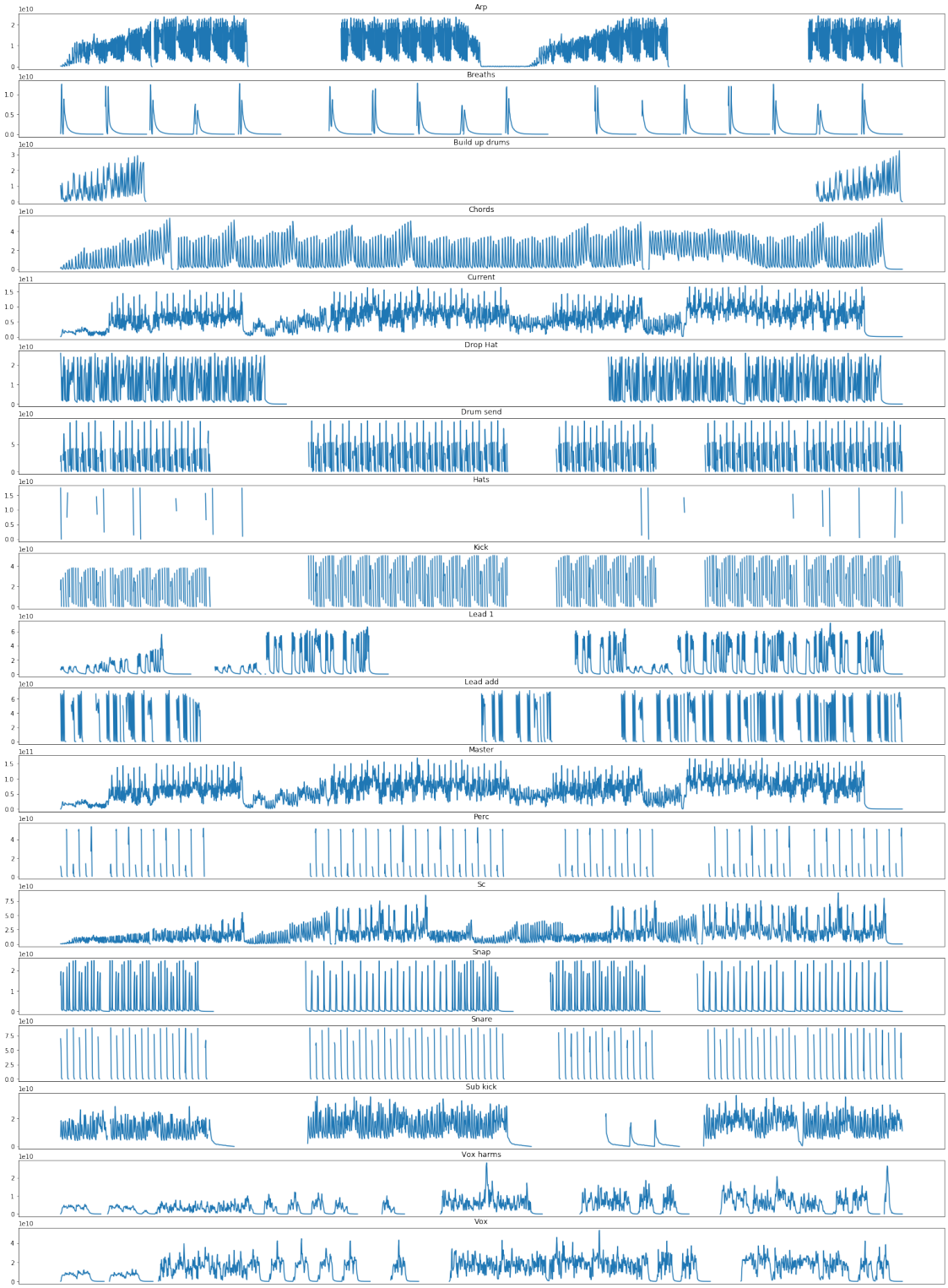
```
axs[0].set_ylim((0, 1))
axs[0].legend(['Master at 30 s', 'Arp at 30 s', 'Arp at 113 s'], prop={'size':␣
↪14}, loc=1)

fig.tight_layout(pad=0.6)
```



Code for generating code based on discussion section of report.

[16]:
```
def getModifiedFourier(series):

    #Calculate the fourier transform for a snipplet of the song
    fourierTrans = fft(series)[0:len(series)//2]
    fourierTrans = np.abs(fourierTrans)

    #Get the frequencies corresponding to the fourier transform that will be␣
↪used for the x-axis
    frequency = np.linspace(0, samplingFrequency/2, len(fourierTrans))
    frequencyIsInfinity = ~ np.isinf(frequency)
    frequency = frequency[frequencyIsInfinity]

    fourierTrans = fourierTrans[frequencyIsInfinity]
    return frequency, fourierTrans

def getModifiedFourierEntropy(channel):

    #We iterate over the whole channel waveform and store results from sliding␣
↪window as lists
    time, entropies = [], []
    for i in range(len(channel) // windowSize):
        t = i * windowSize / samplingFrequency
        time.append(t)

        #Get the fourier transform for this sliding window
        idx = i * windowSize
```

```
        snipplet = channel[idx:idx+windowSize]
        freq, transform = getModifiedFourier(snipplet)

        #We weigh the transform based on the trapezoidal rule and calculate the␣
↪entropy for this window
        weightedTransform =  (freq[1:] - freq[:-1]) * (transform[1:] +␣
↪transform[:-1]) / 2
        localEntropy = weightedTransform * np.log2(weightedTransform)
        entropy = np.sum(localEntropy)
        entropies.append(entropy)
    return time, entropies

fig, axs = plt.subplots(len(channelNames))
fig.set_figheight(2 * len(channelNames))
fig.set_figwidth(27)
for i, name in enumerate(channelNames):
    t, ents = getModifiedFourierEntropy(channels[name])
    axs[i].plot(t, ents)
    axs[i].set_title(name)
    axs[i].set_xticks([])
```


Arp
Breaths
Build up drums
Chords
Current
Drop Hat
Drum send
Hats
Kick
Lead 1
Lead add
Master
Perc
Sc
Snap
Snare
Sub kick
Vox harms
Vox

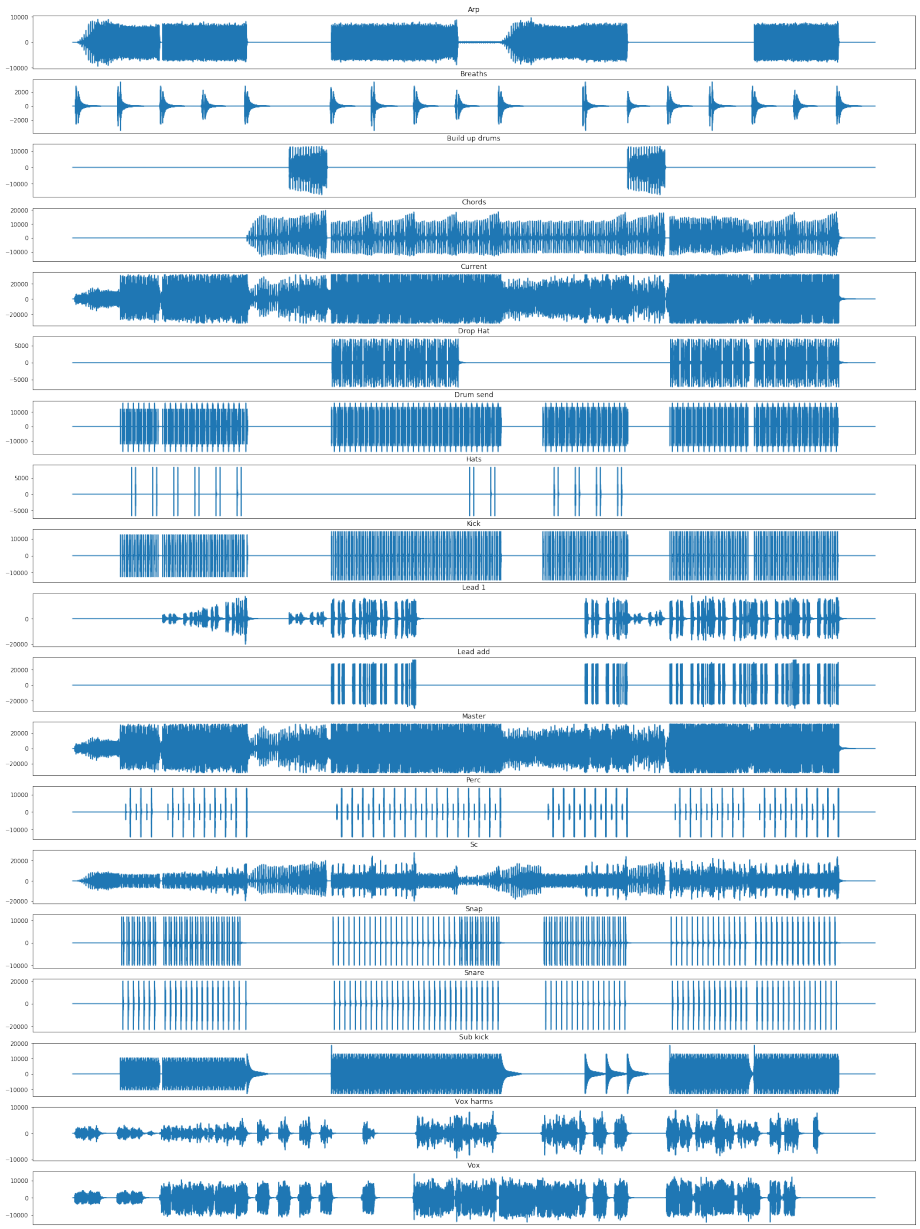


Figure 6: The raw waveform data after exporting from FL Studio.

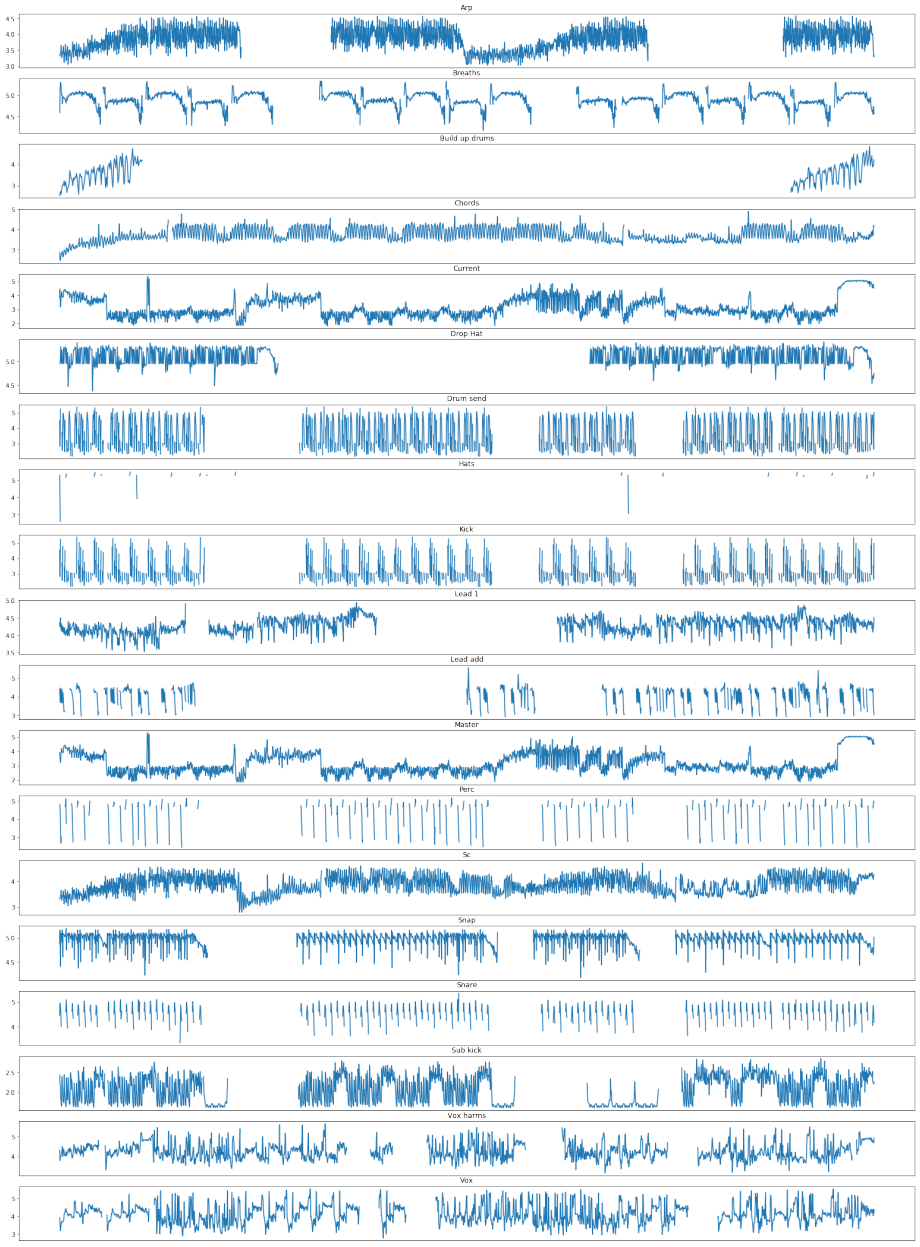


Figure 7: The entropy calculated from all channel waveforms.

Figure 8: The active information calculated from all channel waveforms.

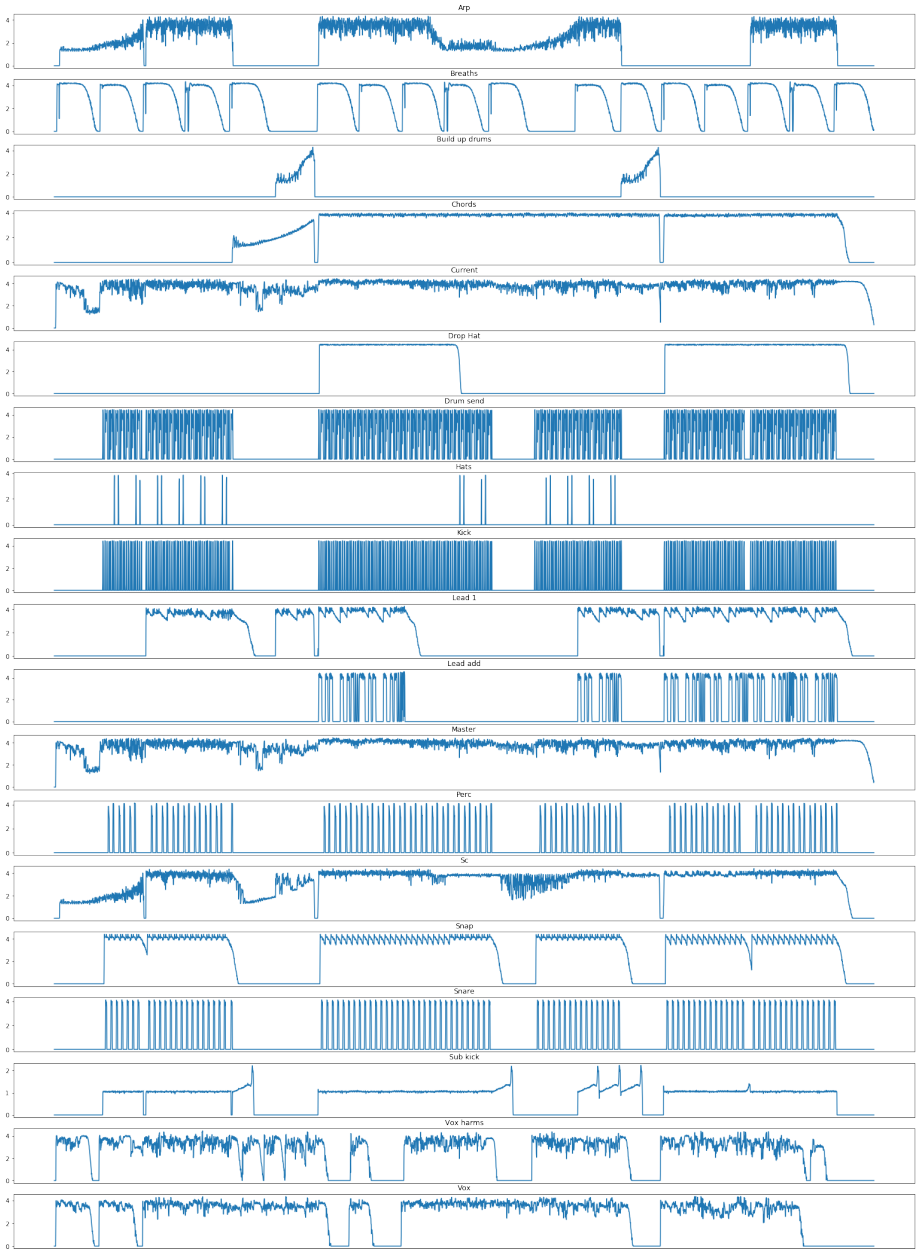


Figure 9: The permutation entropies calculated from all channel waveforms.

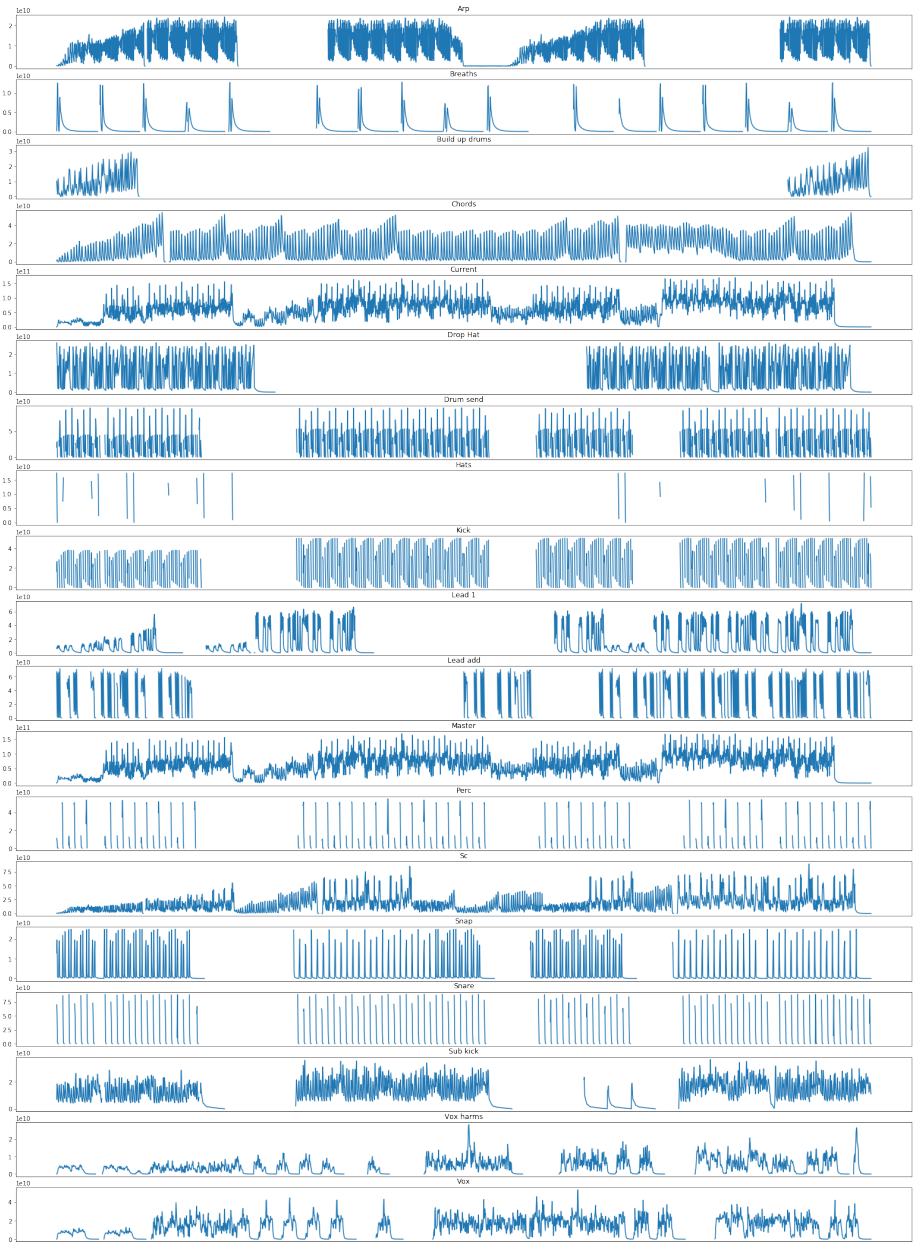


Figure 10: The modified Fourier Entropy based on the discussion section of the report