Soundscape Analysis on Low-power Acoustic Sensors

St. Catherine's College
University of Oxford

Author:
Quei-An Chen

Supervisor:
Professor Alex Rogers

A thesis submitted for the degree of
Master of Science in Computer Science

Trinity Term 2017

Acknowledgement

Foremost, I would like to express my sincere gratitude to my supervisor, Professor Alex Rogers, for the insight and guidance he has given throughout this project, without whose vision this project would never have been possible.

I'm grateful for his help of teaching me how to do proper research, and how to write a well structured dissertation.

My classmates, especially Sterling Poon, Mikayel Samvelyan, Haoche Zhang and Cecile Berillon have been very helpful as well, we discussed our projects, and the discussion and the time we spent together helped me relax a lot.

Last but not the least, I would like to thank my parents, Jiin-Donq Chen and Shiu-Mann Lu, who have been supporting me during my study abroad in the U.K.

Abstract

The importance of extracting important features from acoustic recordings is increasing nowadays, since people care more about environmental issues, and acoustic recordings are able to give a rough idea about the landscape. However, it is difficult for scientists to examine the whole recordings, since they are generally very long, and computationally very expensive to load.

Previous research has shown that some values, called acoustic indices, and be used to reveal important features of the recordings. Recently, these values are still computed using the entire recordings. In this paper, we will establish algorithms that are able to compute them fast and precisely without storing every data, which will in turn save memory and energy, and serve as an instant insight into the recordings.

Contents

1 Introduction

1.1 Motivation

Acoustic recordings of the environment allow us to monitor wild species such as birds, insects and frogs that produce sounds. These species play an important role as indicators of biodiversity and environmental health [1]. In addition to keeping track of species, recorded audio data can also contribute to the study of the temporal and spatial distribution of sound throughout a landscape, known as soundscape ecology, which reflects human impacts on the environment [2][3].

1.2 Previous research

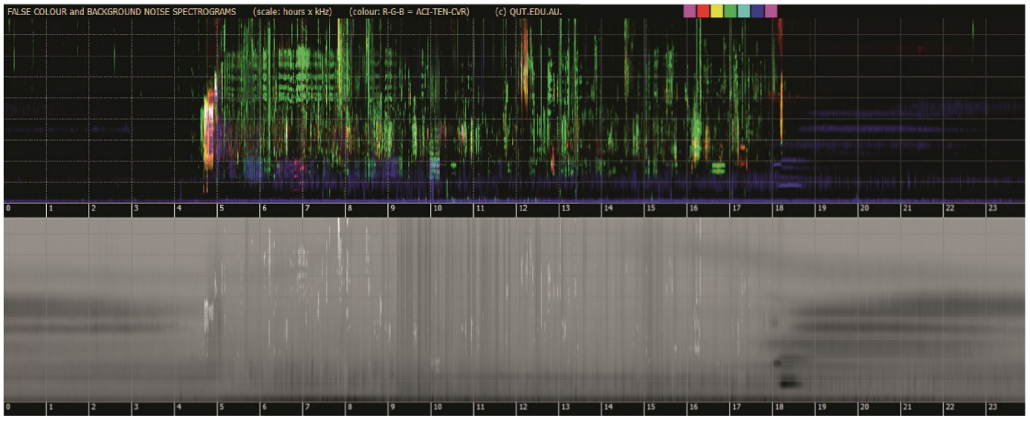
While it is easy to collect much data, the processing presents a great difficulty since the recordings are too long for scientists to listen entirely. Moreover, in order to deal with such huge amount of data (for example, a fourteen day deployment would generate 120GB of sound recording), even with the aid of audio softwares, it is computationally very expensive. Hence, although we have sufficient data, it is hard to extract meaningful information.

To resolve the problem, Wimmer, Towsey, et al. (2013) [4] presented a way to compress the information contained in recordings by using *acoustic indices*. An acoustic index is a scalar or a vector that summarises important characteristics of a recording, such as acoustic power or frequency distribution.

In Wimmer, Towsey, et al. (2015) [5], they discussed three indices that could be used to represent the entire recording. By compressing the recordings to acoustic indices, they were able to compress each one minute data into only some floating point numbers. We'll establish a more detailed description of these indices and calculations in section 3.

Furthermore, they used the indices to produce false color images of the recordings, which serve as a straightforward insight into what actually is contained in

the audio. In this way, without actually listening to the recordings, scientists can directly know what kind of sounds are present at which moments.



A sample false color image in [5]

1.3 Project aims

In previous works, the indices are calculated given the entire data, which means that they recorded and stored every piece of data before the final indices can be calculated on the computer. In terms of data processing, the indices allow us to explore the huge amount of data which is otherwise impenetrable. However, the power consumption and the amount of data recorded remain the same. In reality, regarding power usage and battery life, it is expensive (commercial recorders cost about £1000, but devices with lower cost at about £40 exist) to save up all the data into the memory (SD cards). The device that we deploy to record the sound is called **Audiomoth** (`http://www.openacousticdevices.info/`). It has the advantage that it is low-cost, which allows us to deploy lots of them to collect spatial data. Moreover, it is re-programmable so that we can do any computations that we want on the device.

Since the device is able to do computations while listening, we came up with the idea to compute the acoustic indices at the same time of listening. In other words, in this project, we aim to develop algorithms that enable us to compute the indices

at the same time of listening and recording, to avoid saving unnecessary data. We call these algorithms **online** algorithms, which means that they process the audio data piece by piece in a serial fashion, keeping only a small number of variables, in contrast to the original **offline** algorithms, which requires the entire data at once to do the computations.

First, we exploited the offline algorithms provided in [5], and established online algorithms that give the exact result as their offline counterparts. For the indices that could not be computed exactly in an online fashion, we adapted their formulae to approximate the original algorithms.

After establishing the exact algorithms, we needed to guarantee that the online algorithms are suitable for the device in terms of speed and memory consumption, given the constraints of the device (a maximum clock speed of 48MHz and an internal 32kB SRAM, with additional external 256kB SRAM). We measured the actual computational time on the device, to make sure that these algorithms actually work.

Finally, since some indices are adapted so that they can be computed in an online way, we established formal mathematical proofs to give theoretical close forms of the errors committed by our modified algorithms.

From an experimental perspective, to compare the actual errors of the online algorithms to the offline ones, we deployed the device that computes the acoustic indices along with that only does the recording in the surrounding of St. Catherine's College to collect audio data. The recordings were then used to compute the indices offline using the algorithms provided in [5], and the results were compared with the indices generated online. Lastly, we also examined their ability to restore the information of the original recordings, and reproduced the false color images as described in [5].

1.4 Dissertation structure

The dissertation is organised as follows :

In section 2, we discuss the background of the research, and the tools (hardwares and softwares) that we use in the project.

In section 3, we describe the procedure of signal processing, establish the online algorithms that compute the acoustic indices, and discuss their speed and memory issue respectively.

In section 4, we formulate the theoretical error and show experimental results of the online algorithms, along with false color images that we reproduce as in [5].

In section 5, we conclude the work that we have done in the project, and discuss possible future work.

In section 6, we list some personal thoughts that we had during and after the project.

2 Background

In this section, we list the indices that have been researched before, and discuss their usefulness, along with the tools that we use in the project.

2.1 Acoustic Indices

Previous research has shown that acoustic indices can be informative. In general, an acoustic index is a number or a vector that summarises important features of an audio segment.

In [5], the authors described three acoustic indices, acoustic complexity (ACI), temporal entropy (H[t]) and acoustic cover (CVR), all of which are vectors of size 256, that show interesting features of the recordings. We'll now describe how they are computed and what they show respectively. To begin with, in fact, these indices are all derived from the FFT (Fast Fourier Transform) of the recording, so we start by introducing the FFT :

FFT index: An FFT converts a signal from its temporal domain to a representation in the frequency domain. The inputs are amplitudes of the recording, and the outputs are spectral powers in each frequency bin. The explicit computation is as follows :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad \text{for } k = 0, ..., N-1$$

where N is the number of samples (usually a power of 2) and k is the index of the frequency bin.

N was set to 512 in [5], and as the data are entirely real numbers, all X_k for $k > 256$ are dropped as they are just conjugate numbers of X_{512-k}. Lastly, X_{256}, which corresponds to the exact Nyquist frequency (half of the sample rate), is also dropped to make the output size exactly $N/2 = 256$ for the sake of tidiness.

The FFT index gives an intuitive and obvious profile of the recording, showing what frequencies are active at each moment.

The three particular indices ACI, H[t] and CVR are chosen because they convey different features of the recording, which is an important reason that the authors used them to construct false color images. All of them are derived from the FFT of the recording, in particular, **they are all vectors of size $\mathbf{N/2 = 256}$, where one value represents one frequency bin.** We'll now describe their features one by one :

ACI index: For each frequency bin f in a spectrogram of length L, calculate the average absolute fractional change in spectral amplitude from one index to the next.

$$ACI_f = \sum_{i=1}^{L} |a_i - a_{i-1}| / \sum_{i=0}^{L} a_i$$

where i indexes over all amplitude values in frequency bin f of the spectrogram. $\sum_{i=0}^{L} a_i$ is a normalizing factor.

The ACI index measures the degree of variation of amplitudes during a period of length L, which, in particular, reveals if there are acoustic events happening. Basically, the value is large when some acoustic source breaks the environment from silence.

H[t] index: The entropy of each frequency bin f in a spectrogram of length L. The amplitude values are squared and normalised in order to be treated as a probability distribution. The entropy of the signal is calculated as

$$H[t]_f = -\sum_{i=0}^{L-1} pm_i \log_2 pm_i / \log_2 L$$

where $pm_i = a_i / \sum_{i=0}^{L} a_i$ is the probability mass of amplitude a_i among all amplitudes in frequency bin f.

The H[t] index measures the concentration of the amplitudes. The difference between ACI is that this index is more sensitive to continuous sound producing sources. For example, if a species is singing for a long period, the entropy index will keep being high during that period, whereas ACI will only show a distinguishable peak at the beginning of the signal.

CVR index: The fraction of cells in frequency bin f of the noise-reduced spectrogram of length L where the spectral power exceeds 2 dB. The noise of frequency bin f is calculated as

$$Noise_f = mean(dB_i) + noiseFactor \times std(dB_i)$$

where $mean(dB_i)$ and $std(d_i)$ are the mean and the standard deviation of the distribution of $dB_i = 20\log_{10}(a_i)$ in the spectrogram. In [5], it is said that setting $noiseFactor$ to be greater than 0.1 reduces too much signal, therefore it is set to be exactly 0.1.

Finally,

$$CVR_f = \#\{i \in [0..L] \mid dB_i - Noise_f > 2\}/L$$

The CVR index measures the difference of the signal's distribution with respect to a Gaussian distribution. The noise is modeled as the mean of the signal plus 0.1 times its standard deviation, then compute how much fraction of the amplitudes in the minute is above the threshold. If the signal is a perfect Gaussian distribution, then this value is a constant.

In [5], the indices are computed every minute, which means $L = 5625$. In section 3.1, we will describe how L is determined in more detail.

2.2 Other Useful Functions

As in the calculation of H[t] we need to use logarithmic functions, we list here some existing codes that allow us to accelerate the computation of logarithmic functions. They have the advantage of being fast while losing some precision. We list

two alternatives that computes logarithmic function faster than the built-in $log2f$ below. Further discussion of how they will be used in our paper will be described in section 3.

2.2.1 Fastlog2

A fast $\log_2$ approximation can be obtained by exploiting the format of the float number. By decomposing the mantissa and the exponent (in base 2), $\log_2$ can be obtained by only doing elementary arithmetic operations. More precisely, one can write

$$x = (1 + m_x) \times 2^{e_x}$$
$$\log_2 x = e_x + \log_2(1 + m_x)$$

And interpret the byte representation of x as an integer :

$$I_x = (e_x + B)L + m_x L$$
$$I_x/L - B = e_x + m_x$$
$$\log_2 x = I_x/L - B + \log_2(1 + m_x) - m_x$$

where $B = 2$ and $L = -23$, used in the 32 bit floating-point number representation.

Finally, a rational approximation is used to replace $\log_2(1 + m_x) - m_x$ for $m_x \in [0, 1($.

A sample code is as follows :

```
float fastlog2(float x) {
    union { float f; unsigned int i; } vx = { x };
    union { unsigned int i; float f; } mx = { (vx.i & 0x007FFFFF) | 0x3f000000 };
    float y = vx.i;
    y *= 1.1920928955078125e-7f;

    return y - 124.22551499f - 1.498030302f * mx.f
            - 1.72587999f / (0.3520887068f + mx.f);
}
```

2.2.2 Fasterlog2

In the above code, an rational approximation is used to replace $\log_2(1+m_x)-m_x$. If one wants to be rougher but faster, this value can be replaced by its mean over the segment [0,1], which equals to $\frac{-2+\log(8)}{\log(4)}$. The resulting sample code is as follows :

```
float fasterlog2 (float x){
  union { float f; unsigned int i; } vx = { x };
  float y = vx.i;
  y *= 1.1920928955078125e-7f;
  return y - 126.94269504f;
}
```

2.3 Materials

In this section, we list the hardwares and softwares that are used for testing and for the actual deployment, along with datasets that serve for testing.

2.3.1 Hardwares

Audiomoth AudioMoth is a new device from a university research project carried out by Professor Alex Rogers, University of Oxford and PhD students Andrew Hill, Peter Prince, University of Southampton.

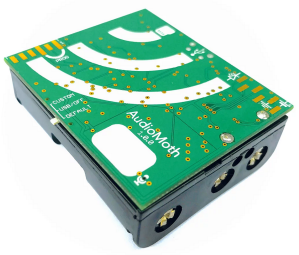


Figure 1: Audiomoth

Its main advantages are that it is low-cost (of less than £40) and easy to re-program using the USB bootloader that is pre-installed. It uses a analog MEMS microphone to record sounds in 16bit integers continuously, and supports different

sample rates up to 192kHz.

Its small size (only $58 \times 48 \times 15$ mm) and low energy consumption (powered by 3 AA batteries) along with its low cost allows us to deploy many of them in different places to cover the whole landscape at the same time.

Computationally, it uses the same processor as EFM32 Gecko boards produced by Silicon Labs, which has a maximum clock speed of 48MHz. This processor, ARM Cortex-M4, has the advantage that it supports optional FPU (Floating Point Unit), which allows us to do computations in 32bit floating points very fast. This feature is essential since most of the indices that we aim to compute are floating point numbers.

Regarding it memory size, it has an internal 32kB SRAM and additional external 256kB SRAM, which allows us to store the data in the SRAM first, then save to the SD card when the SRAM is full to lower the energy consumed by data writing.

Desktop computer For the sake of testing, in order to verify our online algorithms' ability to reproduce the results that offline algorithms yield, we programmed in Python and C on a desktop computer to compare.

The following table sums up the important specifications of the devices :

Name	Memory	Clock speed
Audiomoth	32kB SRAM + 256kB SRAM	48MHz
Desktop computer	8GB RAM	2.70GHz

Table 1: Spec comparison

2.3.2 Softwares

The software used to program EFM32 Wonder Gecko board and Audiomoth is Simplicity Studio (version 4), with development kit Gecko SDK v5.0.0. We use the optimised library arm_cortexM4lf_math that is also included in the SDK to do all

math computations that are required in the algorithms. Finally, the floating point ABI is set to "hard", which means that we enable the FPU to do floating operations on the hardware, which is fast.

For desktop machine testing, we use two languages, Python (version 3.6.1) and C (gcc version 5.3.0). Every graphic result in this paper is generated in Python.

2.3.3 Datasets

We recorded sounds in New Forest, Southampton and in the surrounding of St. Catherine's College, Oxford to evaluate the algorithms that we establish in the paper.

3 Computationally Efficient Online Algorithms for Computing Acoustic Indices

Having acknowledged the previous works, especially how acoustic indices are obtained, we will describe the works that we have done during the project, including the sound processing that is done before the acoustic indices can be computed, and the online algorithms that we run on the device.

3.1 Signal Processing

Before computing the acoustic indices, either in an online or offline fashion, we need to do some processing on the raw audio data to make sure the following computations yield informative results.

The sound is recorded continuously at a sample rate that is set by the user (usually 48kHz) using signed 16bit integers. As we mentioned in section 2, all indices are derived from FFTs. In [5], the FFTs are performed every 512 samples, and the indices are computed every minute; in our paper, we continue to use this setting.

We discuss first how to compute the indices for one minute.

To begin with, recall in section 2, we need to decide the length L of the spectrogram for an audio of one minute. In general, $L = \lfloor \frac{sample\_rate \times 60}{512} \rfloor$, which also equals to the number of FFTs performed in one minute. In case the sample rate is 48kHz, $\boxed{L = 48000 * 60/512 = 5625}$, as what we mentioned previously. If the argument inside the floor function turns out to not be an integer (for example, if the sample rate is not 48kHz), we discard the final fraction of the minute which has fewer than 512 samples.

Then, for each 512 samples in the spectrogram of length L, we do the following pre-processing to clean the signal before computing the FFT :

1. Remove the DC offset of the signal.

2. Apply a Hamming window of size 512 to smoothen the signal.

Next, the corresponding FFT of size 512 is calculated. In order to establish online algorithms, instead of the data itself, we keep a certain number of variables and update them after each FFT (sections 3.2 - 3.5 will describe what variables stand for in more detail).

Once one minute passed and the variables are updated completely, we do a final computation on the variables to get the indices, and reset the variables to be used for the next minute. The indices are first stored in a buffer of size 32kB, which is the maximum data size that the device can write at a single time. Once it's full, the indices will finally be written to the SD card, to reduce the amount of energy that is consumed while writing.

While the indices are being written to the SD card, the microphone keeps listening, but the audio data is dropped until the writing is finished. The whole cycle described above continues during the period that is set by the user.

Following is a graph summarising all the processing for one minute :

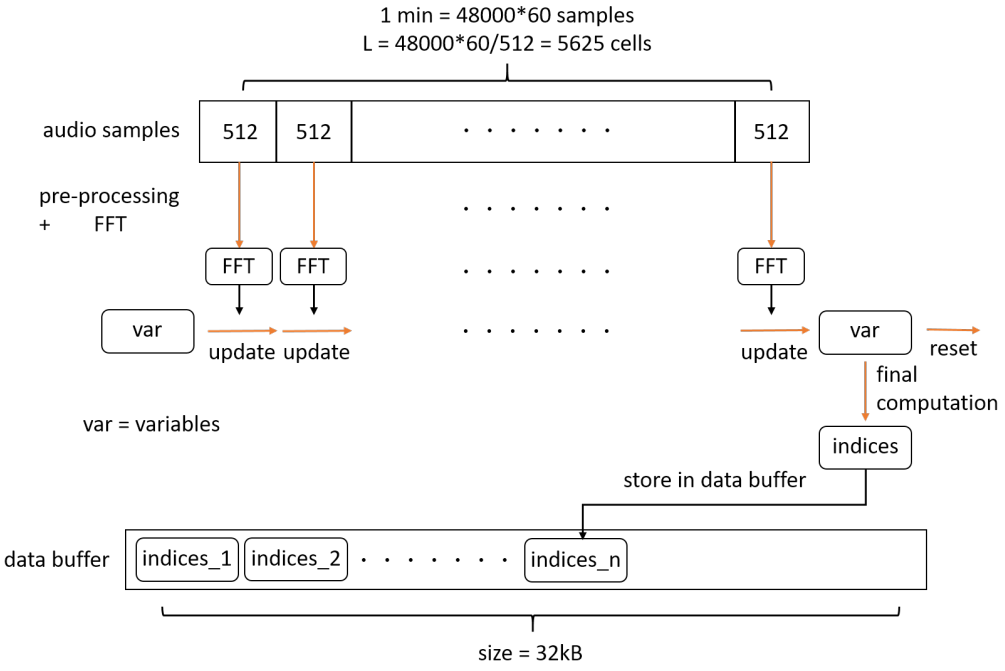


Figure 2: Signal processing for one minute

Orange arrows indicate that there are computations performed. For the right-most arrow, after the variables are reset (the values are set back to zero), they are again used to compute the indices for the next minute. Graphically, this means that arrow is actually linked to the left part of the figure, and the whole process continues during the period set by the user (usually 24 hours or more).

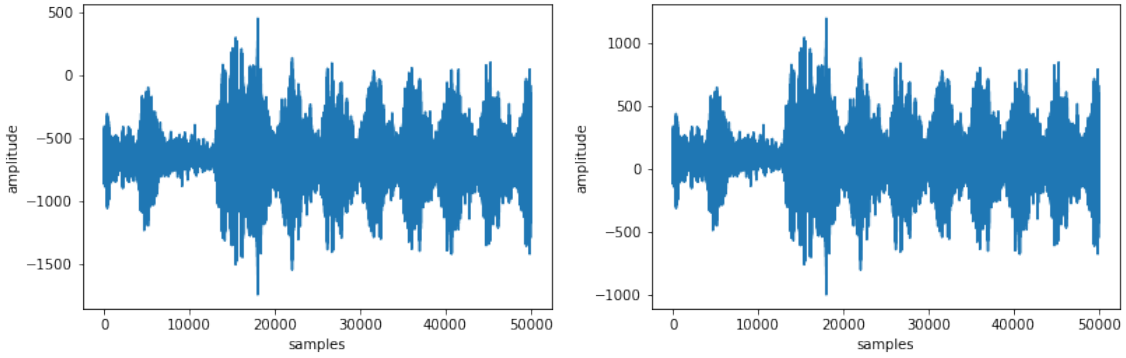
There is a slight modification of the signal processing for the very first minute of the recording. We split this minute into two halves of 30 seconds, use the first half to compute some variables that are needed for the algorithms, and the indices for the minute are calculated using only the data from the second half. This modification is required in the algorithm that computes CVR, and we will discuss more in detail in section 3.6.

We will now start by describing the pre-processing steps and how FFTs are computed, then establish the online algorithms that computes the indices.

3.1.1 DC Offset Removal

A DC offset removal is continuously performed to make sure that the signal has zero offset, to avoid computation errors in the resulting FFTs. To make it compatible with the computation of the indices, we also need an online algorithm to do the job. We will now describe what DC offset is, and how we remove it in an online way.

The effect of DC offset A DC offset is an offsetting of a signal from zero. A non-zero DC offset is usually caused by a fixed voltage offset in the audio chain before the signal is converted to digital values. Below are pictures showing signals with and without DC offset :



(a) With DC offset (b) Without DC offset

Figure 3: The effect of DC Offset

The above signals have exactly the same waveform, while the left one has a DC offset of about -500.

It is important to make sure that the DC offset is removed, since any non-zero DC offset will result in an error in the calculation of the FFT. Recall how FFT is calculated :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad \text{for } k = 0, ..., N-1$$

Therefore, if the signal (x_n) has an offset d, then

$$\begin{aligned} X'_k &= \sum_{n=0}^{N-1} (x_n + d) e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} + d \sum_{n=0}^{N-1} e^{-i2\pi kn/N} \\ &= X_k + d \sum_{n=0}^{N-1} e^{-i2\pi kn/N} \end{aligned}$$

If we look at the sum $\sum_{n=0}^{N-1} e^{-i2\pi kn/N}$:

$$\sum_{n=0}^{N-1} e^{-i2\pi kn/N} = \begin{cases} N & \text{if } e^{-i2\pi k/N} = 1 \\ \dfrac{1 - e^{-i2\pi k}}{1 - e^{-i2\pi k/N}} & \text{otherwise.} \end{cases}$$

The condition $e^{-i2\pi k/N} = 1$ is satisfied if and only if k/N is an integer. Since $k = 0, ..., N-1$, the only case where it is fulfilled is when $k = 0$. If $k \neq 0$ (the second case), the sum is equal to 0 since $e^{-i2\pi k} = 1$ for every integer k.

Therefore, we can rewrite the result as :

$$\sum_{n=0}^{N-1} e^{-i2\pi kn/N} = \begin{cases} N & \text{if } k = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Hence, the offset of the FFT becomes :

$$X'_k = \begin{cases} X_k + Nd & \text{if } k = 0 \\ X_k & \text{otherwise.} \end{cases}$$

As a result, the offset of frequency bin 0 is amplified to a great degree, which we can avoid by removing the DC offset in the signal before the calculation of the indices.

DC offset removal algorithm The offline algorithm that removes the DC offset of a signal is rather simple, we only need to compute it's mean value to be the offset, and subtract that value from the original signal. To do this in an online way, we can consider the DC offset as a constant signal of frequency 0Hz, therefore, applying a high-pass filter will attenuate its power. A standard way to construct a high-pass filter is by expressing the output as the sum of the decaying contribution of the previous output and the contribution of the change in input.

Mathematically, we can write :

$$output_i = \alpha \cdot output_{i-1} + (input_i - input_{i-1})$$

where

- $\alpha = \dfrac{1}{2\pi f_c \Delta_T + 1}$
- f_c is the cutoff frequency
- Δ_T is the time span between two consecutive inputs

If we use 48kHz sample rate, $\Delta_T = 1/48000$, and we want f_c to be as small as possible to avoid attenuating spectral powers of frequencies other than 0Hz. As a result, choosing a value of α to be close to 1 is desirable. In our implementation, we choose $\alpha = 0.995$, which corresponds to $f_c = 38.2$ Hz.

Following is the algorithm implemented in C :

```
/* DC filter */

int32_t filteredOutput;
int32_t scaledPreviousFilterOutput;

for (int i = 0; i < size; i++) {

    int16_t sample = multiplier * data[i];

    scaledPreviousFilterOutput = (int32_t) (0.995f * previousFilterOutput);

    filteredOutput = sample - previousSample + scaledPreviousFilterOutput;

    data[i] = (int16_t) filteredOutput;

    previousFilterOutput = filteredOutput;

    previousSample = (int32_t) sample;

}
```

The variable *multiplier* is a factor to normalise the volume across different oversampling rates, which is usually equal to 1. Note that the audio samples are recorded using 16bit integers, but since we need to multiply the values by 0.995, which is a floating point number, we use the FPU to do computations in 32bit floating points. For the same reason, the filtered output is first computed as a 32bit integer, then converted back into a 16bit integer.

Following is the figure showing the above signal with DC offset, and its output when applied the DC offset removal algorithm :

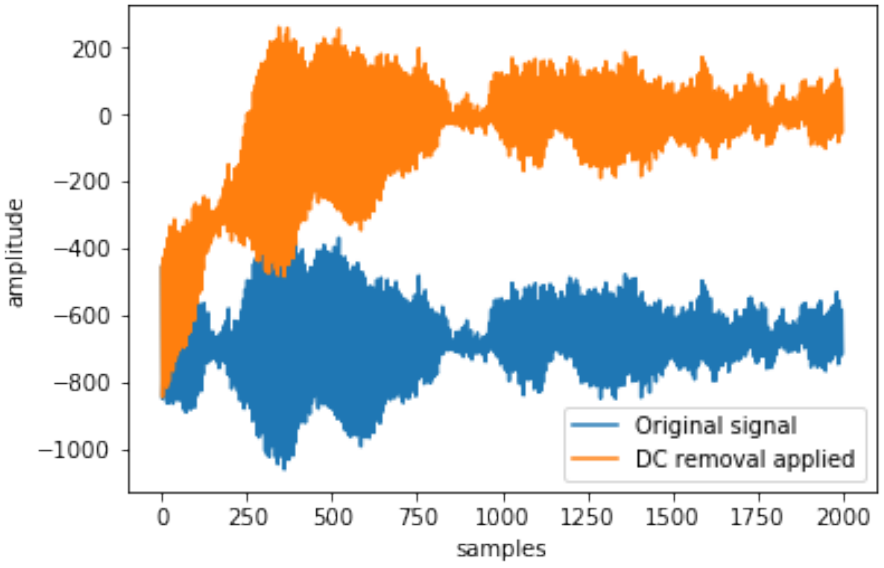


Figure 4: DC removal algorithm

As we can see, after about 500 samples, the DC offset is completely removed. We can then proceed to the next step of the pre-processing, then perform FFT and compute the indices.

3.1.2 Hamming Window

After the DC offset is removed, we need to regularise the signal to get correct FFT results. Originally, as the name suggests, FFT is a variation of Fourier Transform. Therefore, its application is mainly restricted to periodic functions, since some information about the signal may be lost when applying to non-periodic functions. To minimise this effect, we apply a Hamming window to make the signal approximately periodic. We will now describe how we set and use the Hamming window in this section.

A Hamming window is an array of size equal to the size of the FFT (512 in our case). The explicit formula is as follows :

$$Hamming(n) = 0.54 - 0.46\cos(\frac{2\pi n}{511}) \quad \text{for } n = 0, ..., 511$$

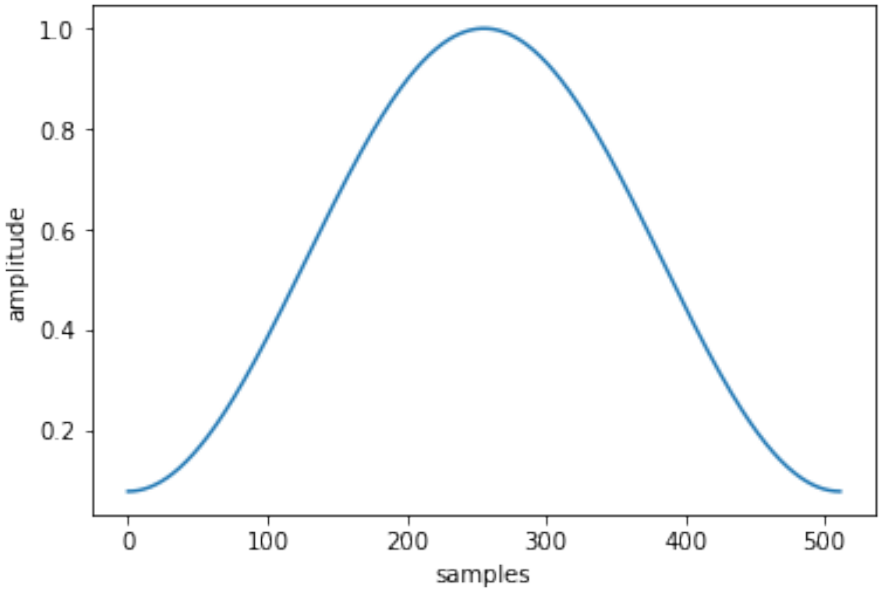


Figure 5: Hamming window of size 512

If we don't apply the Hamming window, then the end of each 512 samples will not match the start of the next piece of 512 samples, which makes the signal discontinuous. In terms of FFT, lots of high-frequency peaks will show up as errors of the discontinuity. Therefore, we apply Hamming window to make sure the starts and ends matches. Following is a signal's waveform before and after the application of the above Hamming window :

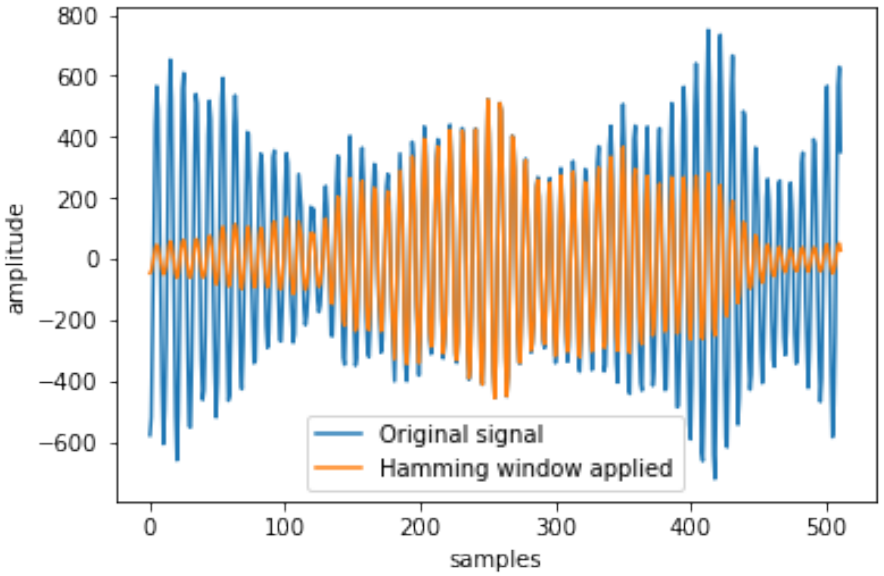


Figure 6: Signal before/after applying the Hamming window

Finally, in terms of the implementation, as the values of Hamming window array are fixed, we compute them beforehand, and hard-code the array to save the computational time.

3.1.3 FFT

After the above pre-processing is done, the signal is now clean and neat to perform any operations. As all indices are derived from the FFTs of the signal, we need to first compute the FFTs.

The FFT is computed as a real number FFT using arm math library `https://www.github.com/ARM-software/CMSIS/blob/master/CMSIS/DSP_Lib/Source/TransformFunctions/arm_rfft_f32.c` to gain more speed.

3.2 Online Algorithm Specifications

Given the correct FFTs of every 512 audio samples, we can now proceed to the computation of the indices themselves. In contrast to a desktop machine, since Audiomoth only has a limited amount of computational power and memory space available, we need to design our algorithms such that the device can handle all the processing. Basically, **speed** and **memory space** are two main issues that we need to tackle. Recall that the processor runs at a maximum speed of 48MHz, and the total memory is at most 288kB, with these computational and memory constraints, our online algorithms needs to be fast, and must not occupy too much memory. The below paragraphs will discuss our respective solutions.

As the FFTs come as a serial data (one for every 512 samples), we can keep track of only a small number of variables, and update them every 512 samples according to the corresponding FFT, to save the memory.

In terms of speed, since the audio is recorded at a rather high sample rate (usually 48kHz), this means that any update of the values should be computed within $512/48000 = 10.66$ milliseconds, before the next piece of audio data becomes ready for treatment. Knowing that some of the operations take more time than the others, we replace slow ones with faster alternatives. For example, on Audiomoth, a division takes 14 clock cycles, whereas a multiplication only takes 1. Thus, if somewhere in the algorithm we need to divide by a same value x many times, then it is better to compute $1/x$ first, then use multiplications instead. In the same way, we aim to optimise every computation in the algorithm.

A detailed table that specifies the computation cycles for different operations can be found in the manual of Wonder Gecko :

Operation	Cycle count
Absolute value	1
Addition	1
Subtraction	1
Multiplication	1
Division	14
Load N floats	1+N
Store N floats	1+N
Square root	14

Table 2: Floating point instructions

Before going to algorithm details, we list here the explanation of the terminology that will be used in the algorithms.

3.3 Terminology

Recall that the indices are in fact vectors of size 256, which corresponds to the number of frequency bins derived from the FFT computation. In all the following algorithms, we will only discuss the procedure that is applied to one frequency bin, since the same procedure is applied to all 256 frequency bins. As a result, for the sake of simplicity, we omit the discussion frequency bin-wise, and the term "variable" will always refer to a floating-point array of size 256.

We'll now describe how we design online algorithms that compute or approximate the indices ACI, H[t], and CVR, while respecting the above specifications.

3.4 Acoustic Complexity Index (ACI)

The original ACI is a ratio between two sums, $ACI_f = \sum_{i=1}^{L} |a_i - a_{i-1}| / \sum_{i=1}^{L} a_i$, where a_i are amplitudes for the frequency bin f. To obtain a sum, we only need one variable storing the partial sum, then successively add new values to it. Since we

also calculate the difference between consecutive amplitudes, we need another variable to store the the previous amplitude in order to compute the absolute difference with the current amplitude. In total, three variables are required.

The full algorithm works as follows :

Algorithm 1 ACI

Input: A series of amplitudes a_i, one at a time.

Output: $ACI_f = \sum_{i=1}^{L} |a_i - a_{i-1}| / \sum_{i=1}^{L} a_i$

1: $aci\_previous \leftarrow 0, sumDiff \leftarrow 0, sum \leftarrow 0$
▷ The numbers that we need to store
2: **for** i from 1 to L **do**
3: Acquire a_i
4: $sumDiff \leftarrow sumDiff + |a_i - aci\_previous|$
5: $sum \leftarrow sum + a_i$
6: $aci\_previous \leftarrow a_i$
7: **return** $sumDiff/sum$

In terms of the complexity, each update requires two additions, one subtraction and one absolute function per frequency bin.

Theoretically, the online algorithm yields the same result as the offline one.

3.5 Temporal Entropy (H[t])

The temporal entropy of the signal is defined by considering the amplitudes as a probabilistic distribution, and compute the entropy of this distribution. It appears to be impossible to obtain the entropy value in an online way, since the corresponding probabilistic distribution requires the whole data to be computed. However, we can use an iterative method to compute H[t] in an online fashion. More precisely, we iteratively update the entropy value for the first amplitudes. By maintaining only

the partial entropy value and the sum of the squared amplitudes, we can compute the final entropy exactly, using the following algorithm :

Algorithm 2 H[t]

Input: A series of amplitudes a_i, one at a time.

Output: $H[t]_f = -\sum_{i=1}^{L} pm_i \log_2 pm_i / \log_2 L$, where $pm_i = \frac{a_i^2}{\sum_{i=1}^{L} a_i^2}$

1: $H \leftarrow 0, sumSquared \leftarrow 0$ ▷ The numbers that we need to store
2: **for** i from 1 to L **do**
3: Acquire a_i, compute a_i^2
4: $H \leftarrow H \times \frac{sumSquared}{sumSquared + a_i^2} - \frac{sumSquared}{sumSquared + a_i^2} \log_2 \frac{sumSquared}{sumSquared + a_i^2} - \frac{a_i^2}{sumSquared + a_i^2} \log_2 \frac{a_i^2}{sumSquared + a_i^2}$
5: $sumSquared \leftarrow sumSquared + a_i^2$
6: **return** $H / \log_2 L$

Proof of the exactitude of Algorithm 2 :

For a sequence $\{a_i\}$ of length n, the entropy is $H_n = -\sum_{i=1}^{n} \frac{a_i^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{a_i^2}{\sum_{i=1}^{n} a_i^2}$. And the entropy H_{n-1}, calculated with respect to the first $n-1$ numbers, is $H_{n-1} = -\sum_{i=1}^{n-1} \frac{a_i^2}{\sum_{i=1}^{n-1} a_i^2} \log_2 \frac{a_i^2}{\sum_{i=1}^{n-1} a_i^2}$.

Therefore,

$$\begin{aligned} H_n - H_{n-1} \times \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} &= -\sum_{i=1}^{n-1} \frac{a_i^2}{\sum_{i=1}^{n} a_i^2} (\log_2 \frac{a_i^2}{\sum_{i=1}^{n} a_i^2} - \log_2 \frac{a_i^2}{\sum_{i=1}^{n-1} a_i^2}) \\ &\quad - \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \\ &= -\sum_{i=1}^{n-1} \frac{a_i^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} - \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \\ &= -\frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} - \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \end{aligned}$$

Hence,

$$H_n = H_{n-1} \times \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} - \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{\sum_{i=1}^{n-1} a_i^2}{\sum_{i=1}^{n} a_i^2} - \frac{a_n^2}{\sum_{i=1}^{n} a_i^2} \log_2 \frac{a_n^2}{\sum_{i=1}^{n} a_i^2}$$

Finally, we can conclude using induction, since for the case $n = 1$ the algorithm is also true (the initial value $H = 0$ is the entropy for a set containing only one number).

Although the exactitude is proven, we still need to verify that the computation is fast enough to actually apply it. As section 2 stated, in addition to the standard built-in $log2f$ function, we have another two alternatives to compute the logarithmic function, which are $fastlog2$ and $fasterlog2$. Among the three choices, some are slow but accurate, while others are fast but inaccurate. In the following paragraphs, we'll examine their usefulness respectively.

log2f The standard logarithmic function used is the built-in $log2f$ in cmath library.

fastlog2 The formula and the implementation is described in section 2.2. The computation now only requires arithmetic operations.

fasterlog2 The formula and the implementation is described in section 2.2. Compared to $fastlog2$, this approximation is much faster as it only requires one multiplication.

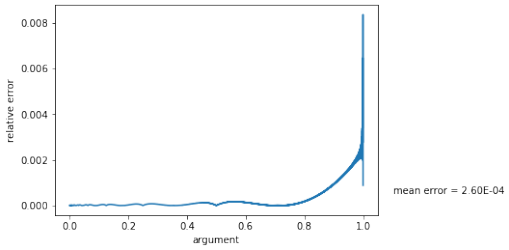
3.5.1 Comparison of Logarithmic Functions

We made comparison of the speed and the relative error (with respect to $log2f$) of the two approximations. We start by examining the relative error :

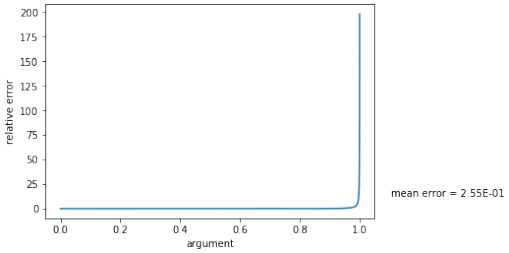
	Interval		
	(0, 1)	(0, 0.001)	(0.999, 1)
$fastlog2$	2.6×10^{-4}	6.63×10^{-6}	1.65×10^{-2}
$fasterlog2$	2.55×10^{-1}	2.04×10^{-3}	3.65×10^{2}

Table 3: Mean relative error comparison

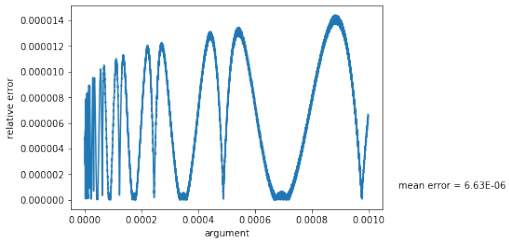
In addition to the average relative error on different intervals, we also show the relative errors as a function of different arguments in the following figures (the left column are errors for $fastlog2$ and the right column are errors for $fasterlog2$) :



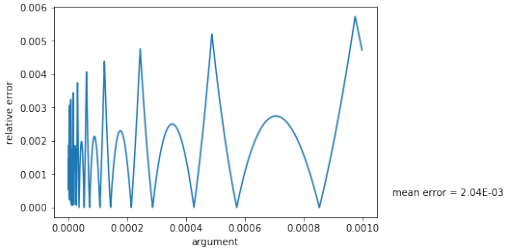
(a) $fastlog2$ on (0,1)



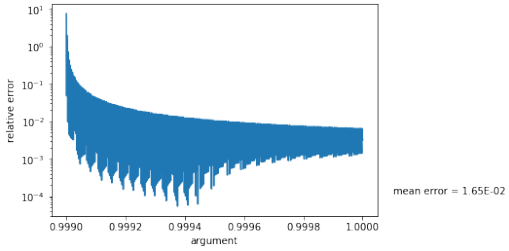
(b) $fasterlog2$ on (0,1)



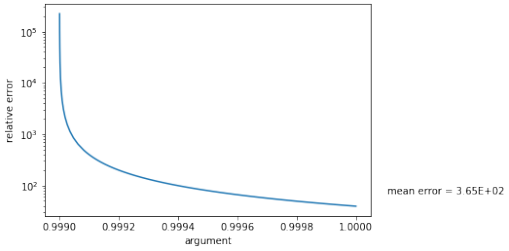
(c) $fastlog2$ on (0,0.001)



(d) $fasterlog2$ on (0,0.001)



(e) $fastlog2$ on (0.999,1)



(f) $fasterlog2$ on (0.999,1)

Figure 7: Relative errors on different intervals

Note that we measured the relative accuracy in three different intervals, this is because the quantities $\frac{sumSquared}{sumSquared + a_i^2}$ and $\frac{a_i^2}{sumSquared + a_i^2}$ are either close to

one or zero experimentally. In the following section, we will describe the influences and how we can improve the original algorithm.

In terms of the speed, we measure the mean clock cycles taken by calling the functions 10000 times. The table below sums up the execution time for the three alternatives :

	Clock cycles	Execution time (ms)
$log2f$	328	0.0069
$fastlog2$	101	0.0021
$fasterlog2$	59	0.0012

Table 4: Speed comparison

The execution time is measured for one call, on a Wonder Gecko starter board with clock frequency = 48 MHz, at which Audiomoth is running. As we can see, the built-in $log2f$ function is 3-6 times slower than the approximations, and $fasterlog2$ is nearly 2 times faster than $fastlog2$. In the following section, we will describe how we choose which approximation to use and why.

3.5.2 Approximated Temporal Entropy

As section 5.2 describes, the exact algorithm cannot be both accurate and fast enough. Even if we make use of the above approximations, we still need to verify that they can be computed fast enough. Basically, regardless of the accuracy, if we use the original algorithm, each update of the variables involves 256 (frequency bins) times 2 log function calls. In terms of time, it corresponds to 1.0752 ms for $fastlog2$ and 0.6144 ms for $fasterlog2$. Remember that there are also other variable updates, and all of the computations need to be done within 10.66 ms. Therefore, we cannot overemphasise the importance of the speed. Two alternatives come up to be viable solutions :

1. Reduce the number of calls of logarithmic functions
2. Use $fasterlog2$ as approximation

Nevertheless, as Table 2 shows, $fasterlog2$ commits a large error when the argument is close to one, which is experimentally very common. We start therefore by examining the feasibility of the first solution. Recall the step where we update H :

$$H \leftarrow H \times fact_i + binaryEntropy(fact_i),$$

Where :

- $fact_i = \frac{\sum_{k=1}^{i} a_i^2}{\sum_{k=1}^{i+1} a_i^2}$
- $binaryEntropy(x) = -x \log_2 x - (1-x) \log_2(1-x)$

We discussed previously possible approximations for logarithmic functions, but more precisely, what we want is an approximation of *binaryEntropy*, since in Algorithm 2, what we care is only its value, not individual logarithmic functions. Thus, a good approximation of *binaryEntropy* is enough, even if we don't use logarithmic functions at all.

To get a such approximation, polynomials are the first alternatives that come to mind. However, since polynomials change sign often, if we want to have a precise one, its degree must be high. Moreover, since *binaryEntropy* is not differentiable at $x = 1$, it is impossible to find a polynomial for which the error maintains the same sign over the segment [0,1]. As a result, it is difficult to keep track of the errors accumulated during the iterations in the algorithm.

Instead, since experimentally most of the $fact_i$'s are close to one, we can actually compare the two terms in $binaryEntropy(x)$ and drop one of them. Essentially,

$$
\begin{aligned}
\lim_{x\to 1} \frac{x\log_2(x)}{(1-x)\log_2(1-x)} &= \lim_{x\to 0} \frac{(1-x)\log_2(1-x)}{x\log_2(x)} && \text{(replace } x \text{ with } 1-x\text{)}\\
&= \lim_{x\to 0} \frac{(1-x)\ln(1-x)}{x\ln(x)} && \text{(divide the numerator and the denominator by } \ln 2\text{)}\\
&= \lim_{x\to 0} \frac{(1-x)\times(-x)}{x\ln(x)} && \text{(since } \ln(1-x) \underset{x\to 0}{\sim} (-x)\text{)}\\
&= \lim_{x\to 0} \frac{-(1-x)}{\ln(x)}\\
&= 0
\end{aligned}
$$

Therefore, dropping $-x\log_2 x$ and only using $approx(x) = -(1-x)\log_2(1-x)$ turns out to be a good approximation. What's better, since the arguments are close to one, the arguments for $\log_2$, which are $1-x$, are now close to zero, where $fasterlog2$ is precise (cf. section 3.5.1, Table 3).

Hence, using $\boxed{approx(x) = -(1-x)fasterlog2(1-x)}$ is fast and precise enough computationally, which actually applies the two possible solutions that we have proposed earlier. We can then apply the following modified algorithm for H[t] :

Algorithm 3 H[t] approximation

Input: A series of amplitudes a_i, one at a time.
Output: An approximate value of the exact H[t]

1: $H \leftarrow 0, sumSquared \leftarrow 0$ ▷ The numbers that we need to store
2: **for** i from 1 to L **do**
3: Acquire a_i, compute a_i^2
4: $H \leftarrow H \times \frac{sumSquared}{sumSquared+a_i^2} - \frac{a_i^2}{sumSquared+a_i^2} fasterlog2 \frac{a_i^2}{sumSquared+a_i^2}$
5: $sumSquared \leftarrow sumSquared + a_i^2$
6: **return** $H/\log_2 L$

Regarding the complexity, for every 512 samples, we need one division to ob-

tain $\frac{sumSquared}{sumSquared + a_i^2}$, and another subtraction $1 - \frac{sumSquared}{sumSquared + a_i^2}$ to obtain $\frac{a_i^2}{sumSquared + a_i^2}$. Then the rest requires three multiplications and one subtraction. At the end, we need one more division to obtain the H[t] value.

Lastly, another advantage of using this approximation is that its error is easy to keep track of. We'll discuss the errors that are accumulated during the iterations in section 4.1.

3.6 Acoustic Cover (CVR)

We move on to the last index, CVR. Recall from section 2, the original CVR is calculated as

$$CVR_f = \#\{i \in [0..L] \mid dB_i - Noise_f > 2\}/L$$

where

- $Noise_f = mean(dB_i) + noiseFactor \times std(dB_i)$
- $dB_i = 20 \log_2(a_i)$
- $noiseFactor = 0.1$

As analysed in the previous sections, logarithmic function are very expensive in terms of time consumed, if we compute the decibel value for each amplitude, we won't have enough time to do other computations. More precisely, if we want the exact decibel values, we need to use the standard $log2f$ function, and calculating the decibel values for all 256 frequency bins takes $256 * 0.0069 = 1.7664$ ms, which is too much along with other updates.

Therefore, in our paper, in order to accelerate the computation (since computing *log* to get dB values is costly), we make a slight modification on the CVR index.

CVR index (modified): Since the conversion to decibel values is computationally expensive, and the choice of the threshold = 2dB seems somehow arbitrary,

we remove the conversion and work directly on the amplitudes, while the concept remains the same :

$$CVR(modified)_f = \#\{i \in [0..L] \mid a_i > Noise_f\}/L$$

To better approach the former one, we set the $noiseFactor$ in the computation of $Noise_f$ to be a higher value : $noiseFactor = 0.3$.

Since we modify the noise as $mean + 0.3 \times std$ of the signal, we are not able to know the noise of the current minute before it actually finishes, so an exact online algorithm doesn't exist. However, we can make the assumption that the environment doesn't change much between each minute, so we can use the noise of the previous minute on the current minute's computations, and use the current minute's noise on the next minute, and so forth.

To compute the noise, we need to keep track of the mean and the standard deviation, which can be computed by storing the sum and the squared sum of the amplitudes. The algorithm is as follows :

Algorithm 4 Modified CVR

Input: A series of amplitudes a_i, one at a time.
Noise level *cvr_noise* of the previous time segment (previous minute)

Output: $CVR_f = \#\{a_i \mid a_i > noise\}/L$
cvr_noise′ used for the next time segment

1: $cvr\_noise, sum \leftarrow 0, sumSquared \leftarrow 0, cvr\_count \leftarrow 0$
▷ The numbers that we need to store
2: **for** i from 1 to L **do**
3: Acquire a_i, compute a_i^2
4: **if** $a_i > cvr\_noise$ **then** $cvr\_count \leftarrow cvr\_count + 1$
5: $sum \leftarrow sum + a_i$
6: $sumSquared \leftarrow sumSquared + a_i^2$
7: $mean \leftarrow \dfrac{sum}{L}$
8: $std \leftarrow \sqrt[2]{\dfrac{sumSquared}{L} - mean^2}$
9: **return** $cvr\_count/L, cvr\_noise = mean + 0.3 \times std$

A slight modification is made on the very first minute of the recording. Since we use the noise of the previous minute in the computation, we need to modify the algorithm for the first minute as there is no corresponding "previous minute". What we do is to split the first minute into two segments of 30 seconds, compute the noise using the first half, and use the second half to compute the CVR index. This is the additional processing that we make at the end of the first 30 seconds as mentioned in section 3.1. The algorithm works the same for all the rest of time.

The update of the noise to be used in the next minute is only required at the end of the current minute, where the algorithm requires two multiplications, two additions, two divisions by N and one square root. For other moments in that minute, only one multiplication and two additions are needed. As stated previously, since L is a constant, we can compute $1/L$ once at the beginning of the recording, and use the multiplication by $1/L$ instead of the division by L in the following

updates. With this optimisation, this algorithm takes at most two additions, four multiplications and one square root operation.

3.7 Space of Variables Needed

Some of the above variables used in different algorithms are actually the same; to sum up, we need in total 7 variables : $aci\_previous$, $aci\_sumDiff$, sum, $sumSquared$, H, cvr_noise and cvr_count, each of which is an floating point array of length 256 (number of frequency bins). The total space occupied is therefore $7 * 256 * 4$ bytes = 7kB, which can be stored in the external SRAM of size 256kB that is linked by the EBI (external bus interface) peripheral of the device.

To make a comparison, if we don't use online algorithms and store all the data, it would require $L * 256 * 4 = 5625$kB of memory space, which the device cannot handle.

3.8 Computation Speed

In terms of the speed, as we mentioned at the beginning of the section, we need to guarantee that for every 512 samples, all updates required (including the computation of the FFT), finishes within 10.66 milliseconds. Along with an FFT of size 512, experimentally, the average operation time for every 512 samples is 5.41 ms, where FFT takes 1.21 ms and all the updates in the algorithms take 4.20 ms. The actual computation is therefore fast enough to be finished before the next piece of data becomes available, which means we can employ the above algorithms without worrying about discontinuous data.

4 Evaluations

The description of the algorithms is finished. We have made some modifications on the original ones, and we have to discuss how much they differ from the original ones. In this section, we will calculate their theoretical and experimental errors respectively.

We start by examining the theoretical errors of the online algorithms. Actually, the algorithm that computes ACI is exact, which means it has theoretically zero error. As for CVR, since in terms of computation, we adapted it in a quite different way than the original one (not using the decibel values), it is impossible to calculate the theoretical error for it. What remains is the H[t] index, and as we only replaced the time-consuming $log2f$ function by an approximation, $fasterlog2$, we are actually able to calculate its theoretical error mathematically. We will now describe the detailed calculation in the following section.

4.1 Theoretical Error of Approximated Temporal Entropy

In section 3.5.2, we established a modified algorithm to compute the temporal entropy index, H[t], approximately. We showed that if the argument is close to one, then the approximation commits barely any error. Recall the comparison of the two terms in *binaryEntropy* :

$$\lim_{x \to 1} \frac{x \log_2(x)}{(1-x)\log_2(1-x)} = 0$$

And :

- $binaryEntropy(x) = -x \log_2 x - (1-x)\log_2(1-x)$
- $approx(x) = -(1-x)\log_2(1-x)$
- $fact_i = \frac{\sum_{k=1}^{i} a_i^2}{\sum_{k=1}^{i+1} a_i^2}$

However, what the limit tells us is only that for **one** iteration, *approx* is close to *binaryEntropy* if the argument is close to 1. To examine the overall accuracy of this

approximation, we need to look at the final error, which is accumulated through **all** iterations.

To begin with, let's compare the modified algorithm to the original (exact) one :

$$H_{exact} \leftarrow H_{exact} \times fact_i + binaryEntropy(fact_i)$$

$$H_{approx} \leftarrow H_{approx} \times fact_i + approx(fact_i)$$

We can then deduce that the error $\Delta_n = (H_{approx} - H_{exact})_n$ for the n^{th} step follows the following recurrence relation :

$$\begin{cases} \Delta_0 = 0 \\ \Delta_n = \Delta_{n-1} \times fact_n + \Delta(fact_n) \end{cases}$$

where $\Delta(x) = approx(x) - binaryEntropy(x) = x \log_2(x)$.

We assume here that $fact_n$'s are all close to one when $n \geq n_0$ (which is almost true experimentally, since $fact_n$ will only decrease much during distinguishable acoustic events, which don't happen frequently). In other words, for $n \geq n_0$, we pose $fact_n = c = 1 - \delta$, where δ is a small value, usually of order 10^{-3}.

We replace $fact_n$ in the recurrence relation for $n \geq n_0$:

$$\Delta_n = c\Delta_{n-1} + \Delta(c) \tag{1}$$

To get an approximation of $\Delta(x)$ when x is close to one, we can use a Taylor expansion since Δ is of class C^∞ over $(0, \infty)$. In particular,

$$\Delta(x) = \frac{x-1}{\ln 2} + o((x-1)) \qquad \text{when } x \to 1$$

developed only to the first order. If we replace this polynomial approximation in (1), then for $n \geq n_0$:

$$\Delta_n = c\Delta_{n-1} + \frac{c-1}{\ln 2} \tag{2}$$

To solve this recurrence relation, it suffices to rewrite it as

$$\Delta_n + \frac{1}{\ln 2} = c(\Delta_{n-1} + \frac{1}{\ln 2}) \tag{3}$$

Therefore

$$\Delta_n = c^{n-n_0}(\Delta_{n_0} + \frac{1}{\ln 2}) - \frac{1}{\ln 2} \tag{4}$$

Theoretically, since $c < 1$, the limit of Δ_n is $-\frac{1}{\ln 2}$. As the index H[t] is defined as $H/\log_2 L$ (cf. Algorithms 2 and 3), we expect that the final error of the approximation is always $\frac{\Delta_n}{\log_2 L} = -\frac{1}{\ln 2 \cdot \log_2 L} = -\frac{1}{\ln L}$, no matter what n_0 and Δ_{n_0} are. Actually, recall that $c = 1 - \delta$, where $\delta \approx 10^{-3}$, this means that if $n - n_0$ is of order 10^3, c^{n-n_0} is going to be approximately $e^{-1} \approx 0.3678$, since $\lim_{n\to\infty}(1 - \frac{1}{n})^n = e^{-1}$. In reality, n is at most L, which equals 5625 (in case the sample rate is 48kHz), so the value c^{n-n_0} depends highly on how fast $(fact_i)$ tends to one, in other words, how small n_0 is.

For example, if $n - n_0$ is only about 1000, then c^{n-n_0} will not be much less than 0.3678; in contrast, if $n - n_0$ is about 3000 or 4000, then we can expect that c^{n-n_0} to be as small as $e^{-3} \approx 0.05$, which makes the error Δ_n converge to $-\frac{1}{\ln L}$ given that $\Delta_{n_0} + \frac{1}{\ln 2}$ is not too big. We are still uncertain about the value of $\Delta_{n_0} + \frac{1}{\ln 2}$, but $-\frac{1}{\ln L}$ could be one possible limit of the error.

Moreover, we can get an asymptotic behaviour of the error Δ_n. If we replace c with $1 - \delta$ in (4), and use the binomial series truncated to the first order $(1 - x)^n = 1 - nx + o(x)$ when $x \to 0$, we get :

$$\begin{aligned}
\Delta_n &= (1 - \delta)^{n-n_0}(\Delta_{n_0} + \frac{1}{\ln 2}) - \frac{1}{\ln 2} \\
&= (1 - \delta(n - n_0))(\Delta_{n_0} + \frac{1}{\ln 2}) - \frac{1}{\ln 2} + o(\delta) \\
&= \Delta_{n_0} - \delta(\Delta_{n_0} + \frac{1}{\ln 2})(n - n_0) + o(\delta)
\end{aligned}$$

Therefore, the error of H[t] is also expected to behave as a straight line asymptotically, with slope $\delta(\Delta_{n_0} + \frac{1}{\ln 2})/\log_2 L$.

To explore what the actual limit is, we need to examine several test data in order to confirm. This will be explained in the following section.

4.2 Experimental Results

In this section, we will check the actual errors of the online algorithms that we established through sections 3.4-3.6 with respect to the original (offline) ones, on the recordings that we made in the surroundings of St. Catherine's College and New Forest.

The recordings are of length 1 hour, and the indices are computed every minute, unless otherwise specified. The date is in the format YYYY/MM/DD HHh in time zone UTC.

We use some additional abbreviations in this section in order to make the figures concise, which are summarised in the following table :

Abbreviation	Description
MRE	Mean Relative Error (with respect to the offline algorithm)
H[t]_e	The H[t] index obtained by Algorithm 2 (exact online algorithm)
H[t]_a	The H[t] index obtained by Algorithm 3 (approximate online algorithm)
H_err	H[t]_a − H[t]_e
p_offline	The values obtained by offline algorithms coded in Python
p_online	The values obtained by online algorithms coded in Python

Table 5: Abbreviations

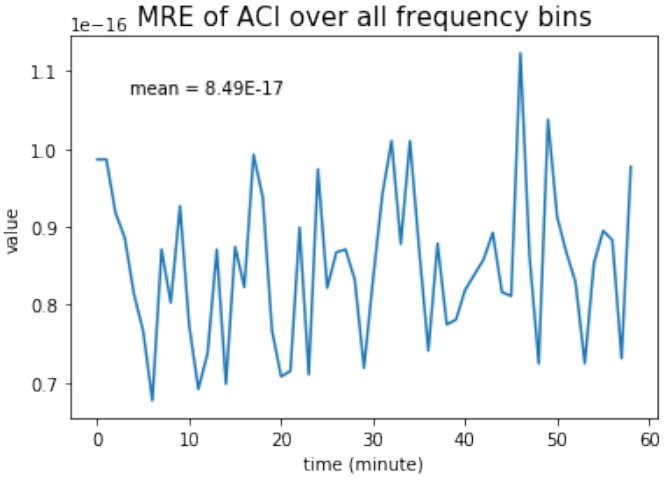
We start by checking Algorithms 1 (ACI) and 2 (H[t]_e), which are supposed to be exact compared to their offline counterparts.

4.2.1 Exact Algorithms (ACI, H[t]_e)

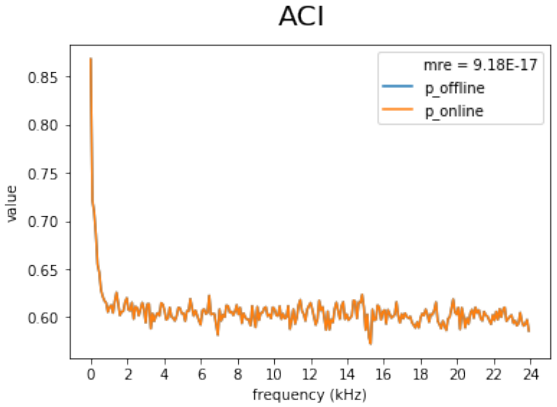
For every minute, we calculate the MRE over all 256 frequency bins of that minute, and plot the value against the time. We also compute the mean value of the MREs over all 60 minutes (shown as text in the figure). Finally, we randomly select two specific minutes and look at the MRE versus different frequencies.

Following are the results for indices ACI and H[t]_e for different recordings :

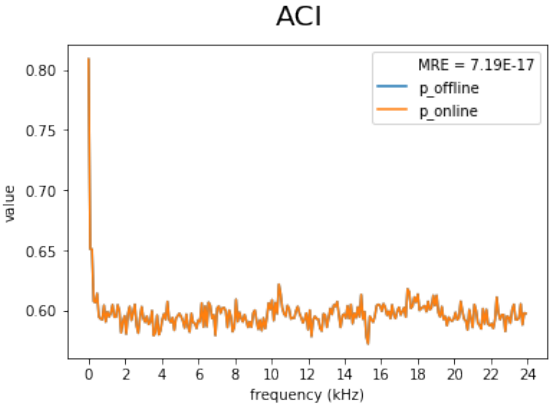
- **2017/08/04 02h, St.Catherine's College**



(a) MRE over all frequency bins

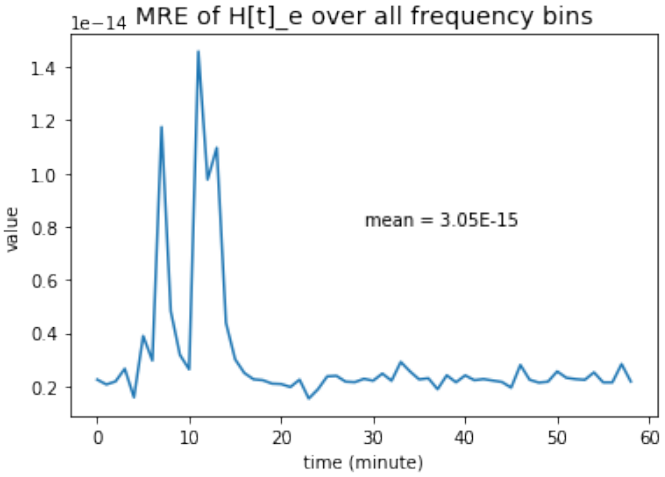


(b) Values at the 3rd minute

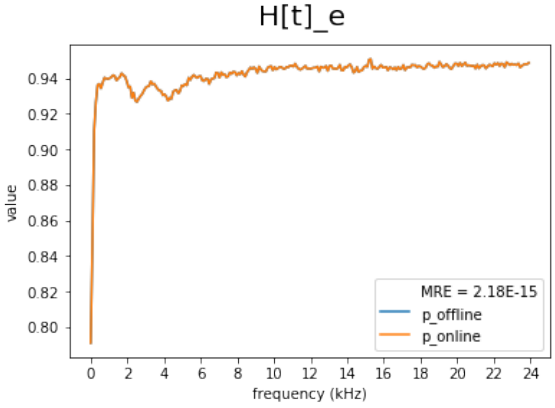


(c) Values at the 30th minute

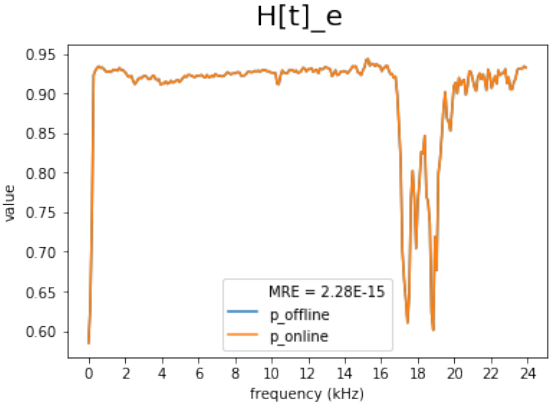
Figure 8: ACI, 2017/08/04 02h, St.Catherine's College



(a) MRE over all frequency bins



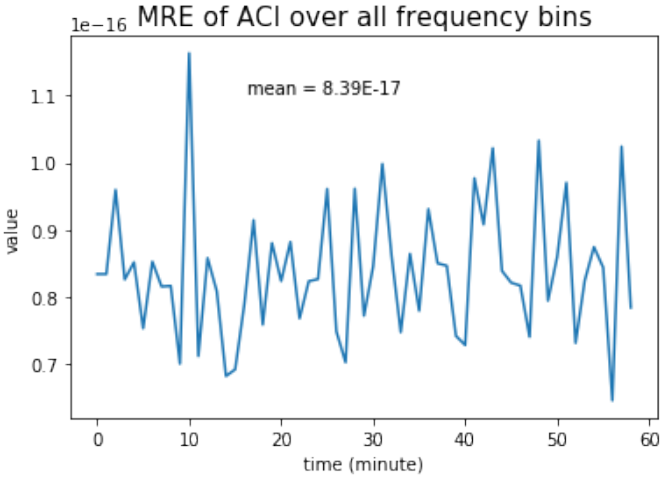
(b) Values at the 3rd minute



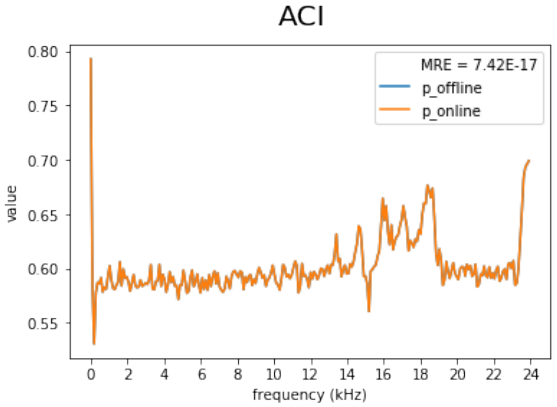
(c) Values at the 30th minute

Figure 9: H[t]_e, 2017/08/04 02h, St.Catherine's College

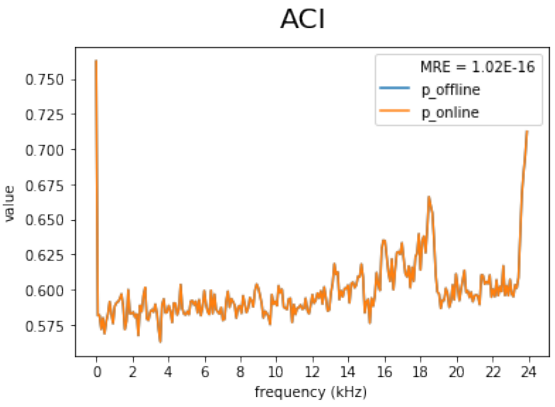
- **2017/08/06 13h, New Forest**



(a) MRE over all frequency bins

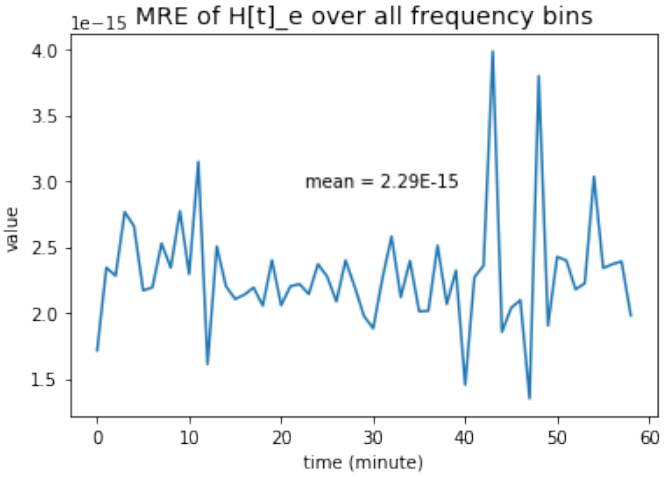


(b) Values at the 40th minute

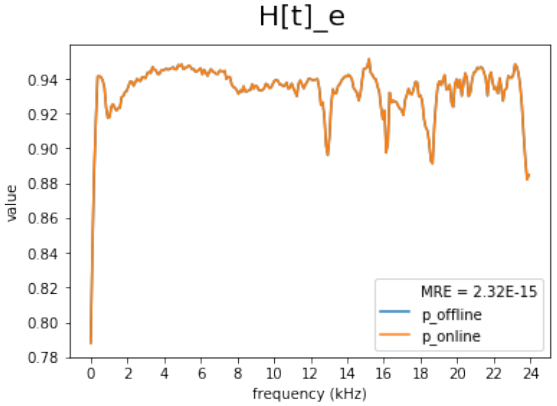


(c) Values at the 58th minute

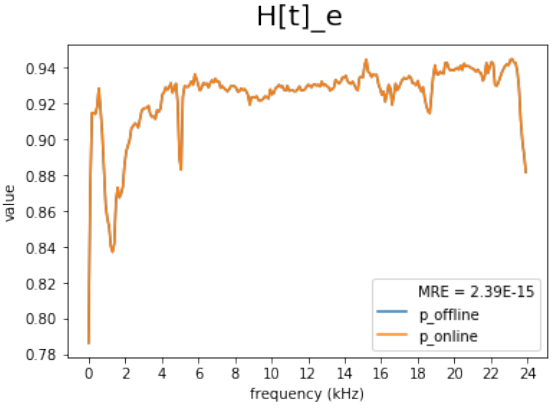
Figure 10: ACI, 2017/08/06 13h, New Forest



(a) MRE over all frequency bins



(b) Values at the 40th minute



(c) Values at the 58th minute

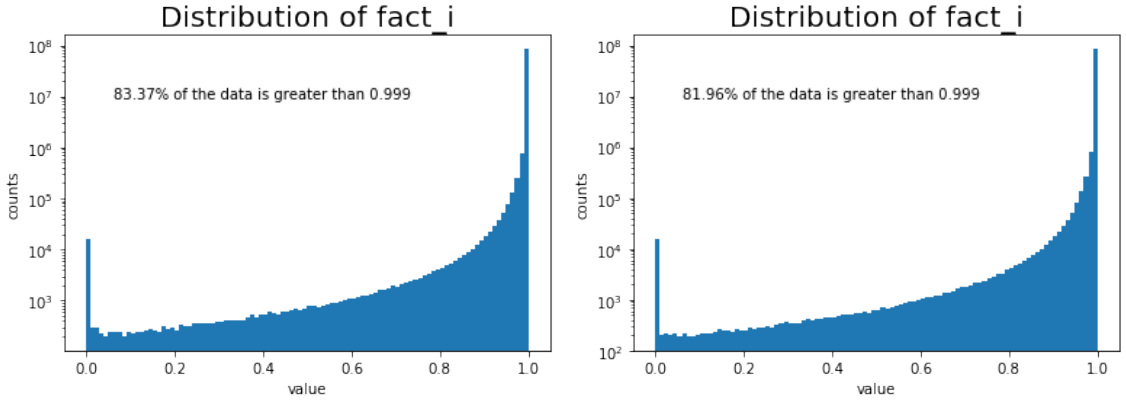
Figure 11: H[t]_e, 2017/08/06 13h, New Forest

As we expected, the exact algorithms don't commit any error.

4.2.2 H[t] Approximation

We will now examine the error committed by Algorithm 3 (H[t] approximation), to see if it corresponds to the theoretical error that we established in section 4.1.

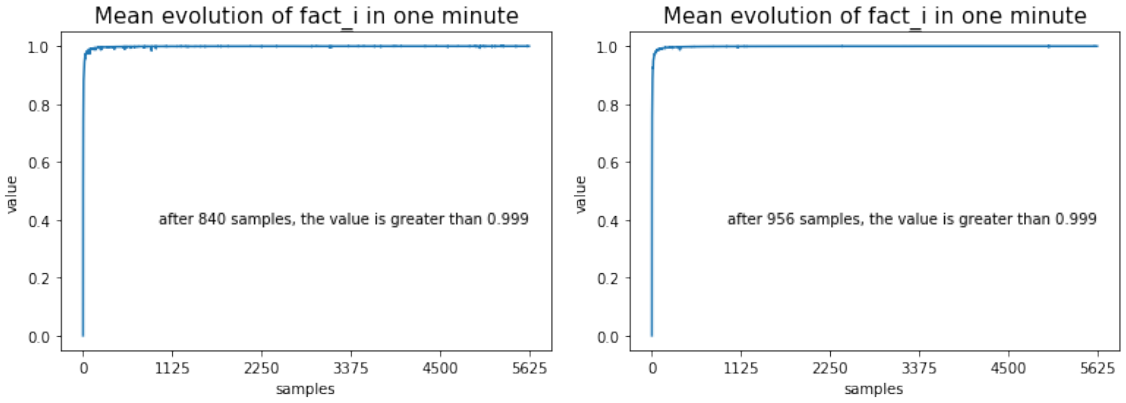
First, we need to verify the assumption that we made : $fact_i = \frac{\sum_{k=1}^{i} a_i^2}{\sum_{k=1}^{i+1} a_i^2}$ are close to one for most of the i's. We plot the distributions of $fact_i$ for different data :



(a) 2017/08/04 02h, St.Catherine's College (b) 2017/08/06 13h, New Forest

Figure 12: Distributions of fact_i

We also plot the mean evolution of $fact_i$ over one minute ($L = 5625$ values) :



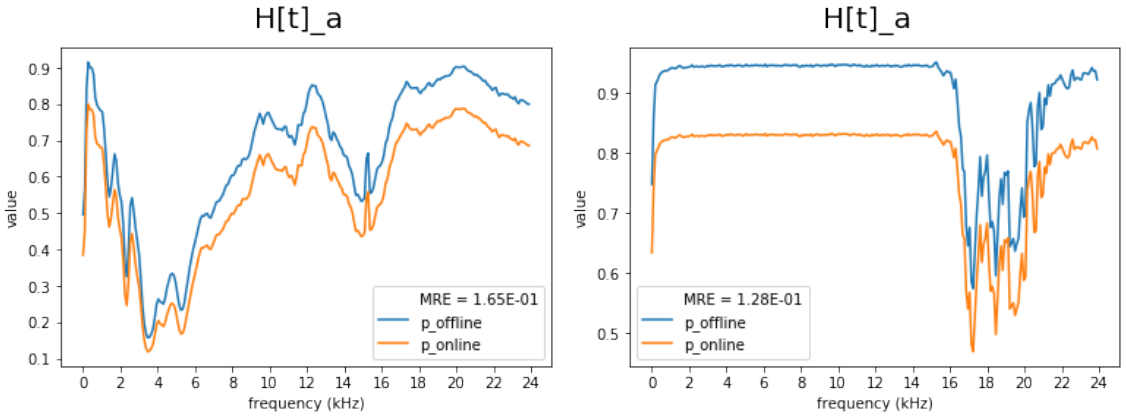
(a) 2017/08/04 02h, St.Catherine's College (b) 2017/08/06 13h, New Forest

Figure 13: Mean evolution of fact_i

To sum up, the convergence of $fact_i$ towards 1 is quite fast, and as a result, most of the $fact_i$'s are indeed very close to one, which confirms the assumption that we

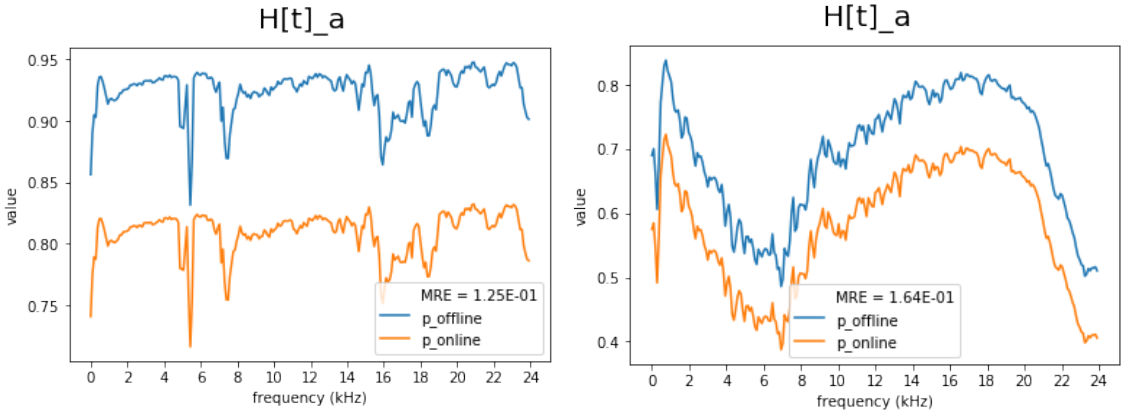
made earlier in the algorithms.

Hence, we can expect that the actual error of H[t]_a also follows the formula that we established in section 4.1. We start by comparing H[t]_a with the exact value given by the offline algorithm :



(a) Values at the 10th minute (b) Values at the 42nd minute

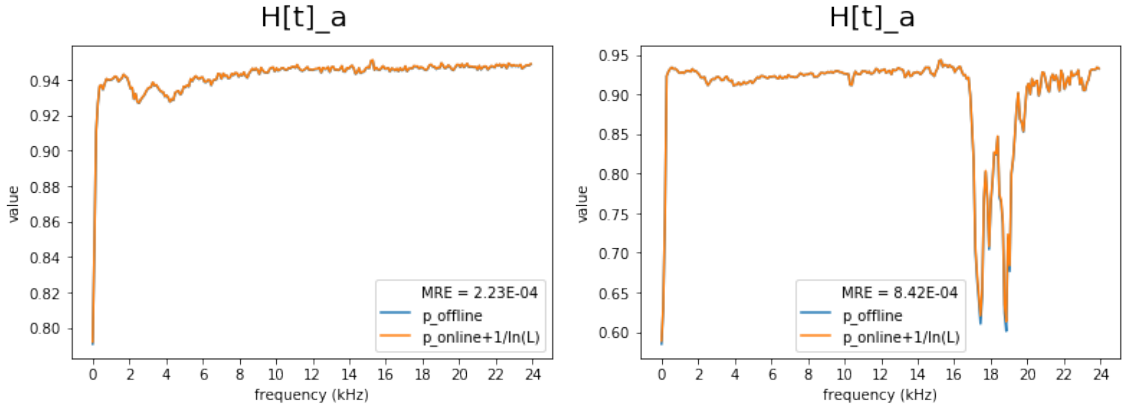
Figure 14: H[t]_a, 2017/08/04 02h, St.Catherine's College



(a) Values at the 5th minute (b) Values at the 28th minute

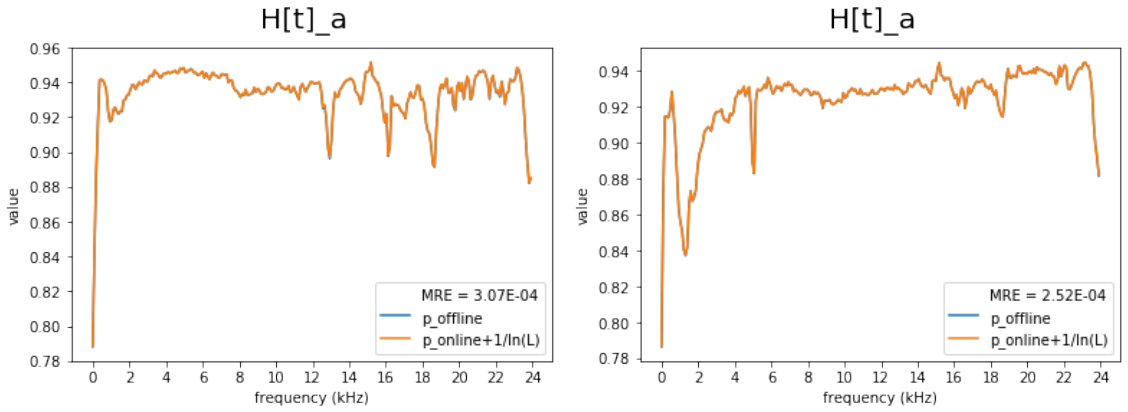
Figure 15: H[t]_a, 2017/08/06 13h, New Forest

The results yielded by the online algorithm (Algorithm 3) seem to have the same form, only slightly translated when compared to the exact value. Recall that in section 4.1, we stated that one possible limit of the error could be $-\frac{1}{\ln L}$, provided that n_0 and $\Delta_{n_0} + \frac{1}{\ln 2}$ are both small. From Figure 13, we know already that $n_0 \approx 1000$, which is small. Therefore, it is reasonable to guess that the limit of the error is actually attained. We start thus by adding $\frac{1}{\ln L}$ to H[t]_a and do the comparison again :



(a) Values at the 3rd minute (b) Values at the 30th minute

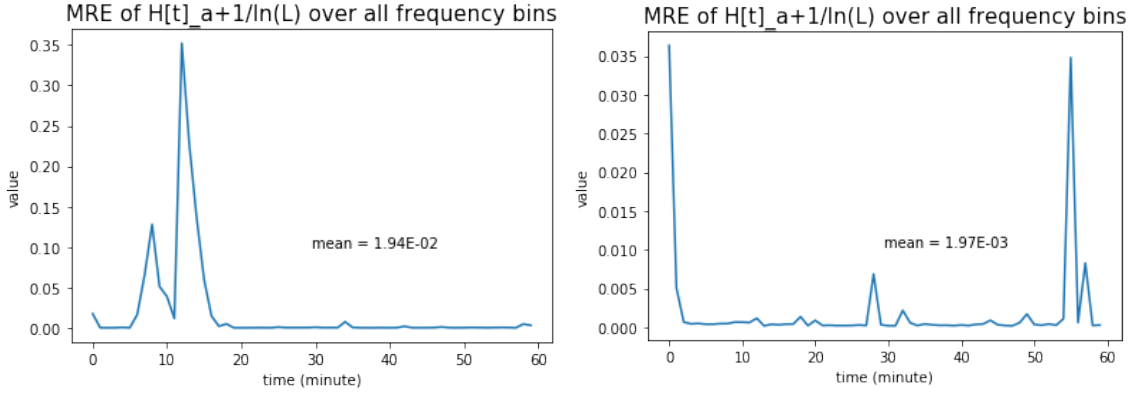
Figure 16: H[t]_a, 2017/08/04 02h, St.Catherine's College



(a) Values at the 40th minute (b) Values at the 58th minute

Figure 17: H[t]_a, 2017/08/06 13h, New Forest

The values match very well across multiple data and at different moments, which confirms that the theoretical error indeed is very close to the real error. We also plot the MRE across the time and all frequency bins to show that, after adding $\frac{1}{\ln L}$ to H[t]_a, the results yielded by the online algorithm matches that of the offline one :



(a) 2017/08/04 02h, St.Catherine's College (b) 2017/08/06 13h, New Forest

Figure 18: MRE of H[t]_a+1/ $\ln L$

We can also verify our second conjecture established previously, that the error

H_err = H[t]_a − H[t]_e follows a straight line when n is sufficiently large.

We start by plotting the mean evolution (over the frequency bins and the time) of H_err in one minute ($L = 5625$):

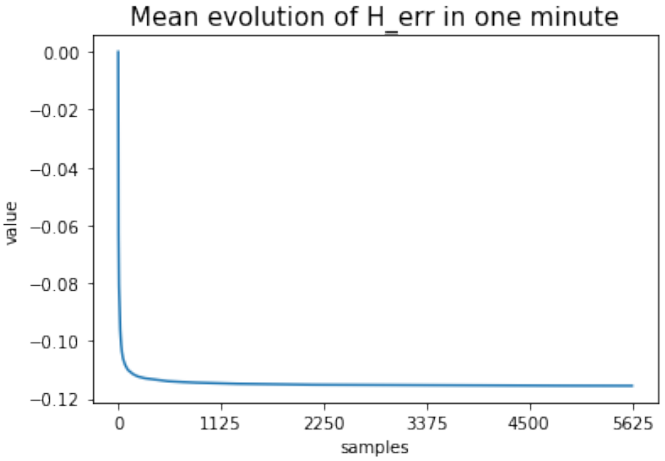
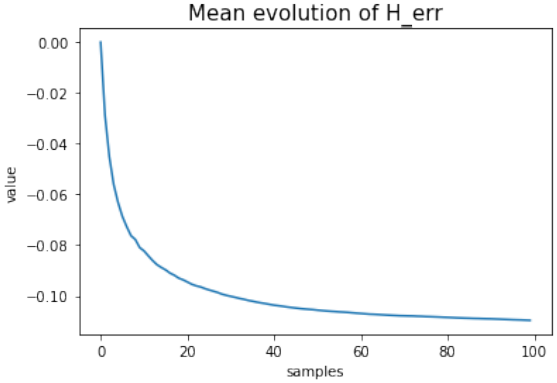
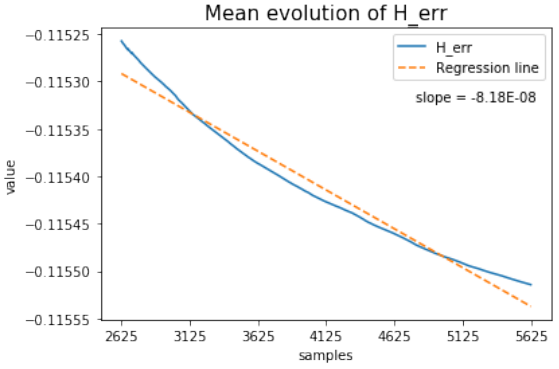


Figure 19: H_err, 2017/08/04 02h, St.Catherine's College

If we zoom in to see different parts of the figure :



(a) The first 100 samples



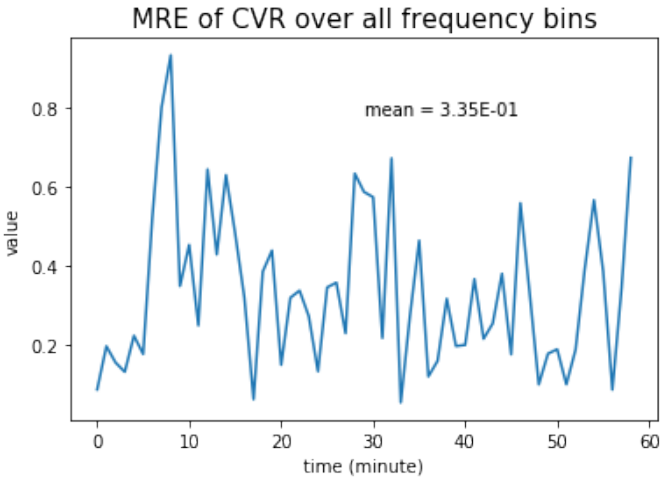
(b) The last 3000 samples

Figure 20: H_err, 2017/08/04 02h, St.Catherine's College

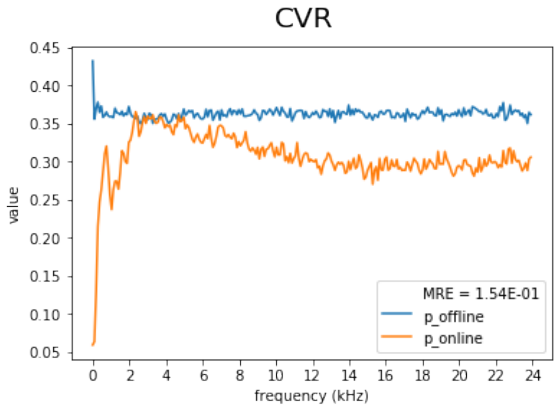
We can see that the error approaches its limit, $-\frac{1}{\ln L} = -0.1158$ very quickly, after 100 samples, it is already close enough to the order 10^{-3}. The right figure shows what we predicted theoretically, that the curve is almost a straight line when n is large (the regression line of the curve is shown in orange).

4.2.3 CVR

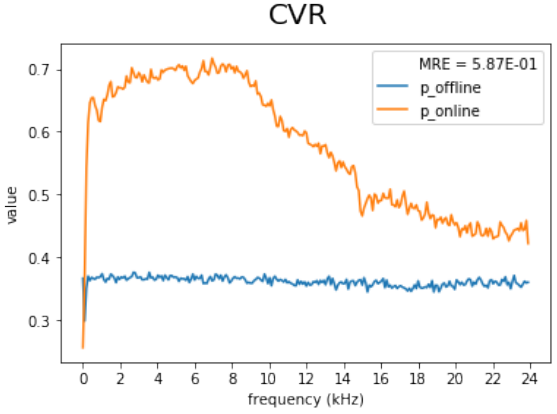
The CVR algorithm is quite different from the original (offline) one, we don't have any theoretical comparisons. We plot the same comparisons as in section 4.2.1 of different data :



(a) MRE over all frequency bins



(b) Values at the 3rd minute



(c) Values at the 30th minute

Figure 21: CVR, 2017/08/04 02h, St.Catherine's College

The CVR indices obtained by the online algorithm differ a lot from the exact values, this is probably because our assumption, that the current minute's noise is the same as the previous minute's, doesn't hold, as the environment is constantly changing. We will discuss how we can improve the calculation in the next section.

4.2.4 False Color Images

As we mentioned in section 2, the authors of [5] presented a way to visualise the recordings with the aid of these indices. We are going to reproduce these images using the results obtained by online algorithms as well.

The indices are first normalised by dividing by their maximum values respectively. We plot the indices using grayscale images, and finally combine them to produce RGB images. In our implementation, red, green, and blue correspond to ACI, H[t], CVR respectively. Note that, although we showed in section 4.2.2 that we had to add $\frac{1}{\ln L}$ to H[t]_a to get the exact value, we didn't do it here since the color scale will not be as clear as the current one for human eyes. Following are some figures for recordings (of different durations, from 6 hours to 12 hours) collected at St. Catherine's College and New Forest :

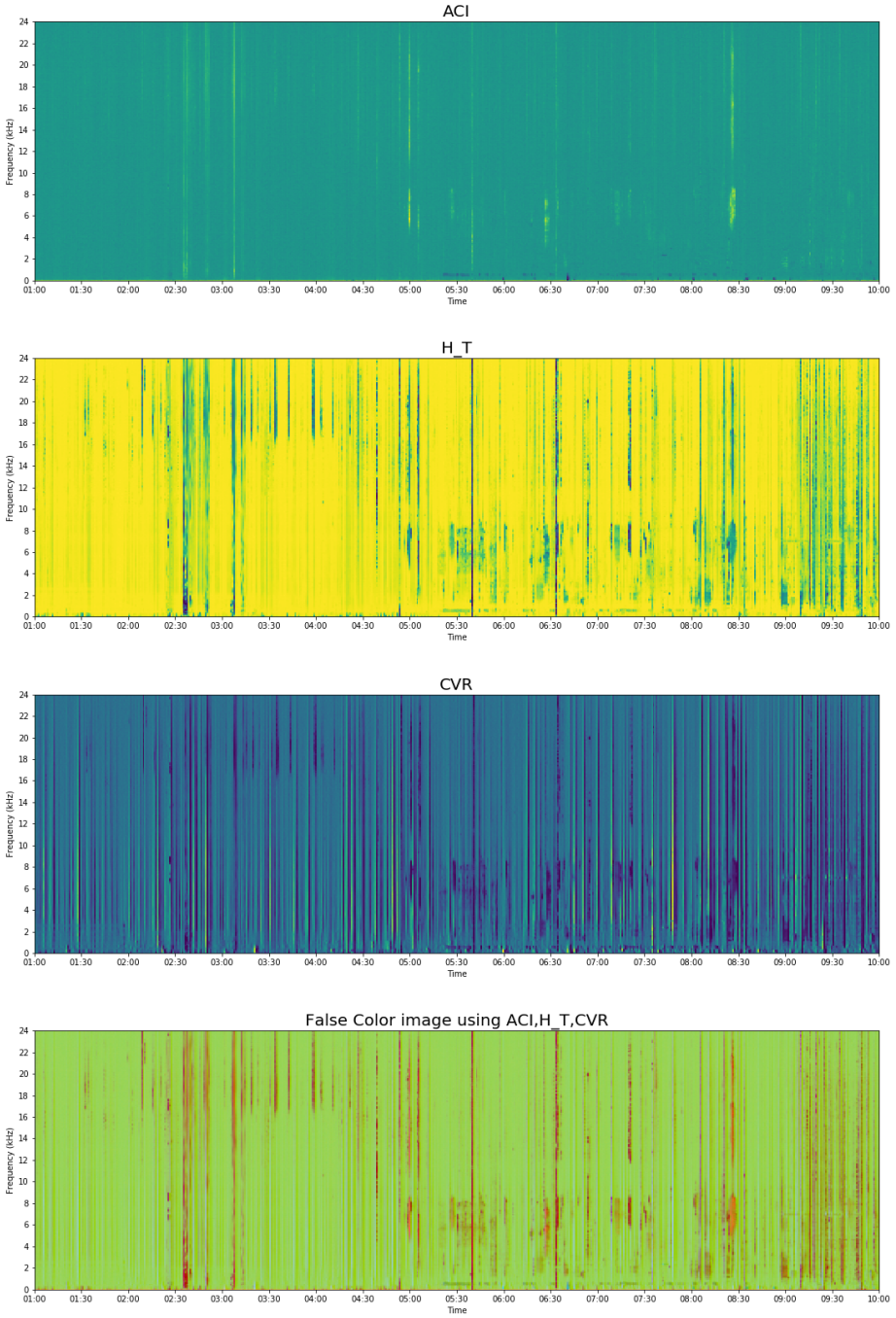


Figure 22: False color image, 2017/08/04 00h, St.Catherine's College

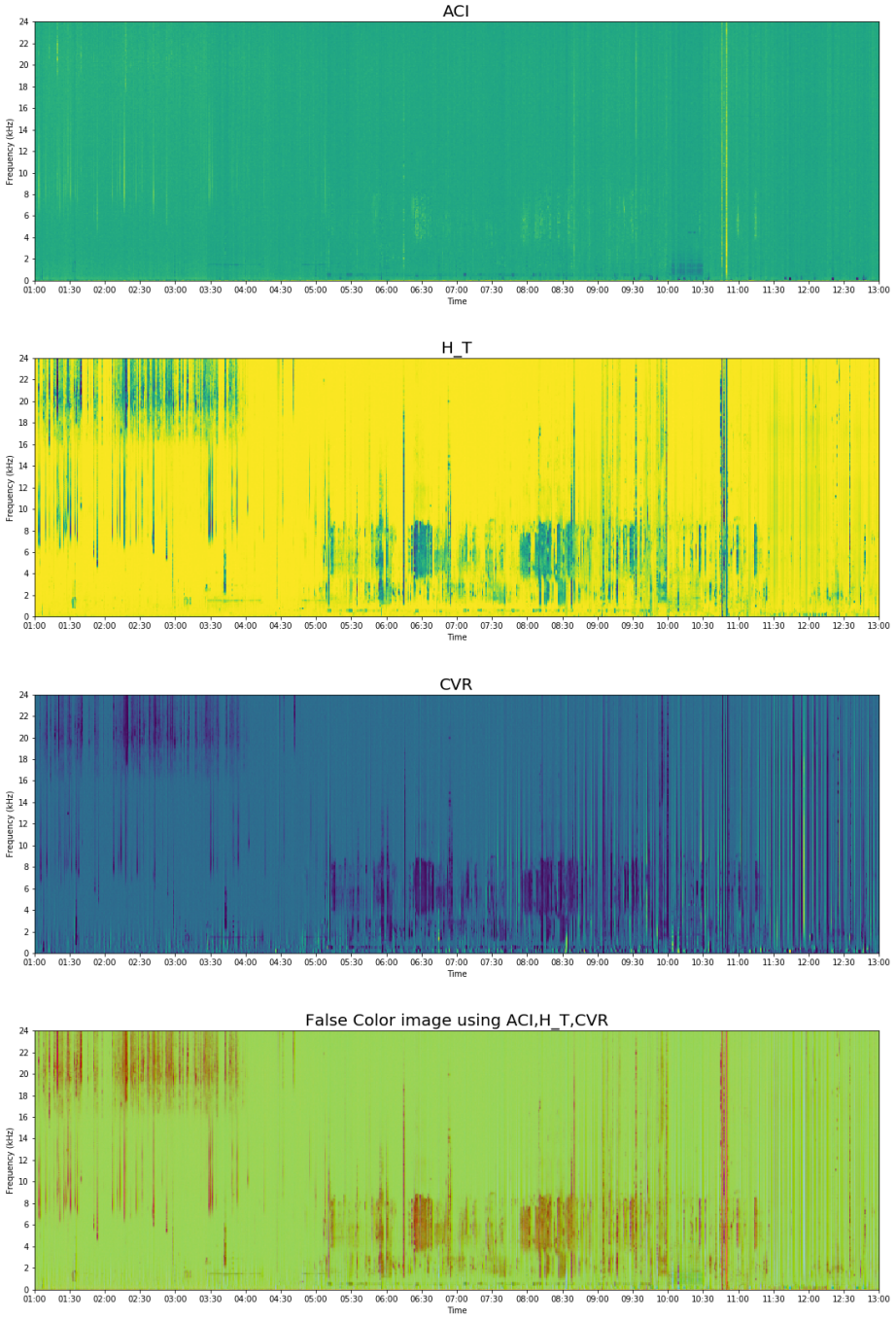


Figure 23: False color image, 2017/08/06 00h, St.Catherine's College

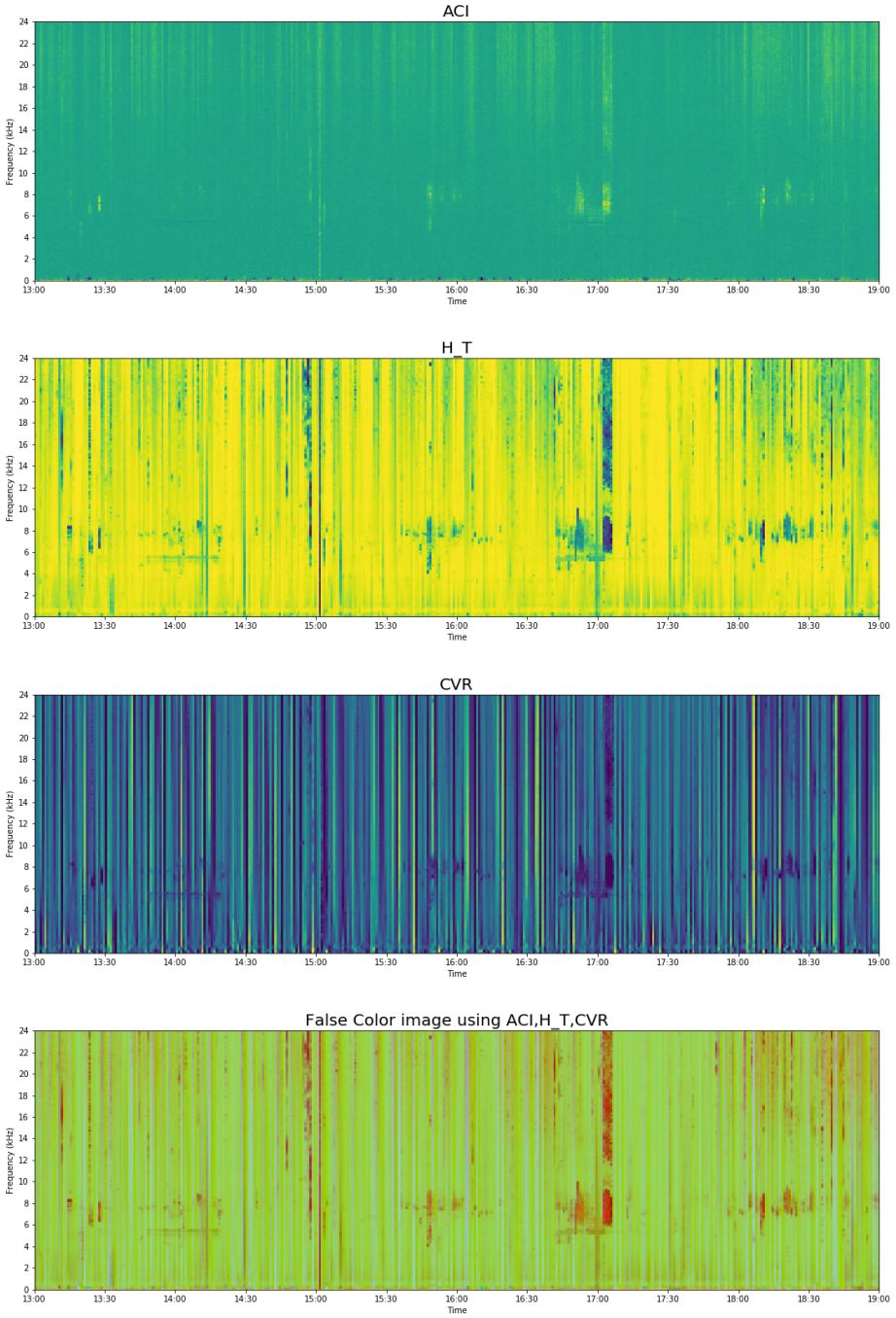


Figure 24: False color image, 2017/08/06 12h, New Forest

We can see many interesting features in the false color images, such as different species singing at night and in the daytime, the peak frequencies at different moment also tell us which species - birds, insects or bats, and so on - are the sources of the sound.

5 Conclusions and Future Work

5.1 Summary

In this paper, we established online algorithms that compute the indices described in [5]. We first examined the feasibility of conceiving exact online algorithms, or made some modifications in case this is impossible. Taking the constraints of the device into consideration, we then made every computations optimised, adapted approximations in order that the calculations be done in time, and make sure we use the least memory space possible. Finally, we did research on the theoretical and experimental errors on the algorithms, and produced false-color images as described [5].

Compared to the actual audio recordings, the false-color images that we obtain by the online algorithms show very well-aligned stripes or spots that represent acoustic events, and allow us to identify the acoustic sources that are present at different moments. These images are already very similar to the ones that are presented in [5], and also very informative.

Therefore, we aim to deploy the devices implemented with these online algorithms, instead of saving all the audio recordings, to explore the soundscape of the wild. In this way, not only can we economise the energy, reduce the data amount - which also saves money in terms of the SD card capacity that is required -, but we can also exploit the features hidden behind the conventionally-saved audio data much more easily.

5.2 Future Work

The main issue that remains unsolved is the usefulness CVR index. We made an assumption that the noises will stay the same during consecutive minutes, but this could be wrong. One possible solution is to use the exact original formula, but other variations could also be possible, since the computations for other indices are already very fast (they only take 5.41 ms while we have 10.66 ms).

Also, because the memory space used to do the computation and store the indices are very small (7kB for the variables and 32kB for the indices, in total 39kB out of 288kB available), we can use this space to store other useful data. For example, use the space to store some features of a given sound source, so we can perform sound detection using the similarities between the indices in an online way. Many applications are currently being implemented, such as gun detection or insect detection.

Last but not least, as the device is re-programmable, we can envision adding more online algorithms for more acoustic indices, and let the user choose which ones to compute according to different uses.

6 Reflections

In this section, I would like to state some of the reflections that I had during the project. At the beginning, I was completely new to Audiomoth's hardwares and Simplicity Studio. The most challenging thing when dealing with hardwares with limited specs is that I need to understand every single operation, and do optimisation everywhere. I learned to replace division with multiplication, substitute modulo operations with counters, and optimise the $log2f$ function with approximations that only involves arithmetic operations.

When working on the approximation of H[t], as stated in the corresponding section, I fist tried to use polynomials with high degrees as approximation, but it turns out that I didn't have any way to control the error that accumulates. After changing it to the current one used in the algorithm, I find that using basic Taylor expansion allows me to deduce the total error easily.

To sum up, this project allowed me to explore things that involve hardwares and signal processing, which I have never touched before, also I learned to construct algorithms that do the job while respecting the constraints that are posed by the hardwares.

References

[1] R. D. Gregory, A. v. Strien. Wild bird indicators: Using composite population trends of birds as measures of environmental health. *Ornithological Science*, 9(1):3–22, 2010.

[2] B. C. Pijanowski, A. Farina, et al. What is soundscape ecology? an introduction and overview of an emerging new science. *Landscape Ecology*, 26(9):1213, 2011.

[3] E. P. Kasten, S. H. Gage, et al. The remote environmental assessment laboratory's acoustic library: An archive for studying soundscape ecology. *Ecological Informatics*, 12:50–67, 2012.

[4] J. Wimmer, M. Towsey, et al. Sampling environmental acoustic recordings to determine avian species richness. *Ecological Applications*, 23:1419–1428, 2013.

[5] M. Towsey, L. Zhang, M. Cottman-Fields, J. Wimmer, J. Zhang, P. Roe. Visualization of long-duration acoustic recordings of the environment. *Procedia Computer Science*, 29:703–712, 2014.