

# Electrical Network Frequency Recording using a Raspberry Pi 3 Model B

Christos Frantzolas

Department of Electrical & Computer Engineering  
University of Patras  
Patras, Greece  
up1053706@upnet.gr

**Abstract**—The Electrical Network Frequency (ENF) Criterion is a forensic technique used to identify the authenticity of a digital recording. When using this method, frequency changes are compared between the background utility hum in the evidence and long-term records of the ENF. Recording this frequency (also called mains frequency or power line frequency) can be performed with the usage of a Raspberry Pi - a small single-board computer. The device's low cost and portability present a great advantage, but the limited computational power and storage capacity create unique problems on how to compute and store the ENF recordings. In this report, a solution is presented, in which the utility signal is first recorded through the audio port of the device, and the EN instantaneous frequency is computed using the Hilbert transform.

**Keywords**—ENF; Raspberry Pi; Complex Trace Analysis; Python

## I. INTRODUCTION

Wide area synchronous grids operate at a specific frequency (50 Hz in Europe, 60 Hz in the USA). This frequency, called the Electrical Network Frequency (ENF) or power line frequency, fluctuates through time due to the changing unbalances in the demand and production of electrical energy. Within the same electrically tied network, the fluctuations pattern is the same. As a result, all the loads in the electrical network will be supplied with an AC voltage that exhibits the same patterns. [1]

Digital recording equipment connected to the electrical grid can pick up the ENF and its harmonics. Thus, every audio recording is coupled with the patterns that were simultaneously present in the ENF. By using a band pass filter on the audio signal to isolate the background utility hum component, and comparing the result to a database of ENF recordings, we can identify the exact date and time of the audio signal recording, as well as any major discontinuities in the evidence. [2]

To create the database of ENF recordings, a device must be connected to the grid in question year-round to monitor the AC voltage signal. Therefore, this monitoring of the ENF requires the continuous commitment of modest computational resources. A personal computer can perform this task for years reliably, but this represents an underuse of its processing power. There are smaller and cheaper devices that can commit to this single task.

The Raspberry Pi Model 3 is a fitting example of such a device. The Raspberry Pi is a small single-board computer hailed for its affordability and portability. The manufacturer of this computer provides the Raspberry Pi OS (formerly Raspbian), a Debian based Linux distribution and promotes the use of Python as its main programming language [3][4].

The use of a Raspberry Pi for ENF recordings presents some challenges related to the hardware constraints of the device, processing power, storage space and accuracy. In this report, a new approach will be analyzed on dealing with these various problems, to create a reliable, accurate and computationally efficient Python application to record the ENF using a Raspberry Pi Model 3.

## II. THE ORIGINAL APPLICATION

This project was primarily inspired by the work of a team from the Department of Electrical and Computer Engineering at the University of Patras, who undertook the task of creating an application for ENF recording using a personal computer and a custom-built specialized sensing circuit, which can be seen in Fig. 1 [5][6].

The original application recorded the EN voltage signal through a modified power supply unit that lowered the voltage from the electrical socket and fed it to the microphone port of a laptop. The signal was sampled at 1000 Hz, using the PyAudio module of Python. Each sample was represented by a 16-bit signed integer number. The resulting digital signal was normalized and then saved as a .wav file using the Python library wave. Each recording lasted approximately an hour and there was a small gap between recordings because the recording and storing of the file were executed consecutively by a single thread.

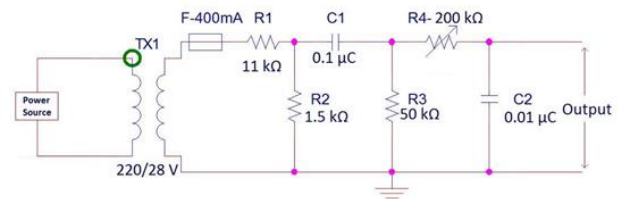


Fig. 1. Schematic diagram of the original sensing circuit [5].

Each hourly recording had a size of about 6.87 MB (1000 samples per second  $\times$  3600 seconds per hour  $\times$  2 Bytes per sample). When the program detected that the time was midnight, it created a zip archive of the previous day's recordings and uploaded it automatically to the cloud.

#### A. Problems with the previous approach

Despite the above-mentioned application's success at the task of ENF recording, there were clear drawbacks to that approach. First and foremost, the use of a laptop as a recording device committed more computational resources than needed to the task. Furthermore, the application only recorded the utility network signal, while the frequency had to be extracted separately. As a result, the recordings contained redundancies that increased the amount of storage space required for the database. Another important problem was the gaps between hourly recordings, because of the time which was needed to store the recordings. All these challenges were the motivation behind the undertaking of this project.

### III. THEORETICAL FRAMEWORK

#### A. Instantaneous Frequency Calculation

As discussed in papers such as [11], complex trace analysis (a method using the Hilbert transform of a signal) is used to calculate its instantaneous frequency. This method considers the real signal  $f(t)$  whose frequency is to be computed as the real part of a complex trace

$$F(t) = f(t) + jf^*(t) \quad (1)$$

Where  $f^*(t)$  is the quadrature trace of  $f(t)$ , which can be obtained through the Hilbert transform of the signal  $f(t)$ .

We can express  $f(t)$  as a sinusoidal function with a time dependent amplitude  $A(t)$  and a time dependent phase  $\theta(t)$ . Thus, the complex trace can be represented as follows:

$$f(t) = A(t) \cos\theta(t) \quad (2)$$

$$f^*(t) = A(t) \sin\theta(t) \quad (3)$$

$$F(t) = A(t) (\cos\theta(t) + j \sin\theta(t)) = A(t) e^{j\theta(t)} \quad (4)$$

Equation (4) demonstrates that if the complex trace and amplitude are known, the instantaneous phase of  $f(t)$  is

$$\theta(t) = \arctan(f^*(t)/f(t)) \quad (5)$$

The instantaneous frequency is determined by the derivative of the instantaneous phase function  $\theta(t)$ .

In the case of the current application, the utility voltage signal is a discrete time signal. We thus must calculate the samples of the Hilbert transform of  $f(t)$  (which we call  $f^*[n]$ ) and then, the discrete complex trace can be written as:

$$F[n] = A[n] e^{j\theta[n]} \quad (6)$$

As a result, the instantaneous analogue frequency  $\phi[n]$  can be calculated by taking the first difference of the instantaneous phase time series. The result is the instantaneous frequency of the original signal, as a fraction of its sampling frequency,  $F_s$ .

$$\phi[n] = F_s (\theta[n] - \theta[n-1]) / 2\pi \quad (7)$$

With a moving average low pass filter, we can define the signal's average frequency, in a certain small timeframe.

### IV. THE RASPBERRY PI BASED APPLICATION

#### A. Hardware

When creating the new application - which runs on a Raspberry Pi 3 Model B - we must consider the special hardware restrictions of the device.

The first problem is that the RPi does not come with a pre-installed audio card. A simple solution that was used for this project is a USB Sound Card Adaptor. In general, the RPi can be connected to other physical hardware to increase its functionality [7]. Also, it seems that the available hardware does not allow for sample rate other than 48 kHz or 44.1 kHz. Thus, we need to sample at one of these frequencies and, after capturing the signal, downsample it.

Just as in the original application, a modified power supply unit can be used to feed the utility signal into the USB Sound Card microphone port. The whole sensing apparatus can be seen in Fig. 2.

#### B. The Python Script

This application was developed using the Python 3 programming language, along with the libraries PyAudio [8], NumPy [9] and SciPy [10].

To facilitate a seamless transition between different recording segments, the app utilizes the Threading module, from Python's standard library. More specifically, the ENF extraction and the archiving of the recordings are completed by threads other than the main thread. Thus, the application is always recording the utility signal, without having to interrupt this process to extract its frequency or store it in file.

#### C. Recording the utility voltage with PyAudio

Recording the utility voltage signal takes place in the main thread of the application, using the PyAudio module. The utility voltage is sampled at 48 kHz from the USB sound card's microphone port, in segments lasting 300 s.

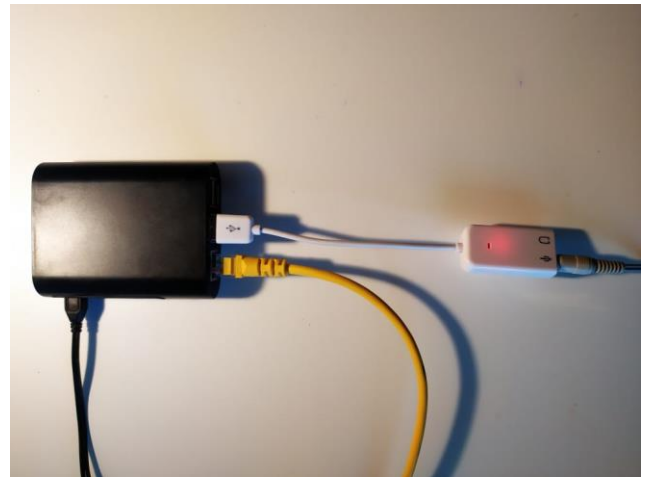


Fig. 2. Photo of the sensing device (Raspberry Pi, Sound card, connected to the internet via Ethernet).

Each recording segment can be characterized by its starting time. The samples are represented by short signed integers (16-bit integers). Once the 300 seconds elapse, the signal is processed by a different thread, before the ENF extraction.

#### D. Signal Processing

The processing of the utility voltage signal aims to downsample the signal, so that the number of samples in the ENF extraction stage is reduced, while also filtering off any noise in the data.

The filtering and downsampling happens in two stages. Firstly, a lowpass sinc filter windowed using a Blackman window (with a cutoff frequency of 4 kHz and a transition band of 2 kHz) is applied to the original discrete-time signal. The filtered signal is then normalized and quantized, before being downsampled to an intermediate frequency of 6 kHz.

After the first downsampling, the signal is filtered again through a similar windowed sinc filter of cut-off frequency of 500 Hz and a transition zone of 200 Hz. Finally, the filtered signal is normalized, quantized and downsampled to a final frequency of 1000 Hz.

The downsampling compresses the signal, while preserving the ENF (close to 50 or 60 Hz) with some of its major harmonics. The lowpass filters remove any higher harmonics, as well as noise introduced to the signal. Without denoising, the conversion of the signal to a lower sampling rate would introduce significant temporal aliasing.

It is important to note that the usage of two filters and an intermediate sampling frequency reduces the amount of time required for the processing, compared to the case of a single filter, with a cut-off frequency of 500 Hz and a transition zone of 200 Hz. This is because the two filters used here are comprised of less terms (97 for the first filter and 121 for the second one), compared to a single filter that achieves the same small transition band of 200 Hz (961 terms in this case).

Considering that the convolve function of NumPy has a time complexity of  $O(NM)$ , in the case of  $\log(N) < M$  - where  $N$  is the number of terms of the signal and  $M$  is the number of terms of the filter - it becomes evident that the two-filter method with an intermediate frequency has a lower time complexity. In fact, the method utilized in this application is

timed faster than using one filter and one direct sampling to 1 kHz.

#### E. Frequency Calculation

Instead of storing the audio file of the utility voltage signal itself, we can extract useful features from it, to reduce the dimensionality of our data. The most essential of these features, in this case, is the ENF as a function of time. We can calculate the recorded signal's instantaneous frequency using the complex trace analysis method, as discussed above. In this case, we want to sample the ENF at 10 Hz, so we calculate the average of 100 instantaneous frequency samples to smoothen the estimated frequency and avoid major spikes.

A potential problem with this method is the big spikes in the calculated frequency that occur especially at the beginning of recording segments. We can mitigate this issue by discarding the first 20 samples of the calculated frequency (before smoothing), as these samples are empirically found to contain the biggest spikes.

#### F. Store and archive the ENF recordings

After the ENF extraction from the signal, the calculated frequencies are stored in a .csv file along with their corresponding timestamps. All recorded segments from the same date and hour are stored in the same file, which is named after the datetime of recording (i.e. the file 2020\_08\_01\_21.csv contains the ENF measurements taken on the 1st of August 2020, approximately between 9 pm and 10 pm local time). Each row contains a timestamp and its corresponding recorded frequency, separated by commas. The timestamp follows the format "yyyy/mm/dd HH:MM:SS.f". The instantaneous frequency is stored in Hz, with 6 significant figures.

Finally, when the application detects that there are available .csv files with recordings from dates other than the current date, the files are grouped by date and archived, using the zip format. Each archived zip file contains all the recordings of a single day, and it is named according to the "yyyy/mm/dd.zip" format. Each zip file is approximately 4 MB, showing a significant reduction in size, compared to the method of storing raw wav files of the recordings. The whole process is shown in Fig. 3.

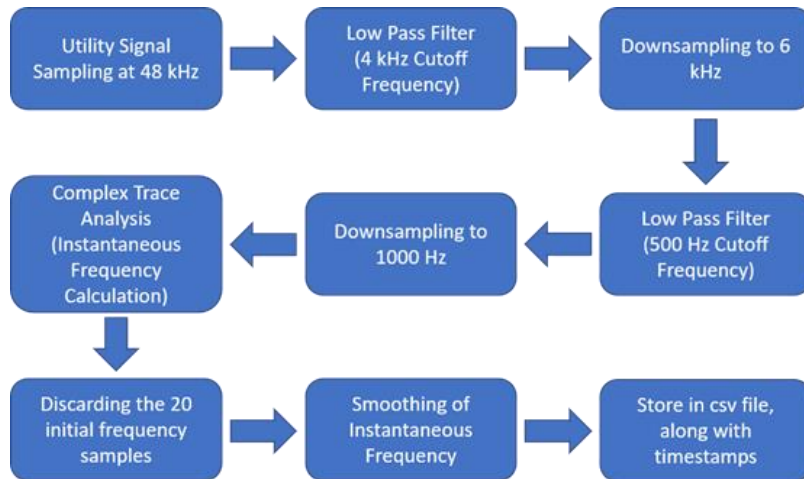


Fig. 3. Block diagram of the processing of each recording segment.

### G. Start up on boot

To ensure that the application is always running without the user having to log into the Raspberry Pi and run the Python script, we can use Cron [12] to schedule the task on reboot. Cron is a tool for scheduling tasks on Unix based systems. By scheduling the app to run on reboot, we can also make sure that the application starts running again even after the Raspberry Pi is unexpectedly shut down (e.g. because of a blackout).

### H. Using the application

To start the application, simply connect the USB sound card to one of the Raspberry Pi's USB ports, and afterwards, boot the device. Then plug the modified power supply to the microphone port of the audio card. The recording of the ENF will begin approximately 20 seconds after the application has started running, to make sure that the device has enough time to boot properly. To ensure that the device is synchronized, the user must connect it to the internet before the beginning of the recording. In the case that there is no internet access, the device will not be properly synchronized in time, and the recordings will be mislabeled.

## V. RESULTS

Using the Raspberry Pi application, we can analyze some of the resulting recordings. From Fig. 4, we can observe the sinusoidal form of the utility signal (after the two-step filtering and downsampling processes), and its corresponding spectrum. The signal has, as expected, a strong frequency component around 50 Hz (the typical ENF in the European Continental Grid) and around its harmonics.

To validate the accuracy of the recordings using the new application, we can compare the calculated ENF for a specific time interval, with the corresponding recordings of the original application. Figures 5 and 6 depict the aforementioned recordings, on the 28th of December, 2019, from both applications. At first glance, the two waveforms don't seem to match.

Nevertheless, if we consider the fact that the original application calculates the ENF in 1 second intervals, while the Raspberry Pi application provides a more fine-grained sampling of the frequency (every 0.1 seconds). If we apply some smoothing to the recorded frequencies (using a simple exponential low pass filter - Fig. 7), we can observe that the two recordings match, albeit with some significant time shift (Fig 8).

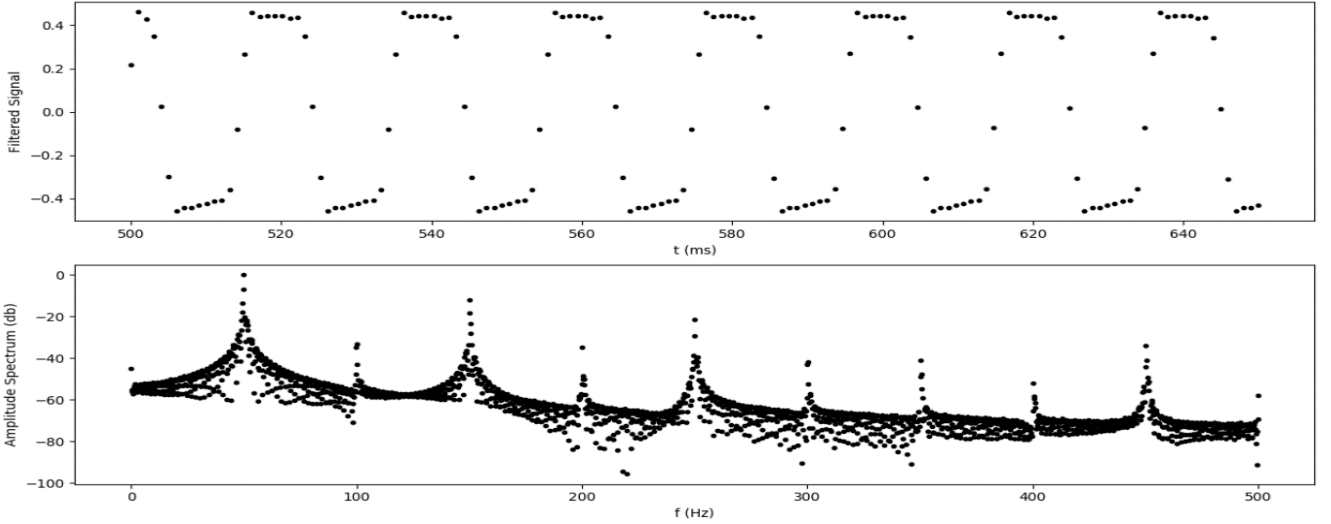


Fig. 4. The wave form (first plot) and the spectrum (second plot) of a recorded segment, after filtering and downsampling.

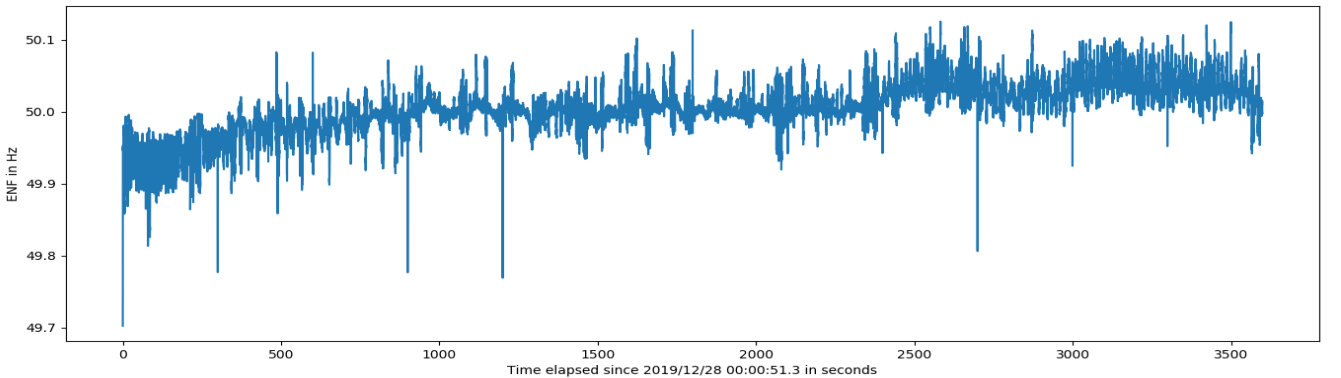


Fig. 5. Approximately an hour of ENF recordings on the 28th of December, 2019, using the Raspberry Pi application.

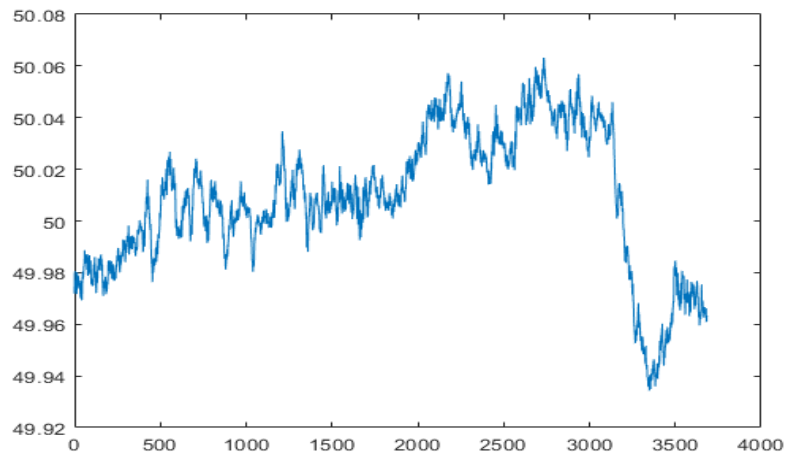


Fig. 6. The same ENF segment recorded using the original application.

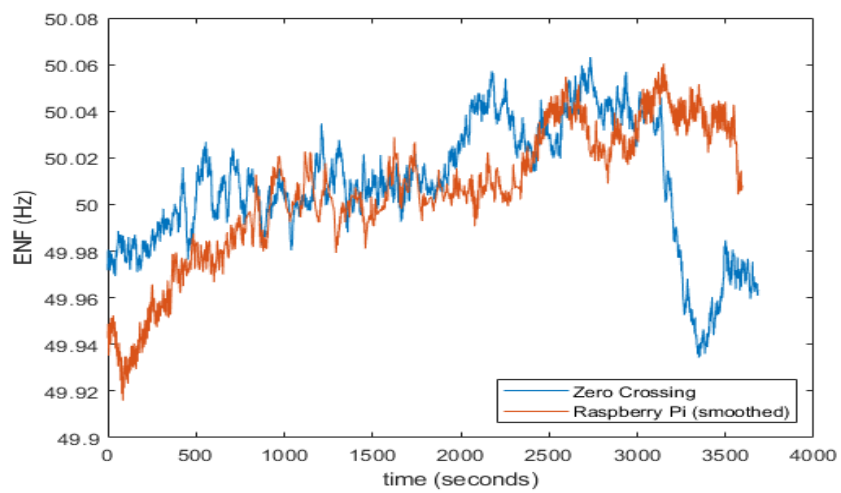


Fig. 7. Comparing the recordings made using the original application, to the ones recorded with the Raspberry Pi application (after smoothing).

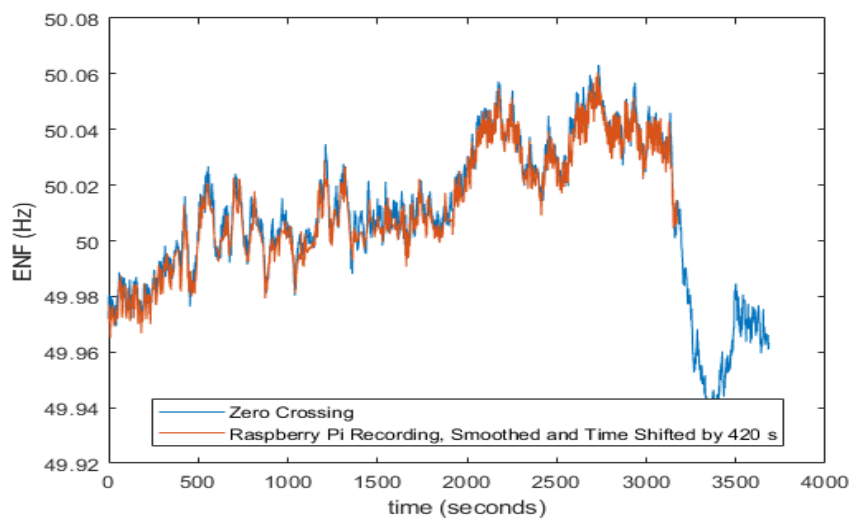


Fig. 8. Comparing the two recordings again, after a time shift of 420 seconds.



### A. The problem of accurate time keeping

One major problem with using the Raspberry Pi as a recording device for ENF extraction is the fact that it lacks a built-in real-time clock (RTC). When it is connected to the internet, the time is set over the network. If there is no network connection available when the device boots up, it will set the system clock to the last time it had before the last shutdown. As the ENF extraction process requires accurate timing, the lack of an RTC renders the recordings useless without a reliable internet connection.

### B. Final adjustments to the application

After noting the discrepancy between the old ENF recording application, some minor adjustments were made to the new application, to reduce the spikes in the recorded frequency. Firstly, the application was modified to calculate the ENF every second, instead of 10 times per second. This modification reduces the size of the resulting file to around 390KB per day. Also, the modified application applies a simple exponential smoothing filter to the time series of produced frequencies, with a smoothing factor of 0.1. Furthermore, a script to upload the resulting files to the cloud was created and integrated to the application. The recordings resulting after the adjustments mentioned above are compared with the non-adjusted recordings in Fig. 9.

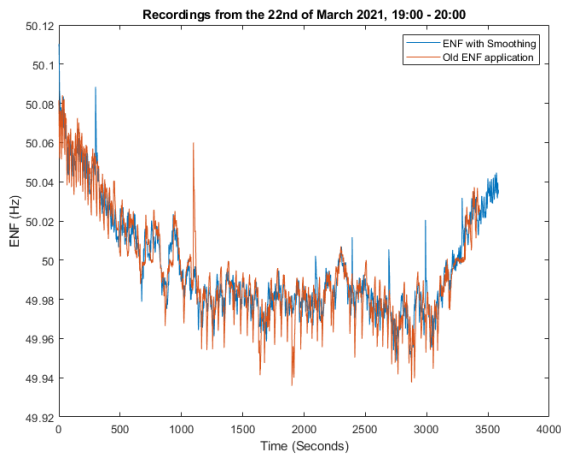


Fig. 9. Comparing the recordings of the new application, before and after the adjustments presented in Section V.B.

## VI. CONCLUSION

As demonstrated in this report, the method presented for extracting and storing the Electrical Network Frequency presents numerous advantages compared to the method developed in [5] and [6]. The application is designed to operate on a device which is better suited to the computational demands of the task. It also directly provides the ENF recordings, and thus helps reduce the storage space required years of archived recordings.

## ACKNOWLEDGMENT

Appreciation is especially expressed to Dr. Athanasios Skodras, for his help, guidance and providing of technical expertise throughout this whole project.

The work of Andreas Triantafyllopoulos, Anastasios Foliadis, George Roustas, Ioannis Krillis, Fanouria Athanasiou, and Maria Papaioannou for the IEEE Signal Processing Cup 2016 is also acknowledged, as their original application was the bases for the undertaking of this endeavor.

## REFERENCES

- [1] Grigoras, Catalin. "Digital audio recording analysis—the electric network frequency criterion." *International Journal of Speech Language and the Law* 12.1 (2005): 63-76.
- [2] Huijbregtse, Maarten, and Zeno Geradts. "Using the ENF criterion for determining the time of recording of short digital audio recordings." *International Workshop on Computational Forensics*. Springer, Berlin, Heidelberg, 2009.
- [3] Raspberry Pi Downloads, The Raspberry Pi Foundation (<https://www.raspberrypi.org/downloads/>)
- [4] Raspberry Pi Documentation: Usage, The Raspberry Pi Foundation (<https://www.raspberrypi.org/documentation/usage/>)
- [5] Andreas Triantafyllopoulos, Anastasios Foliadis, George Roustas, Ioannis Krillis, Fanouria Athanasiou, Maria Papaioannou. "Exploring Power Signatures for Location Forensics of Media Recordings." *IEEE Signal Processing Cup* 2016.
- [6] Andreas Triantafyllopoulos, Ioannis Krillis, Anastasios Foliadis, Athanassios Skodras. "A Hilbert-Based Approach to the ENF Extraction Problem." *IEICE Proceedings Series* 24.A3-3 (2016).
- [7] Raspberry Pi Blog, Introducing Raspberry Pi HATs, The Raspberry Pi Foundation <https://www.raspberrypi.org/blog/introducing-raspberry-pi-hats/>
- [8] Install PyAudio, Python Package Index <https://pypi.org/project/PyAudio/>
- [9] <https://numpy.org/>
- [10] <https://www.scipy.org/>
- [11] Taner, M. Turhan, Fulton Koehler, and R. E. Sheriff. "Complex seismic trace analysis." *Geophysics* 44.6 (1979): 1041-1063.
- [12] <https://www.raspberrypi.org/documentation/linux/usage/cron.md>

## APPENDIX A - PYTHON SOURCE CODE (APP\_NEW.PY)

```
#!/usr/bin/env

# Imports
from sys import byteorder
from array import array
from time import *
from datetime import datetime
import threading
import shutil
from glob import glob
import os

import pyaudio
import csv
import numpy as np
from scipy import signal

import upload_zip

# Delay the app for 20 seconds so that the device can boot properly
sleep(20)

# Global variables
FORMAT = pyaudio.paInt16
RATE = 48000 # RaspberryPi can only record at rates of 48 and 44.1 kHz
CHUNKS_PER_SEC = 2 # Data chunks per second
CHUNK_SIZE = RATE // CHUNKS_PER_SEC # Determining chunk size, in samples
RECLENGTH_IN_SECS = 300 # Seconds of recording before processing and writing on file
RECLENGTH = RECLENGTH_IN_SECS * CHUNKS_PER_SEC # Length of recordings in chunks
DEVICE = 2 # The audio recording device that the data stream comes from
INTER_RATE = 6000 # The sampling rate of first downsampling
FINAL_RATE = 1000 # The final sampling rate of the downsampled recording
PARENT_DIR = "/home/pi/Desktop/CSV_Recording/" # Change it if parent directory of project changes

log = open(PARENT_DIR + "log.txt", "a") # log.txt is the log file for the app
log.write("Running:" + strftime("%Y_%m_%d_%H_%M_%S") + "\n") # Reports on the app running
log.close()

class RecordThreadProcess(threading.Thread):
    """ This class consists of the thread that processes a recording of RECLENGTH chunks
    (downsampling, noise filtering, normalization), computes the frequency every 100 ms
    and stores it in a csv file, named for the date and hour that the recording takes place.
    This class inherits from the Python native threading.Thread class.
    """
    def __init__(self, data, filename, start_time, lpf1, lpf2):
        """ The constructor of this class inherits all arguments from the threading.Thread class
        and has additional arguments:
        data: (array) the samples recorded in the last recording
        filename: (str) the name of the file the data is to be stored in
        start_time: (float) the time the recording began
        lpf1: (numpy array) the first sinc low pass filter
        lpf2: (numpy array) the second sinc low pass filter.
        """
        super(RecordThreadProcess, self).__init__()
        self.data = data
        self.filename = filename
        self.start_time = start_time
        self.lpf1 = lpf1
        self.lpf2 = lpf2
        self.freq = None # The frequency calculated every 100 ms

    def run(self):

        self.data = self.noise_filter(self.data, self.lpf1) # First noise filtering (low pass sinc filter)
        self.data = self.normalize(self.data) # Normalizing data after filtering
        self.data = self.downsample(self.data, RATE, INTER_RATE) # Downsampling to the INTER_RATE frequency

        self.data = self.noise_filter(self.data, self.lpf2) # Second noise filtering
        self.data = self.normalize(self.data) # Normalizing data after filtering
        self.data = self.downsample(self.data, INTER_RATE, FINAL_RATE) # Downsampling to the FINAL_RATE frequency

        ''' The self.freq variable is a dictionary of the frequency of the signal of the recording
        every 1000 ms along with the ordinal number of the recording. '''
        self.freq = self.inst_freq(self.data)

        # Corrects the time stamps of the recording.
        self.time_stamps()

        # Opens a filename.csv file in the ENF_Recordings directory
        csv_file = open(PARENT_DIR + "ENF_Recordings/" + self.filename, "a", newline="")
```

```

# Storing timestamps and frequencies in the file
write_ = csv.writer(csv_file)
for stamp in sorted(self.freq):
    write_.writerow([stamp, str(round(self.freq[stamp], 4))])

csv_file.close()

return

def downsample(self, r, rate1, rate2):
    """Downsamples the input data stream to a new sampling
    frequency rate2. It returns the downsampled data stream downs_r.
    Inputs:
    r: (array) input data stream
    rate1: (int) sampling frequency of r
    rate2: (int) sampling frequency of output
    """
    length = len(r)
    ratio = int(rate1/rate2) # Downsampling ratio
    downs_size = np.floor(length/ratio) # Size of output array
    downs_r = array('h', [r[i*ratio] for i in range(int(downs_size))])
    return downs_r

def normalize(self, snd_data):
    """ Normalizes the input data snd_data and returns
    an array r"""
    MAXIMUM = 16384
    '''The highest value in the input data stream has to have
    an absolute value of 16384, and all the other samples
    values should be scaled accordingly'''
    times = float(MAXIMUM)/max(abs(i) for i in snd_data)

    r = array('h')
    for i in snd_data:
        r.append(int(i*times))
    return r

def noise_filter(self, r, lpf):
    """ Filters the input array r with the low pass
    filter lpf. The function returns the array r_filtered"""
    r_filtered = np.convolve(lpf, r, mode='same')
    return r_filtered

def inst_freq(self, input_signal):
    """ This function calculates the instantaneous frequency
    of the input signal by calculating the instantaneous phase of its complex trace
    and then taking the first difference of this array, scaled by a factor
    of fs/2pi. The resulting values are then averaged to smooth out the
    big spikes in frequency that the method produces. The function returns
    the dictionary of frequencies smoothed"""
    input_signal = np.array(input_signal)
    # Taking the Hilbert transform of the input signal
    analytic_signal = signal.hilbert(input_signal)
    # Finding the phase of the Hilbert transform
    instantaneous_phase = np.unwrap(np.angle(analytic_signal))
    # Taking the first difference of the instantaneous phase array
    instantaneous_frequency = (np.diff(instantaneous_phase) / (2.0 * np.pi) * 1000)
    # Discarding the first 20 frequency samples, as they tend to show big spikes
    instantaneous_frequency = instantaneous_frequency[20:]
    # Creating the dict of smoothed frequencies
    smoothed = {}
    mean = 0
    for i in range(len(instantaneous_frequency)):
        # We sample the frequency at 1 Hz
        freq_rate = 1
        ratio = int(FINAL_RATE/freq_rate)
        if i % ratio == 0 and i > 0:
            mean = mean / ratio
            if i == ratio:
                prev_mean = mean
                mean = 0.1 * mean + 0.9 * prev_mean
                smoothed[int(i/ratio)] = mean
                prev_mean = mean
                mean = 0
            mean += instantaneous_frequency[i]

    return smoothed

def time_stamps(self):
    """ This function turns the keys in the dict of frequencies into
    proper time stamps."""
    temp = {}
    for sec in self.freq:
        if type(sec) == int:
            stamp = sec + self.start_time # Adds to the sec value the start time of the recording.

```



```

        stamp = datetime.fromtimestamp(stamp) # Creates a timestamp out of the number.
        stamp = stamp.strftime("%Y/%m/%d %H:%M:%S") # Formats the timestamp.
        temp[stamp] = self.freq[sec]
self.freq = temp

```

```

class ArchiveThreadProcess(threading.Thread):
    """ This class consists of the thread that archives all the files from a previous date.
        The thread first checks whether archiving is needed and, if this is the case, it proceeds
        to create a zip file that contains all the recordings that come from the same day.
        This class inherits from the Python native threading.Thread class.
    """

    def __init__(self):
        """ The constructor of this class inherits all arguments from the threading.Thread class
            with no extra arguments. """
        super(ArchiveThreadProcess, self).__init__()

    def run(self):
        """ Begins archiving if it is needed. """
        if self.archive_needed():
            self.archive()
        return

    def archive_needed(self):
        """ This function evaluates whether or not there is the need to archive recordings.
            This is the case when there are files in the ENF_Recordings director, from at least 2
            different days. """
        filelist = glob(PARENT_DIR + 'ENF_Recordings/*.csv')
        filelist.sort() # The oldest file becomes the first on the list
        if filelist != []:
            first_file = filelist[0]
            first_file = first_file.split('/')
            first_date = first_file[-1][0:10] # The date of the newest file
        else:
            return False

        for file in filelist:
            file = file.split('/')
            date = file[-1][0:10] # The date of the newest file
            if date != first_date:
                return True # Return True only if the two days are different
        return False

    def archive(self):
        """ This function creates a zip archive of all files in ENF_Recordings that come from
            the oldest date of all the files that are in the folder. It first creates a list of
            these files and then moves them to the temp directory, before archiving them and then
            deleting them. It finally uploads the file to Google Drive. """

        infiles = []
        filelist = glob(PARENT_DIR + 'ENF_Recordings/*.csv')
        filelist.sort()
        if filelist != []:
            first_file = filelist[0]
            first_file = first_file.split('/')
            first_date = first_file[-1][0:10] # The oldest date of all files in ENF_Recordings

        for i in range(len(filelist)):
            file = filelist[i]
            file = file.split('/')
            date = file[-1][0:10]
            if date == first_date:
                infiles.append(file[-1]) # All the files with the same date as first_date are added to the infiles list

        for file in infiles:
            shutil.move(PARENT_DIR + "ENF_Recordings/" + file,
                        PARENT_DIR + "temp/" + file) # All files in the infiles list are moved to temp

        shutil.make_archive(PARENT_DIR + 'ENF_Archives/' + first_date,
                            'zip', PARENT_DIR + 'temp/') # The files in temp are added to an archive
        # The name of the archive is the date of the recordings that it contains

        # The files in the temp directory are deleted
        filelist = glob(PARENT_DIR + 'temp/*.csv')
        for f in filelist:
            os.remove(f)

        log = open(PARENT_DIR + "log.txt", "a") # log.txt is the log file for the app
        log.write("Running:" + strftime("%Y_%m_%d_%H_%M_%S") + "\n") # Reports on the app running
        log.close()
        upload_zip.upload_zip(first_date + ".zip")

def make_lpf(cut, trans, rate):
    """Makes a low pass filter with a cutoff frequency cut, a transition band

```

```

    of trans hz that corresponds to a sampling rate rate. The filter is a sinc
    filter combined with a Blackman window. The function returns a numpy
    array containing the filter terms."""

fc = cut/rate # Cutoff frequency as a fraction of the sampling rate (in (0, 0.5)).
b = trans/rate # Transition band, as a fraction of the sampling rate (in (0, 0.5)).
# The number of terms of the filter
N = int(np.ceil((4 / b)))

if not N % 2:
    N += 1 # Make sure that N is odd.

n = np.arange(N)

# Compute sinc filter.
h = np.sinc(2 * fc * (n - (N - 1) / 2))
# Compute Blackman window.
w = 0.42 - 0.5 * np.cos(2 * np.pi * n / (N - 1)) + \
    0.08 * np.cos(4 * np.pi * n / (N - 1))

# Multiply sinc filter with window.
h = h * w

# Normalize to get unity gain.
filter = h / np.sum(h)

return filter

def record():
    """ Records a stream of data from the audio device
    consisting of RECLENGTH chunks. Each chunk consists of
    CHUNK_SIZE bytes, in a little endian, signed,
    short int representation. The function also records the
    start time of the recording and the file in which the data
    should be recorded in. The output is data (an array of bytes),
    filename and start_time."""

    # Every file is named according its date and hour
    filename = strftime("%Y_%m_%d_%H") + '.csv'
    start_time = time() # This is the start time of the recording

    # Opening a pyaudio stream
    p = pyaudio.PyAudio()
    stream = p.open(format=FORMAT, channels=1, rate=RATE,
                    input=True, output=True,
                    frames_per_buffer=CHUNK_SIZE,
                    input_device_index=DEVICE)

    # Initializing array optimized for signed short ints
    data = array('h')

    num_chunks = 0
    while 1:
        # Reading data from stream, chunk by chunk
        # little endian, signed short
        snd_data = array('h', stream.read(CHUNK_SIZE))
        if byteorder == 'big':
            snd_data.byteswap()

        # Adding data to array
        data.extend(snd_data)
        num_chunks += 1
        # The loop exits when RECLENGTH chunks are read
        if num_chunks >= RECLENGTH:
            break

    stream.stop_stream()
    stream.close()
    p.terminate()

    return data, filename, start_time

if __name__ == '__main__':

    lpf1 = make_lpf(4000, 2000, RATE) # Low pass filter for first downsampling
    lpf2 = make_lpf(500, 200, INTER_RATE) # Lpf for second downsampling

    while 1:
        # Main loop of the app
        # First archive files if necessary, in a separate thread from recording
        arch_thread = ArchiveThreadProcess()
        arch_thread.start()

```

```

# Data is recorded with the record() function
data, filename, start_time = record()

# The data is processed and the frequencies recorded are added to the filename file.
rec_thread = RecordThreadProcess(data, filename, start_time, lpf1, lpf2)
rec_thread.start()

```

## APPENDIX B. - PYTHON SOURCE CODE (UPLOAD\_ZIP.PY)

```

#!/usr/bin/env

# Imports
from time import *
from pydrive.drive import GoogleDrive
from pydrive.auth import GoogleAuth
import os

def upload_zip(file):
    PARENT = "/home/pi/Desktop/CSV_Recording/"
    path = PARENT + "ENF_Archives/"

    gauth = GoogleAuth()
    gauth.LoadCredentialsFile(PARENT + "mycreds.txt")
    if gauth.credentials is None:
        gauth.LocalWebserverAuth()
    elif gauth.access_token_expired:
        gauth.Refresh()
    else:
        gauth.Authorize()
    gauth.SaveCredentialsFile(PARENT + "mycreds.txt")
    drive = GoogleDrive(gauth)

    fid = '1J_S8z7wer3VrCaBk5KrhU0FoLMDaMohg'

    f = drive.CreateFile({"parents":[{"kind":"drive#fileLink", "id": fid}], "title":file})
    f.SetContentFile(os.path.join(path, file))
    f.Upload()
    f = None
    log = open(PARENT + "log.txt", "a") # log.txt is the log file for the app
    log.write("Uploaded " + file + " " + strftime("%Y_%m_%d_%H_%M_%S") + "\n") # Reports on the app running
    log.close()

    return

```