# itsFM

*Release 0.0.1*

Thejasvi Beleyur

Nov 05, 2020

## EXAMPLES!

CHAPTER

ONE

# INTRODUCTION

The *itsfm* package identifies regions of sound with and without frequency modulation, and allows custom measurements to be made on them. It's all in the name. Each of the task behind the identification, tracking and segmenting of a sound can be done independently.

The sounds could be bird, bat, whale, artifical sounds - it should hopefully work, however be aware that this is an alpha version package at the moment.

The basic workflow involves the tracking of a sounds frequency over time, and then calculating the rate of frequency modulation (FM), which is then used to decide which parts of a sound are frequency modulated, and which are not. Here are some examples to show the capabilities of the package.

The broad idea of this package is to achieve a loose coupling between the I,T, S in the package name. *itsfm* can do all or one of the below.

- I : Identify sounds by frequency modulation. An input audio can have multiple sounds in it, separated by silence or fainter regions.
- T : Track the sound's frequency over time. The PWVD method allows tracking a sound's frequency with high temporal resolution.
- S : Segment according to the frequency modulation. Calculates the local rate of frequency modulation over a sound and classifies parts of it as frequency modulated (FM) or constant frequency (CF)

*Warning : The docs are constantly under construction, and is likely to change fairly regularly like the stairs in Hogwarts. Do not be surprised by dramatic changes, but do come back regularly to see improvements!*

CHAPTER

TWO

# LET'S CUT TO THE CHASE : SOME EXAMPLES *NOW*

## 2.1 Basic Examples

This is a set of relatively straightforward examples. Start with the bat call example! The bat example is the only one with all the plots already rendered due to RAM limitations - sorry about that! You can of course always download all the examples and run them as individual .py or Jupyter notebooks!

All of the frequency tracking in *itsfm* is based on generating what is called the Pseudo Wigner-Ville Distribution (PWVD). To get a *very* quick idea of how it compares to a spectrogram checkout the 'Segmenting with the PWVD method' example.

### 2.1.1 Bat call example

The <INSERTNEWNAME> package has many example recordings of bat calls thanks to the generous contributions of bioacousticians around the world:

```
import matplotlib.pyplot as plt
import numpy as np
import itsfm
from itsfm.run_example_analysis import contributors
print(contributors)
```

Out:

```
Cannot import SoundFile!!
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/itsfm-0.0.1-py3.7.egg/itsfm/data/__init__.py:17: UserWarning:

 The package soundfile could not be imported properly. Check your installation.Using␣
↪the scipy.io package for now.
  warnings.warn(msg1+msg2)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/itsfm-0.0.1-py3.7.egg/itsfm/data/__init__.py:66: WavFileWarning: Chunk␣
↪(non-data) not understood, skipping it.
  fs_original, audio = wav.read(each)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/itsfm-0.0.1-py3.7.egg/itsfm
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/itsfm-0.0.1-py3.7.egg/itsfm/data_contributors.csv
0          Aditya Krishna
1              Aiqing Lin
2    Klaus-Gerhard Heller
```

(continues on next page)

(continued from previous page)

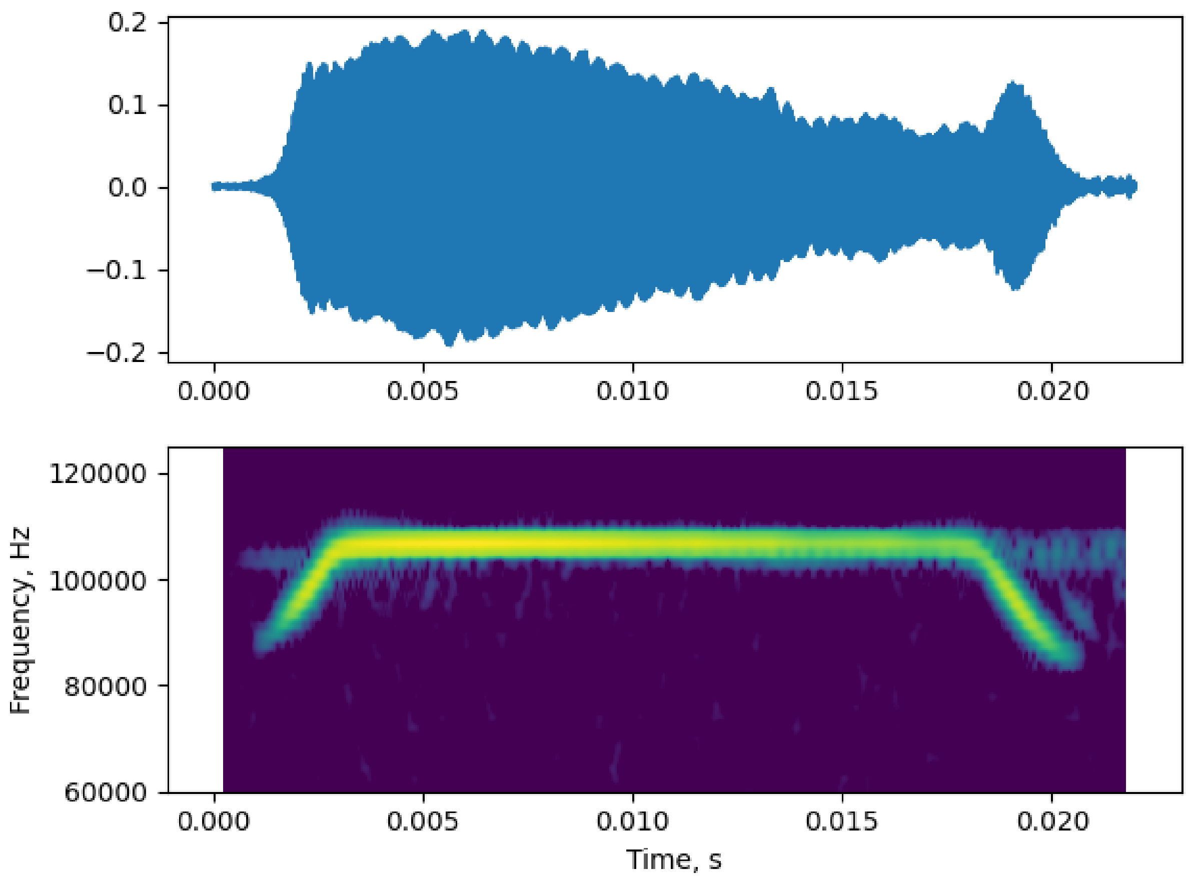
```
3               Neetash MR
4          Laura Stidsholt
Name: people, dtype: object
```

```
from itsfm.data import example_calls, all_wav_files
```

### Separating the constant frequency (CF) and frequency-modulated parts of a call

Here, let’s take an example *R. mehelyi/euryale(?)* call recording. These bats emit what are called ‘CF-FM’ calls. This is what it looks like.

```
bat_rec = list(map( lambda X: '2018-08-17_34_134' in X, all_wav_files))
index = bat_rec.index(True)
audio, fs = example_calls[index] # load the relevant example audio

w,s = itsfm.visualise_sound(audio,fs, fft_size=128)
# set the ylim of the spectrogram narrow to check out the call in more detail
s.set_ylim(60000, 125000)
```



Out:

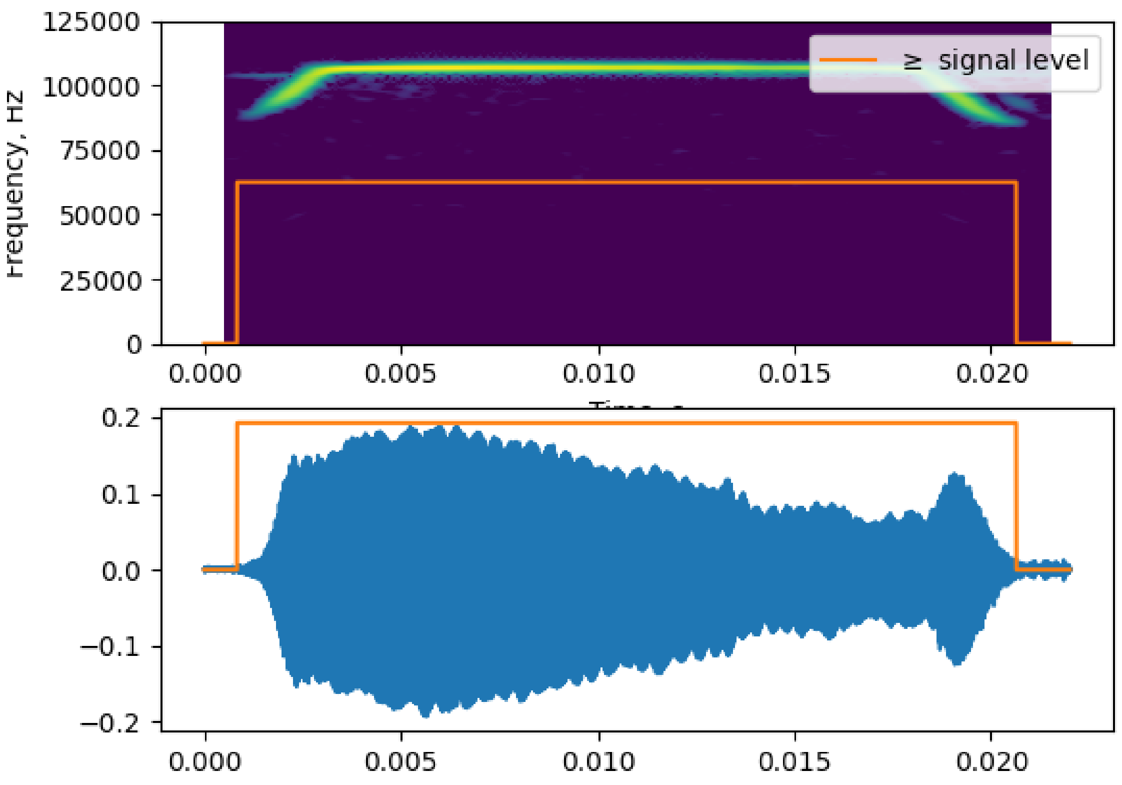
```
(60000.0, 125000.0)
```

Now, let's segment and get some basic measurements from this call. Ignore the actual parameter settings for now. We'll ease into it later !

```
non_default_parameters = {
                            'segment_method':'pwvd',
                            'signal_level':-26, # dBrms re 1
                            'fmrate_threshold':2.0, # kHz/ms
                            'max_acc':2.0, # kHz/ms^2
                            'window_size':int(fs*0.0015) # number of samples
                            }
outputs = itsfm.segment_and_measure_call(audio, fs,
                                           **non_default_parameters)

# load the results into a convenience class
# itsFMinspector parses the output and creates diagnostic plots
# and access to the underlying diagnostic data itself

output_inspect = itsfm.itsFMInspector(outputs, audio, fs)
```

Let's check that the threshold we chose actually matches the region of audio we're interested in

```
output_inspect.visualise_geq_signallevel()
```



Out:

```
(<AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

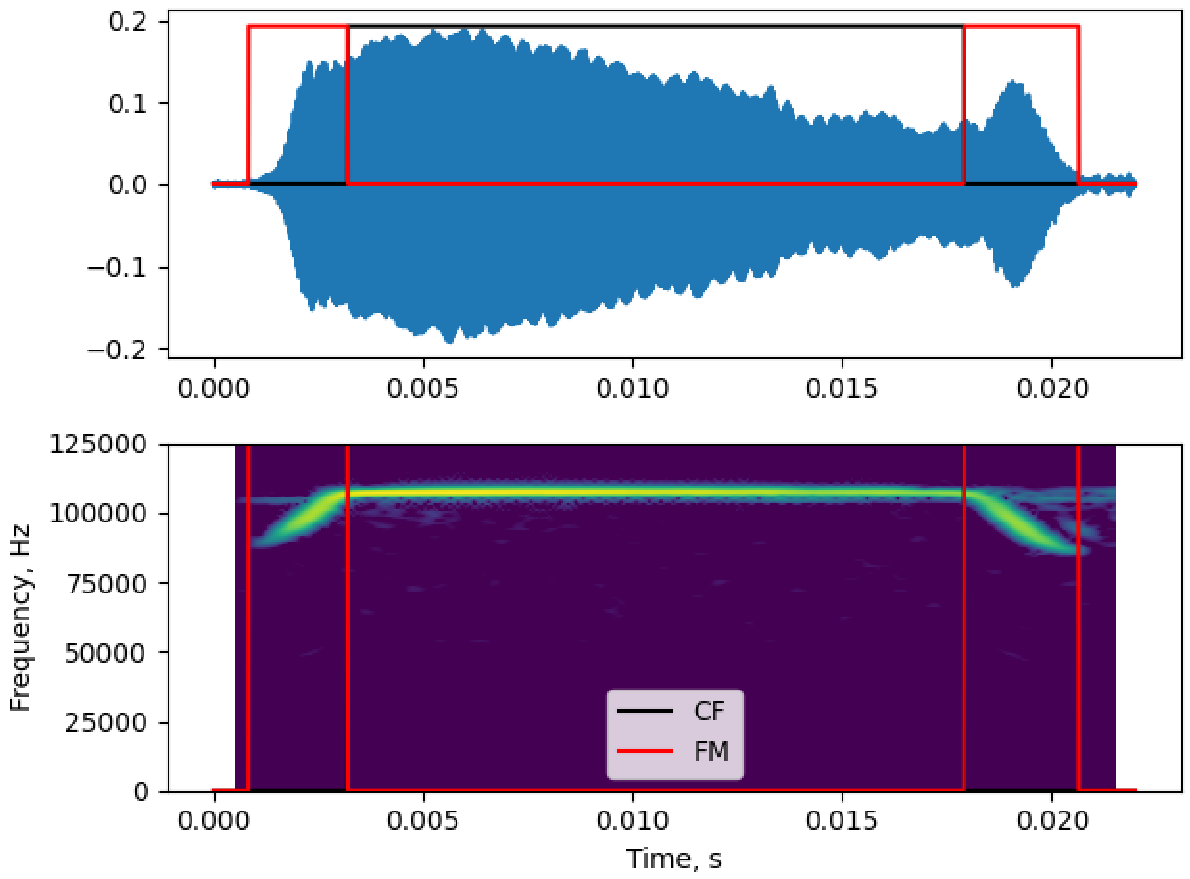
Let's take a look at how long the different parts of the call are.

```
output_inspect.measurements
```

## Verifying the CF-FM segmentations

Here, let's see where the calls are in time and how they match the spectrogram output

```
output_inspect.visualise_cffm_segmentation()
plt.tight_layout()
plt.savefig('pwvd_cffm_segmentation.png')
```

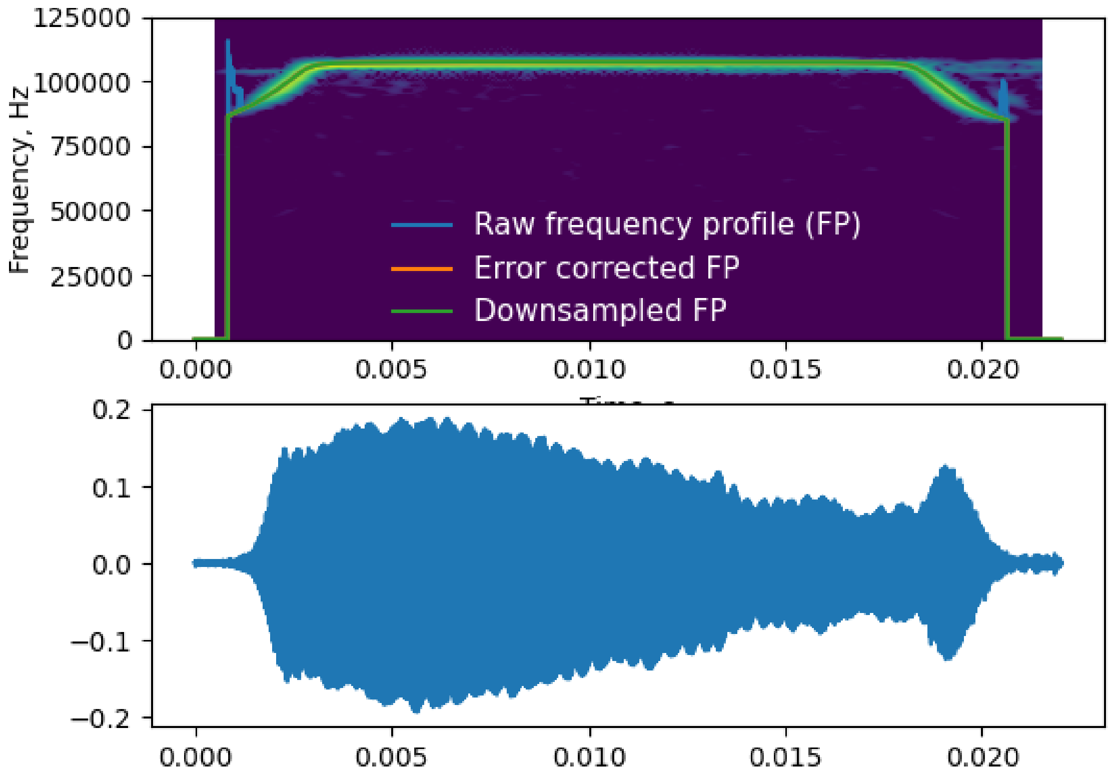


Even without understanding what's happening here, you can see the 'sloped' regions are within the red boxes, and the 'relatively even region is in the black box. These are the FM and CF parts of this call.

### The underlying frequency profile of a sound

The CF and FM parts of a call in the 'pwvd' method is based on actually tracking the instantaneous frequency of the call with high temporal resolution. With this profile, the rate of frequency change, or modulation can be calculated for each region. Using a threshold rate of the frequency modulation, call regions above and below it can be easily identified!

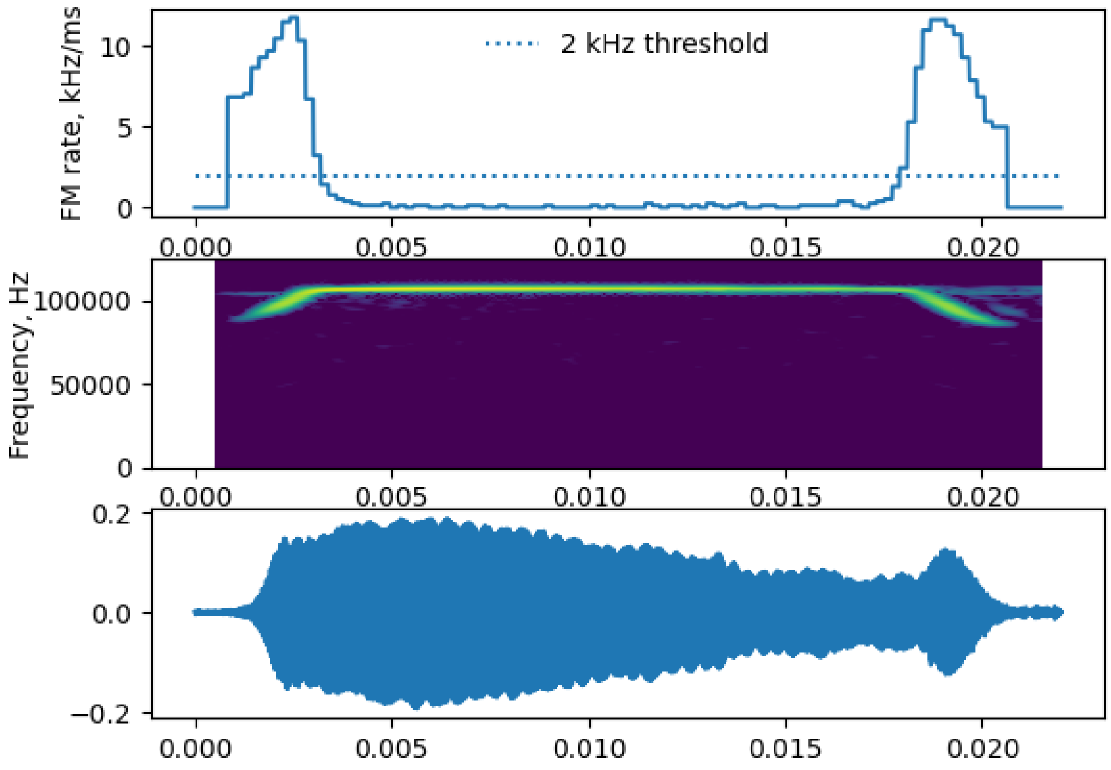
```
s,w = output_inspect.visualise_frequency_profiles()
s.legend_.remove()

handles, labels = s.get_legend_handles_labels()
labels_new = ['Raw frequency profile (FP)','Error corrected FP','Downsampled FP']
l = s.legend(handles, labels_new, loc=8, fontsize=11,
               borderaxespad=0., frameon=False, labelcolor='w')
s.set_ylabel('Frequency, Hz', labelpad=-1.5)
plt.savefig('pwvd_freqprofiles.png')
```



You can see from the plot above that the frequency profile of the sound shows a relatively constant frequency region of the call in middle and with frequency modulated regions in the middle.

### The underlying frequency modulation rate

```
fmrate_plot, spec, waveform = output_inspect.visualise_fmrate()
fmrate_plot.hlines(2,0,audio.size/fs, linestyle='dotted',label='2 kHz threshold')
fmrate_plot.legend(frameon=False)
plt.savefig('pwvd_fmrate_diagnostic.png')
```



### Performing measurements on the CF and FM parts of a call

We were just able to get some measurements on the Cf and FM parts of the call. What if we want *more* information, eg. the rms, and peak frequency of each CF and FM call part? This is where <insertname> has a bunch of inbuilt and customisable measurement functions.

```
inbuilt_measures = [itsfm.measure_peak_frequency,
                    itsfm.measure_rms]

non_default_parameters['measurements'] = inbuilt_measures
```

The `output` is a tuple with 3 objects in it related to the segmentation individual call parts and the measurements made on them. We're happy with the actual segmentation, and so won' be making anymore diagnostic plots, and won' need to call `itsFMInspector` anymore. We can unpack the outputs into its components and just view the measurements.

```
seg_out, call_parts, results_inbuilt = itsfm.segment_and_measure_call(audio, fs,
                                              **non_default_parameters
                                                  )
results_inbuilt
```

The results are output as a pandas DataFrame, which means they can be easily saved as a csv file if you were to run it in your system. Each row corresponds to one identified CF or FM region in an audio recording.

## Defining custom measurements

If the inbuilt measurement functions are not enough - then you may want to write your own. See the documentation for what a measurement function must look like by typing `help(itsfm.measurement_function)`. The 'peak_to_peak' function below calculates the difference between the highest negative and highest positive value. This effectively the maximum range of values that the signal takes.

```
def peak_to_peak(whole_audio, fs, segment, **kwargs):
    '''
    Calculates the range between the minimum and the maximum of the audio
    samples.
    '''
    relevant_audio = whole_audio[segment]
    peak2peak = np.max(relevant_audio) - np.min(relevant_audio)
    return {'peak2peak':peak2peak}

custom_measure_fn = [peak_to_peak]

# add the custom_measure list to the :code:`non_default_parameters` dictionary
#
non_default_parameters['measurements'] = custom_measure_fn



seg_out, call_parts, results_custom = itsfm.segment_and_measure_call(audio, fs,
                                                                  **non_default_
↪parameters
                                                  )
results_custom
```

Of course, needless to say, you can also mix and match inbuilt with custom defined measurement functions.

```
mixed_measures = [peak_to_peak, itsfm.measure_rms]
non_default_parameters['measurements'] = mixed_measures

seg_out, call_parts, results_mixed = itsfm.segment_and_measure_call(audio, fs,
                                              **non_default_parameters
                                                  )
results_mixed
```

**Total running time of the script:** ( 0 minutes 17.901 seconds)

### 2.1.2 Segmenting with the PWVD method

The 'PWVD' method stands for the Pseudo Wigner-Ville Distribution. It is a class of time-frequency representations that can be used to be gain very high spectro- temporal resolution of a sound [1,2], and can outdo the spectrogram in terms of how well it allows the tracking of frequency over time.

#### How does it work?

The PWVD is made by performing a local auto-correlation at each sample in the audio signal, with a window applied onto it later. The FFT of this windowed- auto correlation reveals the local spectro-temporal content. However, because of the fact that there are so many auto-correlations *and* FFT's involved in its construction - the PWVD can therefore take much more time to generate.

#### Note

The 'tftb' package [3] is used to generate the PWVD representation in this package. The website is also a great place to see more examples and great graphics of the PWVD and alternate time-frequency distributions!.

#### References

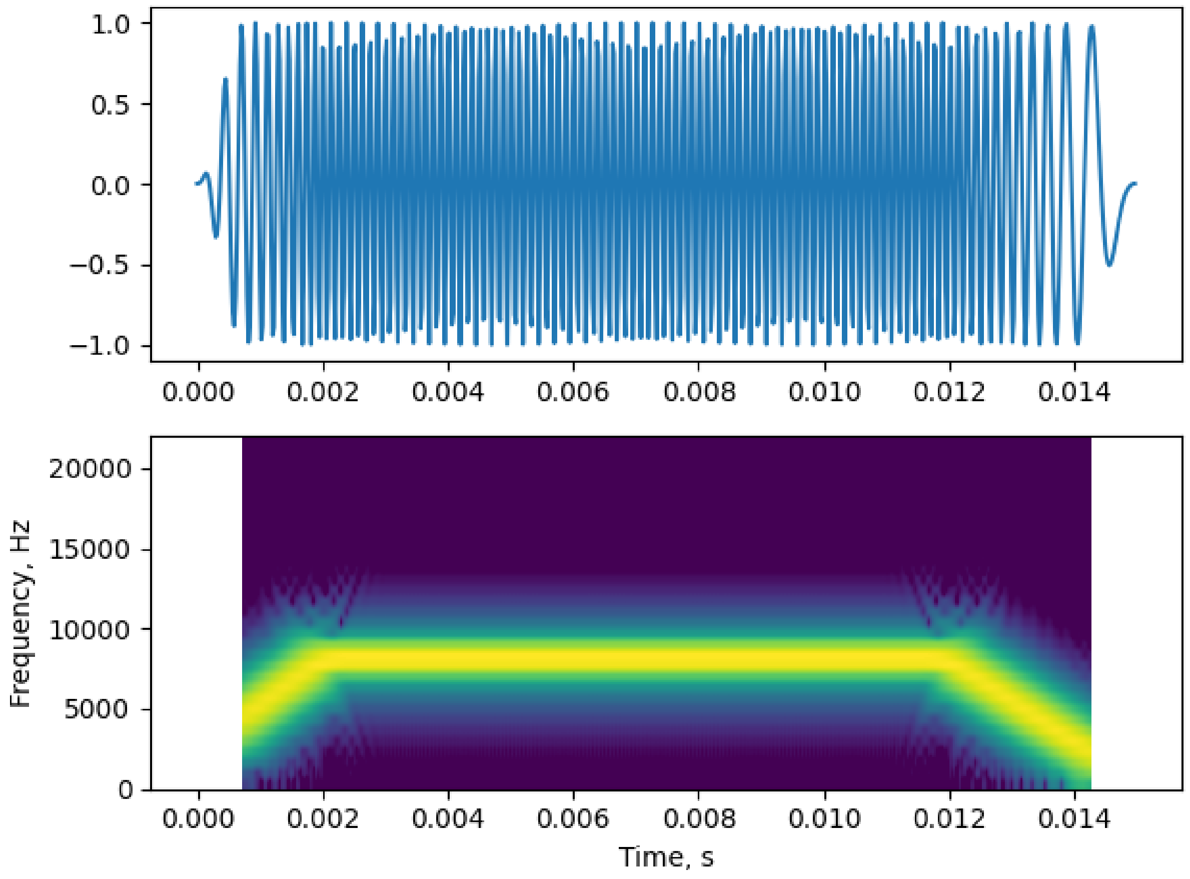
[1] Cohen, L. (1995). Time-frequency analysis (Vol. 778). Prentice hall.

[2] Boashash, B. (2015). Time-frequency signal analysis and processing: a comprehensive reference. Academic Press.

[3] Jaidev Deshpande, tftb 0.1.1, https://tftb.readthedocs.io/en/latest/auto_examples/index.html

Let's begin by making a synthetic CF-FM call which looks a lot like a horseshoe/leaf nosed bat's call

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as signal
import itsfm
from itsfm.frequency_tracking import generate_pwvd_frequency_profile
from itsfm.frequency_tracking import pwvd_transform
from itsfm.simulate_calls import make_cffm_call
from itsfm.segment import segment_call_into_cf_fm

fs = 44100
call_props = {'cf':(8000, 0.01),
                     'upfm':(2000,0.002),
                     'downfm':(100,0.003)}

cffm_call, freq_profile = make_cffm_call(call_props, fs)
cffm_call *= signal.tukey(cffm_call.size, 0.1)

w,s = itsfm.visualise_sound(cffm_call, fs, fft_size=64)
```

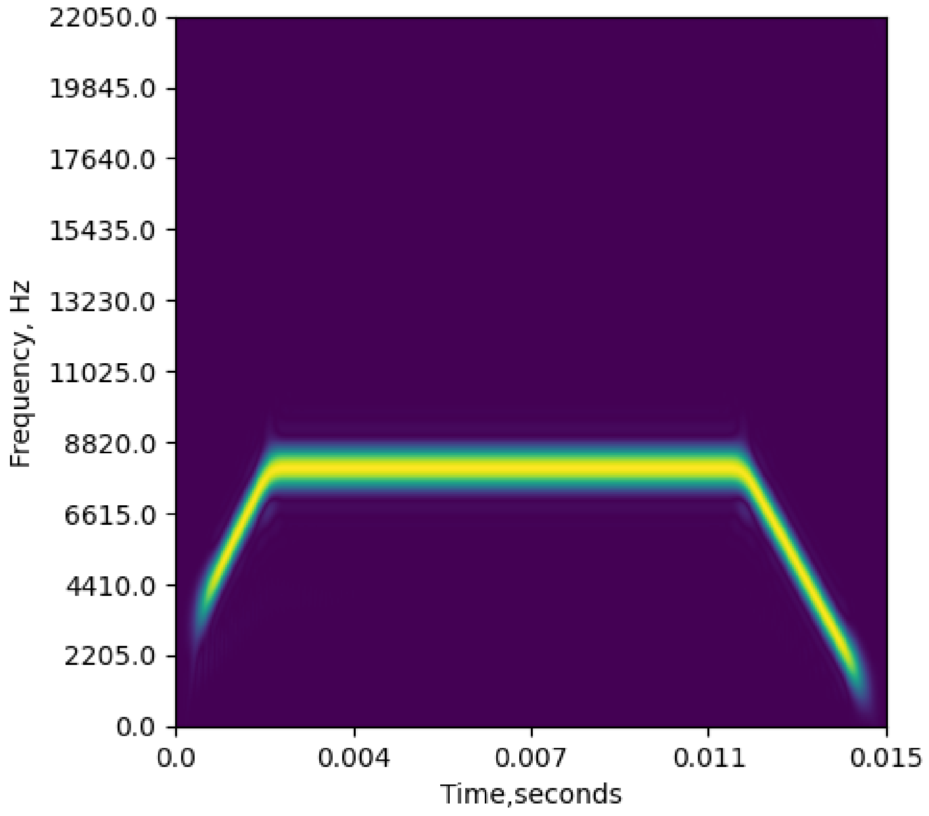


The PWVD is a somewhat new representation to most people, so let's just check out an example

```
pwvd = pwvd_transform(cffm_call, fs)
```

The output is an NsamplesxNsamples matrix, where Nsamples is the number of samples in the original audio.

```
plt.figure()
plt.imshow(abs(pwvd), origin='lower')
num_rows = pwvd.shape[0]
plt.yticks(np.linspace(0,num_rows,11), np.linspace(0, fs*0.5, 11))
plt.ylabel('Frequency, Hz')
plt.xticks(np.linspace(0,num_rows,5),
           np.round(np.linspace(0, cffm_call.size/fs, 5),3))
plt.xlabel('Time,seconds')
```

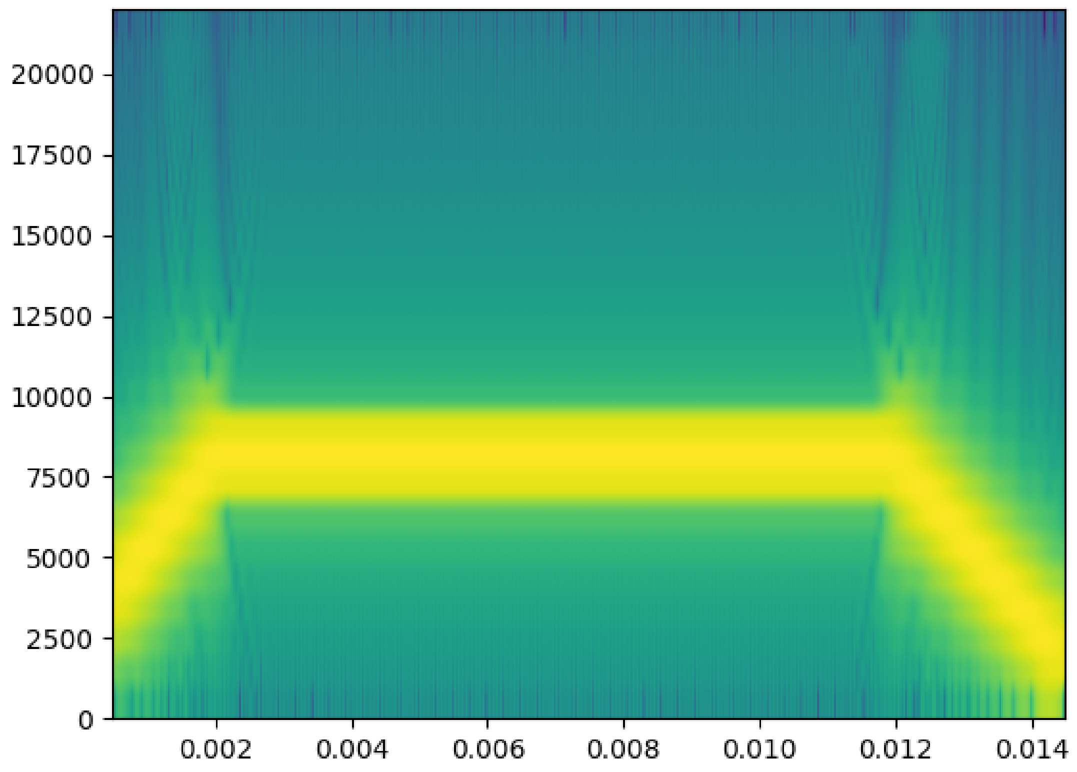


Out:

```
Text(0.5, 0, 'Time,seconds')
```

In comparison to the 'crisp' time-frequency representation of the PWVD, let's compare how a spectrogram with comparable parameters looks:

```
onems_samples = int(fs*0.001)
plt.figure()
out = plt.specgram(cffm_call, Fs=fs, NFFT=onems_samples, noverlap=onems_samples-1)
```



The dominant frequency at each sample can be tracked to see how the the frequency changes over time. Let's not get into the details right away, and proceed with the segmentation first.

```
cf, fm, info = segment_call_into_cf_fm(cffm_call, fs, segment_method='pwvd',
                                                                    window_
↪size=50)
```

The *segment_call_into_cf_fm* provides the estimates of which samples are CF and FM. The *info* object is a dictionary with content that varies according to the segmentation method used. For instance:

```
info.keys()
```

Out:

```
dict_keys(['moving_dbrms', 'geq_signal_level', 'raw_fp', 'acc_profile', 'spikey_
↪regions', 'fmrate', 'cleaned_fp', 'fitted_fp'])
```

**Total running time of the script:** ( 0 minutes 0.851 seconds)

### 2.1.3 The peak-percentage method

The peak percentage method works if the constant frequency portion of a sound segment is the highest frequency. For instance, in CF-FM bat calls, the calls typically have a CF and one or two FM segments connected.

This method is loosely based on the spectrogram based CF-FM segmentation in [1], but most importantly it differs because it is implemented completely in the time-domain.

#### How does it work?

A constant frequency segment in any sound leads to a peak in the power spectrum. The same audio is high-passed and low-passed at a threshold frequency that's very close (eg. 99% of the peak frequency)

> and just below the peak frequency. This creates two versions of the same sound, one with an emphasis on the CF, and one with the emphasis on

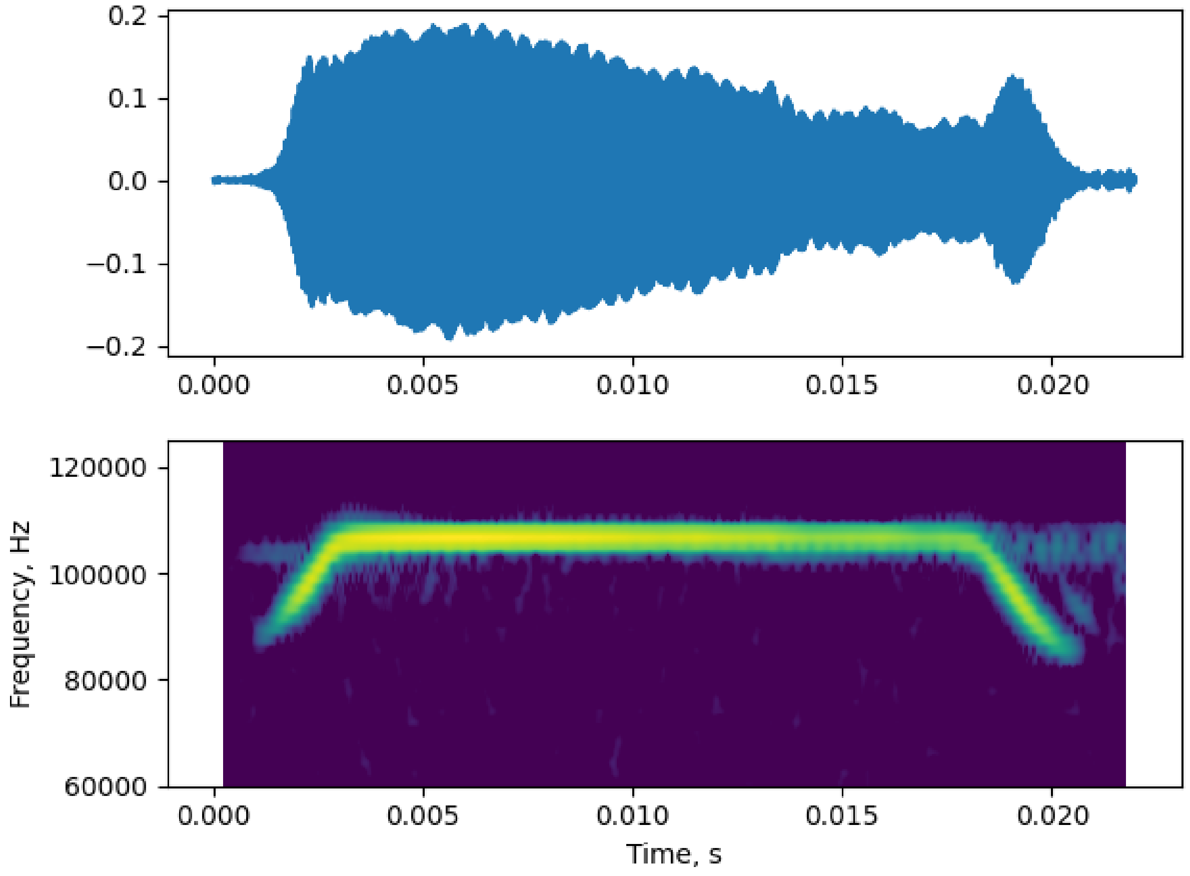
the FM. By comparing the two sounds, the segmentation proceeds to detect CF and FM parts.

References

**[1]Schoeppler, D., Schnitzler, H. U., & Denzinger, A. (2018). Precise Doppler shift compensation in the hipposiderid bat,** Hipposideros armiger. Scientific reports, 8(1), 1-11.

```
import matplotlib.pyplot as plt
import scipy.signal as signal
import itsfm
from itsfm.simulate_calls import make_cffm_call
from itsfm.segment import segment_call_into_cf_fm
```

```
from itsfm.data import example_calls, all_wav_files

bat_rec = list(map( lambda X: '2018-08-17_34_134' in X, all_wav_files))
index = bat_rec.index(True)
audio, fs = example_calls[index] # load the relevant example audio

w,s = itsfm.visualise_sound(audio,fs, fft_size=128)
# set the ylim of the spectrogram narrow to check out the call in more detail
s.set_ylim(60000, 125000)
```



Out:

```
(60000.0, 125000.0)
```

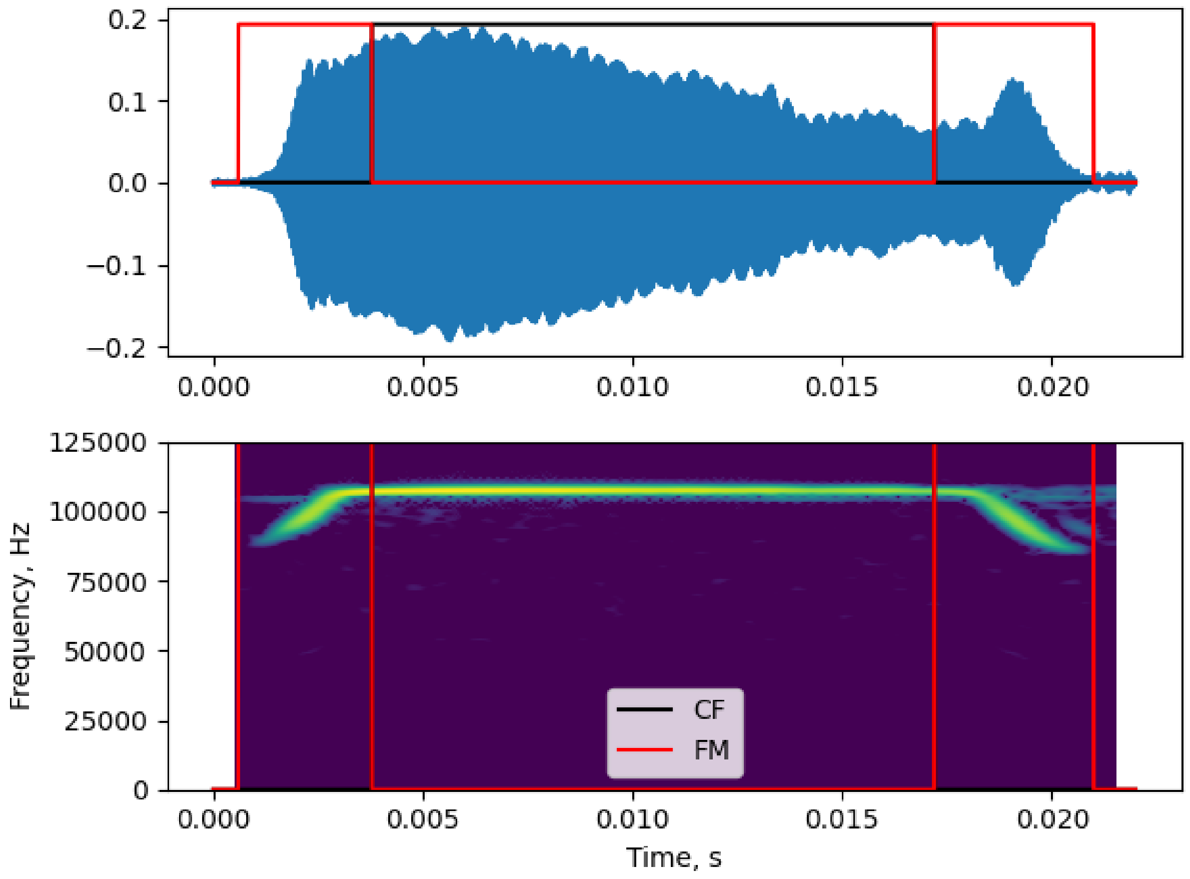
Now, let's proceed to run the peak-percentage based segmentation.

```
non_default_params = {'segment_method':'peak_percentage',
                      'window_size':int(fs*0.0015),
                      'signal_level':-30,
                      'double_pass':True}
outputs = itsfm.segment_and_measure_call(audio, fs,
                                         **non_default_params)

# load the results into a convenience class
# itsFMinspector parses the output and creates diagnostic plots
# and access to the underlying diagnostic data itself

output_inspect = itsfm.itsFMInspector(outputs, audio, fs)
```

### Verifying the CF-FM segmentations

Here, let's see what the output of the peak-percentage method shows

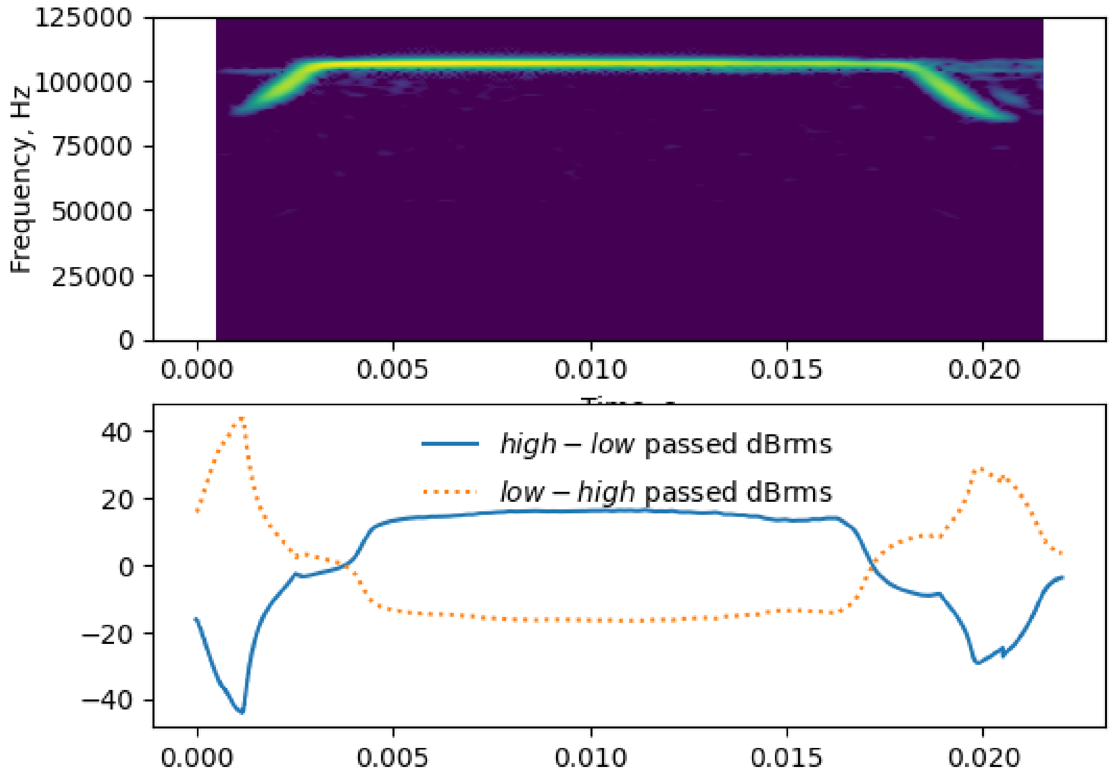
```
output_inspect.visualise_cffm_segmentation()
plt.tight_layout()
plt.savefig('pwvd_cffm_segmentation.png')
```



### Low/high passed audio profiles

Let's also take a look at the low and high -passed audio profiles. The regions where the dB rms of the high-passed audio is greater than the low-passed audio is considered CF and vice-versa is considered FM.

```
spec, profiles = output_inspect.visualise_pkpctage_profiles()
profiles.legend(loc=9, frameon=False)
plt.savefig('pkpctage_profiles.png')
```



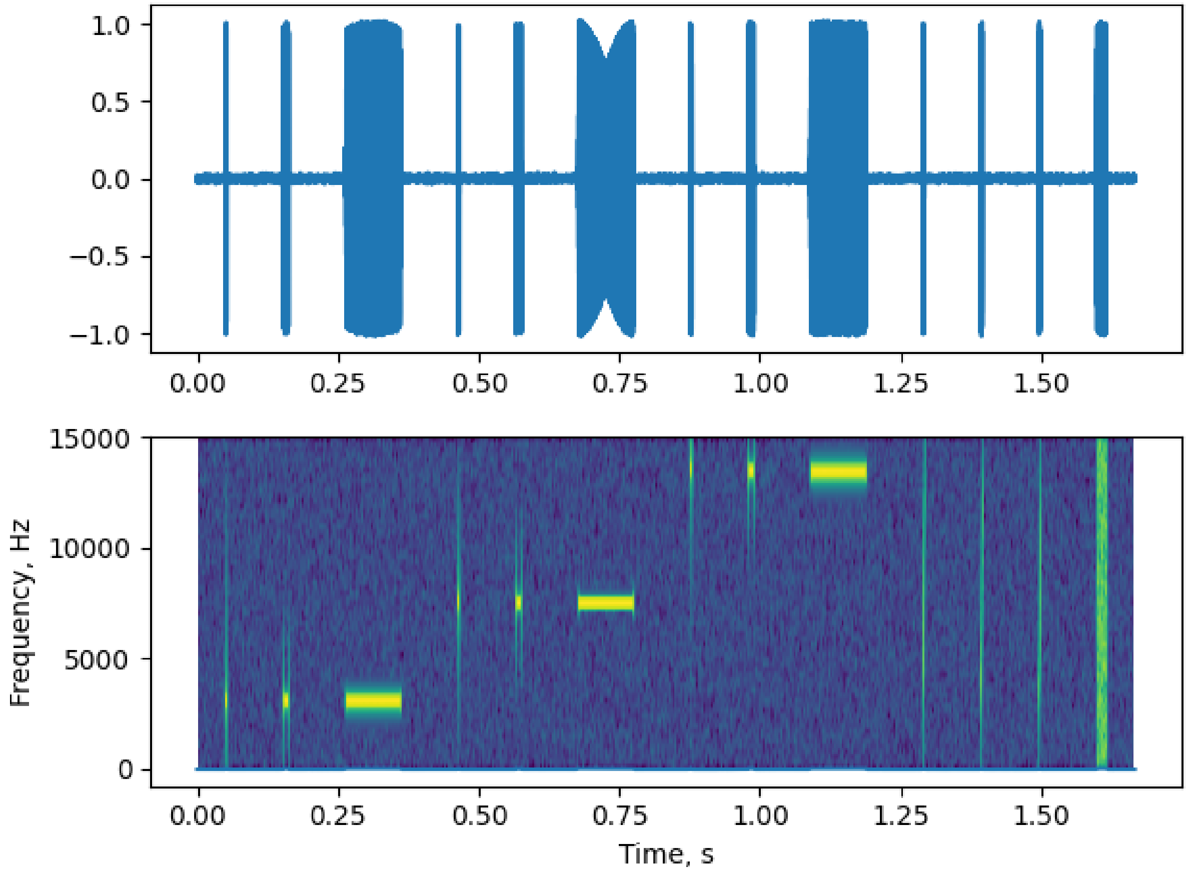
The two profiles match the expected CF/FM regions fairly well.

**Total running time of the script:** ( 0 minutes 2.397 seconds)

### 2.1.4 Finding the right parameter setting with the call zoo

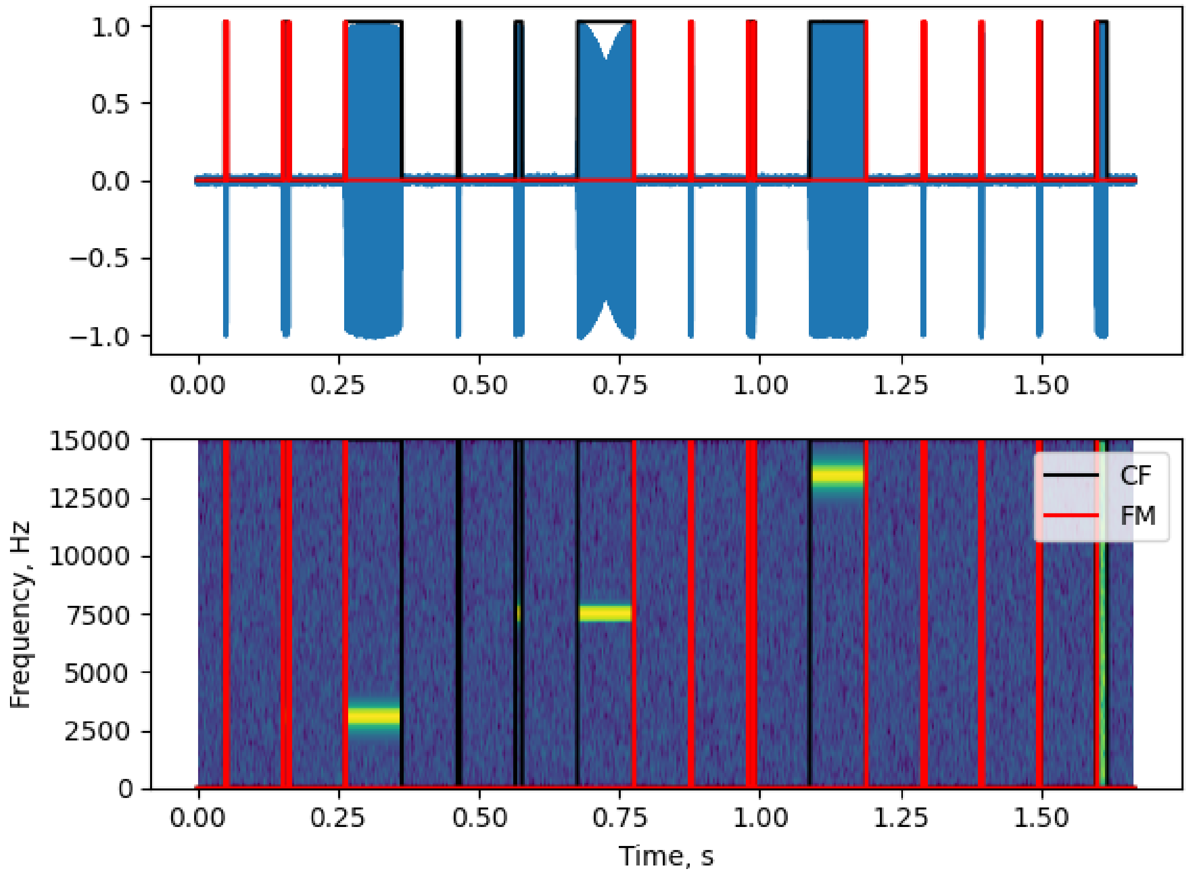
The 'call zoo' is an inbuilt collection of sounds which were made for testing the package. It has a variety of sounds to assess the accuracy of the segmentation and measuring capabilities of the pacakge.

```
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(82319)
import itsfm
from itsfm.simulate_calls import make_call_zoo, add_noise
from itsfm.segment import segment_call_into_cf_fm


fs=30000

freq_profile, call_zoo = make_call_zoo(fs=fs, gap=0.1)
add_noise(call_zoo, -40)

itsfm.visualise_sound(call_zoo, fs, fft_size=128)
itsfm.plot_movingdbrms(call_zoo,fs, window_size=int(fs*0.001))
```



Now, let's run the segmentation on this sound

```
segment_parameters = {'window_size' : int(fs*0.001),
                      'segment_method':'pwvd',
                      'signal_level': -30,
                      'sample_every':0.25*10**-3}
segment_out = segment_call_into_cf_fm(call_zoo, fs, **segment_parameters)
cf, fm, info = segment_out
itsfm.visualise_cffm_segmentation(cf,fm,call_zoo,fs, fft_size=128)
```



Out:

```
(<AxesSubplot:>, <AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>)
```

Now, the results show that some sounds are being recognised, but a closer look the results indicate there's too much silence on either side of the sounds, and the FM sweeps at the end have been mis-classified as CF sounds. Why is this happening? This kind of apparent errors typically come from a bad match between the recordings properties and the default parameter values in place. The 'issues' can be sorted out most of the time by playing around with the parameter values.

**Total running time of the script:** ( 0 minutes 13.804 seconds)

### 2.1.5 Bird song example

Here we'll use the recordings of a common bird, the great tit (*Parus major*). The recording is an excerpt of a bigger recording made by Jarek Matusiak (Xeno Canto, XC235125) - give it a listen here.

#### Note

As of version 0.0.X, this recording is also a very good example of how multi-harmonic sounds can't be tracked very well!

```
import matplotlib.pyplot as plt
plt.rcParams['agg.path.chunksize'] = 10000
import numpy as np
import scipy.signal as signal
import itsfm
from itsfm.data import example_calls, all_wav_files,folder_with_audio_files

great_tit_rec = list(map( lambda X: 'Parus_major_Poland' in X, all_wav_files))
index = great_tit_rec.index(True)
full_audio, fs = example_calls[index] # load the relevant example audio

#
w,s = itsfm.visualise_sound(full_audio, fs, fft_size=512)
s.set_ylim(0,10000)
```

The complete audio recording takes a long time to run, and so let's focus on the sections between 0.8-1.5s. It contains one example of the three types of the great tit's calls.

```
t_start, t_stop = 0.8, 1.5
selection = slice(int(fs*t_start), int(fs*t_stop))
audio = full_audio[selection]

w,s = itsfm.visualise_sound(audio, fs, fft_size=256)
s.set_ylim(0,10000)
```

The bird song has a three types of calls, a smooth frequency modulated sweep a constant frequency tone, and the last element has a rather rapid frequency sweep which then transitions into a constant frequency segment.

#### Setting the correct signal level

The frequency profile of a sound is calculated only for those chunks of the audio that are above a threshold dBrms, called the *signal_level*. Make a moving dBrms plot to see which a sensible signal threshold to set

```
plt.figure()
a = plt.subplot(211)
itsfm.plot_movingdbrms(audio, fs, window_size=int(0.005*fs))
plt.subplot(212, sharex=a)
out = plt.specgram(audio, Fs=fs, NFFT=256, noverlap=255)
a.grid()
```

With this plot, we can see that a level of -34 dB rms with a 5ms window will choose the song elements well. Let's try it out.

```
non_default_params = {
                    'segment_method':'pwvd',
                    'signal_level':-34,
                    'window_size':int(fs*0.005),
                    'pwvd_window':0.010,
                    'medianfilter_size':0.005,
                    'sample_every':20*10**-3
                    }

output = itsfm.segment_and_measure_call(audio, fs,**non_default_params )

bird_inspect = itsfm.itsFMInspector(output,audio,fs, fft_size=512)
```

First, let's check if we're actually picking up the bird signals reliable with the *signal_level* we chose.

```
bird_inspect.visualise_geq_signallevel()
```

And let's look at the measurements

```
bird_inspect.measurements
```

We see there are 9 valid sound segments picked up, and their start and stop times are displayed. How have they been classified?

```
bird_inspect.visualise_cffm_segmentation()
```

Whoops, it seems like they've all been classified as CF parts. Even though the audio actually has FM parts in it, or so we think. Well, whether something is frequency modulated or not is set by the *fmrate_threshold*. We need to correct the situation by setting it to a more sensible value.

### Setting a non-default FM rate

The segmentation of sounds into FM and CF regions happens by looking at the FM rate over the sound. Whenever a region crosses the FM rate threshold, it is considered an FM region. Let's check out the FM rate over the sound with the current parameters, and then choose a more sensible, non-default *fmrate_threshold* parameter.

```
bird_inspect.visualise_fmrate()
```

As you can see the constant frequency and modulated parts are being tracked pretty well, but they're not being classified properly. The CF or FM classification is based on the estimated reate of frequency modulation over the sound, ,the *fmrate_threshold*. The default if 1kHz/ms, which is a *lot* if you think about it. At this rate, the bird would have gone from 20kHz to 20 Hz in about 20 milliseconds, and you would have *barely* heard it. This default FM rate is set to pick up FM regions in bats, and so it needs to be adjusted for other animals.

The fm segments in the great tits song correspond to an FM rate of >= 0.005 kHz/ms. Remember that all frequency modulation rates are in kHz/ms. Let's set this as the threshold and proceed to segment.

```
non_default_params['fmrate_threshold'] = 0.02 #

output_newrate = itsfm.segment_and_measure_call(audio, fs,
                                        **non_default_params)

newrate_inspect = itsfm.itsFMInspector(output_newrate, audio, fs, fft_size=512)
```

And let's look at the measurements

```
newrate_inspect.measurements
```

Let's check the the segmentation output again now

```
newrate_inspect.visualise_cffm_segmentation()
```

So, it's improved, and there seem to be mainly FM regions in at the edges of the sounds. Is this real, or an artifact of the frequency profile fitting. Let's inspect the actual frequency profiles underlying the *fmrate* calcultions

```
newrate_inspect.visualise_frequency_profiles()
```

The issue with the third element is that there's a multiple harmonics and this may cause the local frequency estiamte to vary up and down . We can try to overcome the effect of non-peak frequencies using the `percentile` parameter. The `percentile` essentially

*to be completed. . . .*

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 2.1.6 Setting the correct *max_acc* value

Some of the methods in the <INSERTNAME> package estimate the instantaneous frequency at sample-level resolution. Most methods will suffer from edge effects which cause the estimated instantaneous frequency to spike especially at the start and end of

> the sound or due to noise.

The typical way these spikes are dealt with is to calculate an aboslute frequency accelaration profile along the frequency profile. Any regions above a certain threshold are considered anomalous, and an (sort of) extrapolation is attempted using the nearest non-anomalous regions.

### An example frequency profile

Let's create an example sound, and use the PWVD method to track the instantaneous frequency over time.

```
import numpy as np
from itsfm.frequency_tracking import generate_pwvd_frequency_profile, frequency_spike_
↪detection
from itsfm.simulate_calls import make_fm_chirp
import matplotlib.pyplot as plt
from itsfm.view_horseshoebat_call import plot_accelaration_profile, time_plot
```

Let's create a hyperbolic chirp, this is a nice example because the the hyperbolic chirp shows a nice variation in frequeny velocity over time. This means the accelaration varies from low–>high. But what is an 'acceptable' value of accelaration to allow. Let's inspect the accelaration profile itself to understand what accelaration values are 'normal' and which values correspond to the spikes caused by the edge effects and noise.

```
fs = 22100
chirp = make_fm_chirp(500, 5000, 0.100, fs, 'logarithmic')

raw_fp, frequency_index = generate_pwvd_frequency_profile(chirp,
                                                          fs, percentile=99)
plt.figure()
time_plot(raw_fp,fs)
```

The spikes caused by edge effects are there here too- even without noise. Let's check out the typical accelaration profile of this sound, and pay special attention to the values towards the ends.

```
acc_plot  = plot_accelaration_profile(raw_fp, fs)
acc_plot.set_ylim(0,0.5) # show a limited y-axis, because the frequency spikes mess␣
↪up the display
```

Remember that the accelaration of the frequency is calcualted at a per-sample resolution and thus may not show too much variation – but the profile still shows outliers! Looking at this plot we can see that a value $\geq 0.1$ kHz/ms $^{2}$ is likely to be an outlier.

Now, we know a way to set sensible *max_acc* values for our own recordings - let's see how this translates to outlier detection in the frequency profile:

```
spikey_regions, acc_profile = frequency_spike_detection(raw_fp, fs, max_acc=0.1)

plt.figure()
a = plt.subplot(211)
time_plot(raw_fp, fs)
plt.plot( np.argwhere(spikey_regions)/fs, raw_fp[spikey_regions],
         '*', label='Anomalous spikes in frequency profile')
plt.legend()
a.set_title('Detected spikes in frequency profile')
a.set_ylabel('Frequency, Hz')
a.set_xticks([])
b = plt.subplot(212)
time_plot(acc_profile, fs)
plt.plot( np.argwhere(spikey_regions)/fs, acc_profile[spikey_regions], '*')
b.set_ylim(0,0.5)
b.set_title('Frequency accelaration profile')
b.set_ylabel('Frequency accelaration, $kHz/ms^{2}$')
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

### 2.1.7 Inbuilt and custom measurements on CF and FM segments

By default, a baic set of information/measurements is given for each recognised CF/FM segment in the input audio, its start, stop and duration.

Let's begin by making a synthetic CF-FM call which looks a lot like a horseshoe/leaf nosed bat's call

```
import matplotlib.pyplot as plt
import scipy.signal as signal
from itsfm.simulate_calls import make_cffm_call
from itsfm.view_horseshoebat_call import visualise_call
from itsfm.user_interface import segment_and_measure_call
```

Lets now create a sound that's got only one CF and one FM component in it. Horseshoe/leaf nosed bats emit these kinds of calls too.

```
fs = 44100
call_props = {'cf':(8000, 0.01),
'upfm':(8000,0.002), # not that the 'upfm' frequency starts at the CF frequency!
'downfm':(100,0.003)}

cffm_call, freq_profile = make_cffm_call(call_props, fs)
cffm_call *= signal.tukey(cffm_call.size, 0.1)
```

(continues on next page)

(continued from previous page)

```
w,s = visualise_call(cffm_call, fs, fft_size=64)
```

Now, segment and measure using the 'peak pecentage' method

```
output = segment_and_measure_call(cffm_call, fs,
                                  segment_method = 'peak_percentage',
                                  peak_percentage=0.95,
                                  window_size=44)
segment_info, call_parts, results, _ = output
```

If everything went well, the output should give us one CF and one FM component. The parameters may need to be tweaked based on the sampling rate and the amount of frequency modulation in the calls. This is true especially for sounds with a 'curvature' in the frequency profile, because sometimes the frequency change may be gradual and then become sudden, eg in the transition between CF and FM in this example call.

Another important aspect to notice is that the window size has been set to 44 samples. This corresponds to a short window of ~0.1ms. This short window size is used to compare the relative CF and FM emphasised dB rms profiles (see 'The peak percentage method').

```
print(results)
```

What if we want more than just the duration of each component? There are inbuilt functions such which allow the measurement of the rms, peak-amplitude, peak frequency and terminal frequency of each segment. Let's get the peak frequency and peak amplitude for all segments

```
from itsfm.measurement_functions import measure_peak_frequency, measure_peak_amplitude

added_measures = [measure_peak_amplitude, measure_peak_frequency]

output = segment_and_measure_call(cffm_call, fs,
                                  peak_percentage=0.95,
                                  window_size=44,
                                  measurements=added_measures)

segment_info, call_parts, results, _ = output

print(results)
```

Now, what if this is not what we're looking for and we needed to get, say, the *dB peak* amplitude? This calls for a custom measurement function. Each measurement function follows a particular pattern of three inputs and one output. See the *measurement_function* documentation or call it through the help

```
from itsfm import measurement_functions as measure_funcs
help(measure_funcs)
```

Let's also take a look at the source code for one of the measurement functions we just used above *measure_peak_amplitude*:

```
import inspect
print(inspect.getsource(measure_peak_amplitude))
```

The output needs to be a dictionary with the measurement names and values in the keys and items respectively.

So, now let's get the dB peak value of our audio segments

```
import numpy as np
from itsfm.signal_processing import dB

def measure_dBpeak(audio, fs, segment, **kwargs):
    relevant_audio = audio[segment]
    dB_peak_value = dB(np.max(np.abs(relevant_audio)))
    return {'dB_peak': dB_peak_value}


output = segment_and_measure_call(cffm_call, fs,
                                  peak_percentage=0.95,
                                  window_size=44,
                                  measurements=[measure_dBpeak])

segment_info, call_parts, results, _ = output

print(results)
```

So, looking at the dB peak value tells us that both CF and FM components are pretty strong, and of comparable levels. Both are close to 0 dB (re 1), which means they're pretty close to the maximum signal value.

Just like the *measure_dBpeak*, we can chain a series of inbuilt or custom measurement functions in a list - and the outputs will all appear as a wide-formate Pandas DataFrame.

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 2.1.8 Segmenting real-world sounds correctly with synthetic sounds

It's easy to figure out if a sound is being correcly segmented if the signal at hand is well defined, and repeatable, like in many technological/ engineering applications. However, in bioacoustics, or a more open-ended field recording situation, it can be very hard to know the kind of signal that'll be recorded, or what its parameters are.

Just because an output is produced by the package, it doesn't always lead to a meaningful result. Given a set of parameters, any function will produce an output as long as its sensible. This means, with one set of parameters/methods the CF segment might be 10ms long, while with another more lax parameter set it might be 20ms long! Remember, as always, GIGO (Garbage In, Garbage Out):P.

How to segment a sound into CF and FM segments in an accurate way?

### Synthetic calls to the rescue

Synthetic calls are sounds that we know to have specific properties and can be used to test if a parameter set/ segmentation method is capable of correctly segmenting our real-world sounds and uncovering the true underlying properties.

The *simulate_calls* module has a bunch of helper functions which allow the creation of FM sweeps, constant frequency tones and silences. In combination, these can be used to get a feeling for which segmentation methods and parameter sets work well for your real-world sound (bat, bird, cat, <insert sound source of choice>)

### Generating a ‘classical’ CF-FM bat call

```
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as signal
from itsfm.simulate_calls import make_cffm_call,make_tone, make_fm_chirp, silence
from itsfm.view_horseshoebat_call import visualise_call
from itsfm.segment_horseshoebat_call import segment_call_into_cf_fm
from itsfm.signal_processing import dB, rms

fs = 96000
call_props = {'cf':(40000, 0.01),
                     'upfm':(38000,0.002),
                     'downfm':(30000,0.003)}

cffm_call, freq_profile = make_cffm_call(call_props, fs)
cffm_call *= signal.tukey(cffm_call.size, 0.1)


w,s = visualise_call(cffm_call, fs, fft_size=128)
```

Remember, the terminal frequencies and durations of the CF-FM calls can be adjusted to the calls of your species of interest!!

### A multi-component bird call

Let’s make a sound with two FMs and CFs, and gaps in between

```
fs = 44100

fm1 = make_fm_chirp(1000, 5000, 0.01, fs)
cf1 = make_tone(5000, 0.005, fs)
fm2 = make_fm_chirp(5500, 9000, 0.01, fs)
cf2 = make_tone(8000, 0.005, fs)
gap = silence(0.005, fs)

synth_birdcall = np.concatenate((gap,
                                 fm1, gap,
                                 cf1, gap,
                                 fm2, gap,
                                 cf2,
                                 gap))

w, s = visualise_call(synth_birdcall, fs, fft_size=64)
```

### Let there be Noise

Any kind of field recording *will* have some form of noise. Each of the the segmentation methods is differently susceptible to noise, and it’s a good idea to test how well they can tolerate it. For starters, let’s just add white noise and simulate different signal-to-noise ratios (SNR).

```
noisy_bird_call = synth_birdcall.copy()
noisy_bird_call += np.random.normal(0,10**(-10/20), noisy_bird_call.size)
noisy_bird_call /= np.max(np.abs(noisy_bird_call)) # keep sample values between +/- 1
```

Estimate an approximate SNR by looking at the rms of the gaps to that of a song component

```
level_background = dB(rms(noisy_bird_call[gap.size]))

level_song = dB(rms(noisy_bird_call[gap.size:2*gap.size]))

snr_approx = level_song-level_background

print('The SNR is approximately: %f'%np.around(snr_approx))

w, s = visualise_call(noisy_bird_call, fs, fft_size=64)
```

We could try to run the segmentation + measurement on a noisy sound straight away, but this might lead to poor measurements. Now, let's bandpass the audio to remove the ambient noise outside of the song's range.

**Total running time of the script:** ( 0 minutes 0.000 seconds)

## 2.2 Detailed Examples Gallery

This is a collection of detailed and more technically oriented examples illustrating the effect and role of various parameters on the effectiveness of frequency tracking and CF-FM segmentation.

### 2.2.1 'Difficult' example

The <insertname> package was mainly designed keeping horseshoe bat calls in mind. These calls are high-frequency (>50kHz) and short (20-50ms) sounds which are quite unique in their structure. Many of the default parameter values reflect the original dataset. In fact, many of the default parameters don't even work for some of the example datasets themselves! It should be no surprise that unpredictable things happen when segmentation and tracking is run with default values.

This example will guide you through understanding the various parameters that can be tweaked and what effect they actually have. It is not an exhaustive treatment of the implementation, but a 'lite' intro. For more details of course, the original documentation should hopefully be helpful.

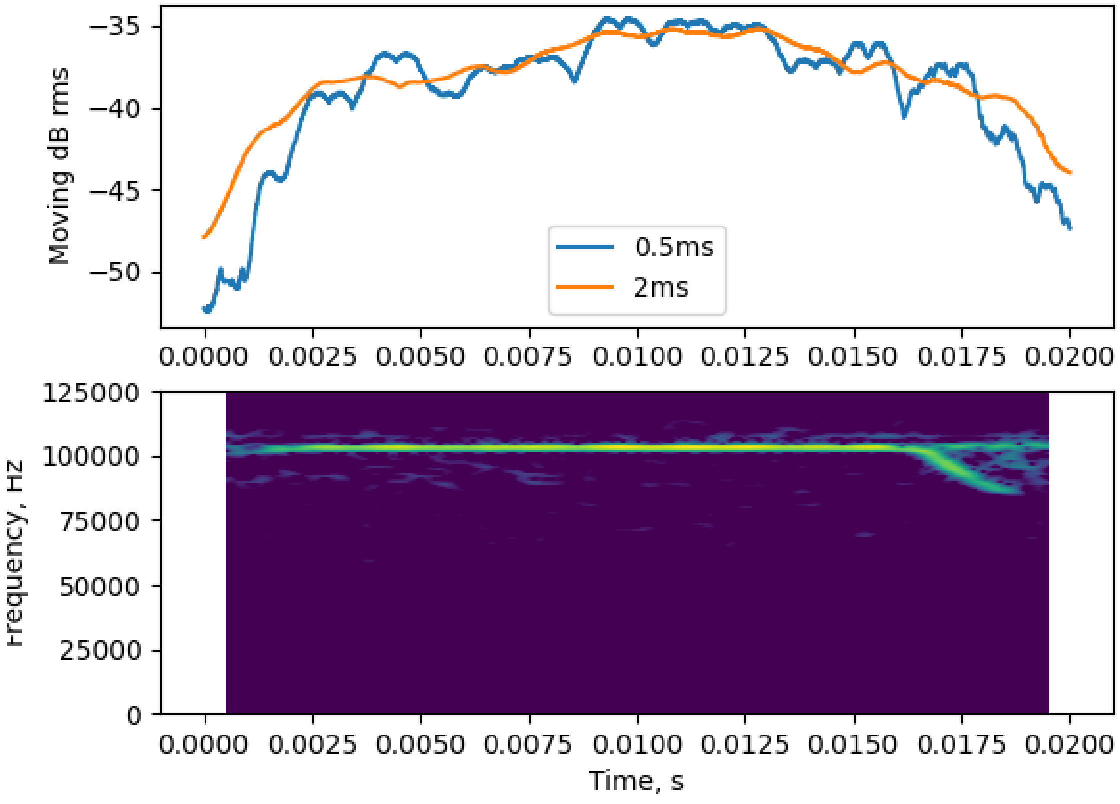
```
from matplotlib.lines import Line2D
import matplotlib.pyplot as plt
import itsfm
from itsfm.data import example_calls, all_wav_files

# a chosen set of tricky calls to illustrate various points

tricky_rec = list(map( lambda X: '2018-08-17_23_115' in X, all_wav_files))
index = tricky_rec.index(True)
audio, fs = example_calls[index] # load the relevant example audio
```

### Step 1: the right *signal_level*

In the given audio segment, the first step is to identify what is background and what is signal. The signal of interest is identified as being above a particular dB rms, as calculated y a moving dB rms window of a user-defined *window_size*.

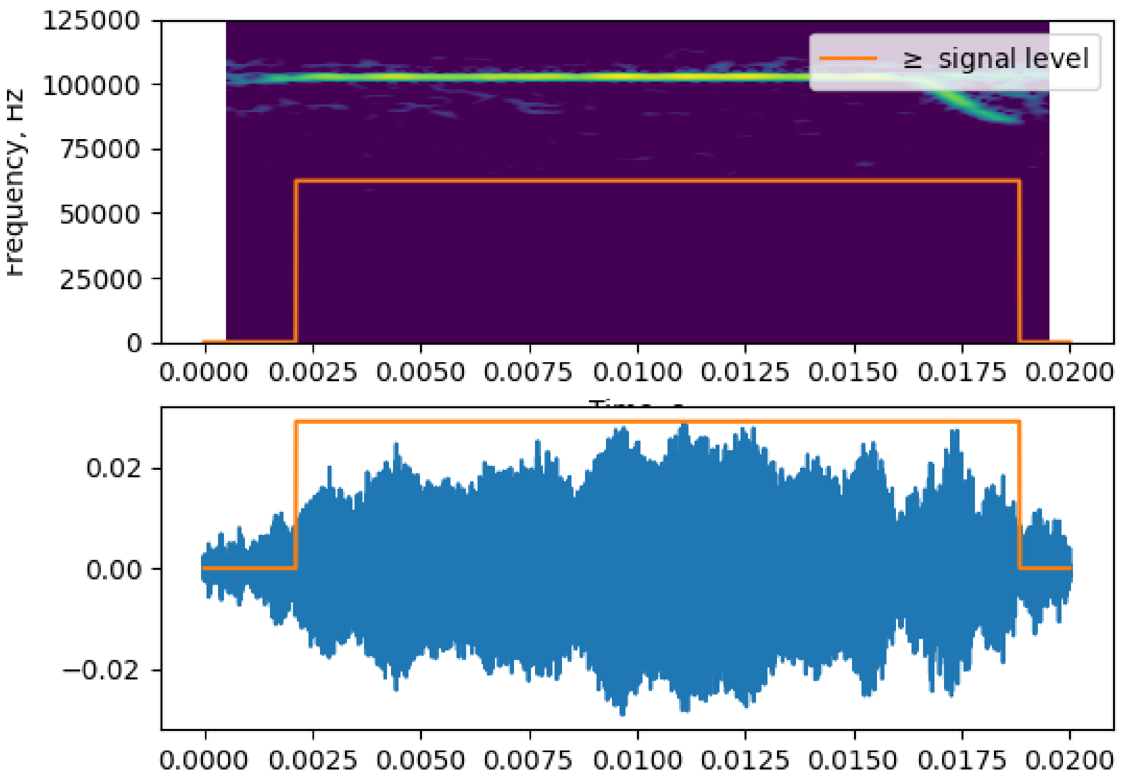
If we want high temporal resolution to segment out the call, we need a short *window_size*. Let's try out 0.5 and 2ms for now.

```
halfms_windowsize = int(fs*0.5*10**-3)
twoms_windowsize = halfms_windowsize*4
plt.figure()
ax = plt.subplot(211)
itsfm.plot_movingdbrms(audio, fs, window_size=halfms_windowsize)
itsfm.plot_movingdbrms(audio, fs, window_size=twoms_windowsize)

first_color = '#1f77b4'
second_color = '#ff7f0e'
custom_lines = [Line2D([0],[0], color=first_color),
                Line2D([1],[1],color=second_color),]
ax.legend(custom_lines, ['0.5ms', '2ms'])
plt.ylabel('Moving dB rms')
plt.subplot(212, sharex=ax)
_ = itsfm.make_specgram(audio, fs);
```



The fact that the 0.5ms moving rms profile is so 'rough' is already a bad sign. The signal of interest is any region/s which are above or equal to the *signal_level*. When the moving rms fluctuates so wildly, the relevant signal region may be hard to capture because it keeps going above and below the threshold - leading to many tiny ‘íslands’. Let’s choose the 2ms *window_size* because it doesn’t fluctuate crazily and is also a relatively short time scale in comparison the the signal duration. -40 dB rms seems to be a sensible value when we compare the approximate start and end times of the signal with the dB rms profile.

```
keywords = {'segment_method':'pwvd',
            'signal_level':-40,
            'window_size':twoms_windowsize}

outputs = itsfm.segment_and_measure_call(audio, fs,**keywords)
output_inspector = itsfm.itsFMInspector(outputs, audio, fs)

output_inspector.visualise_geq_signallevel()
```

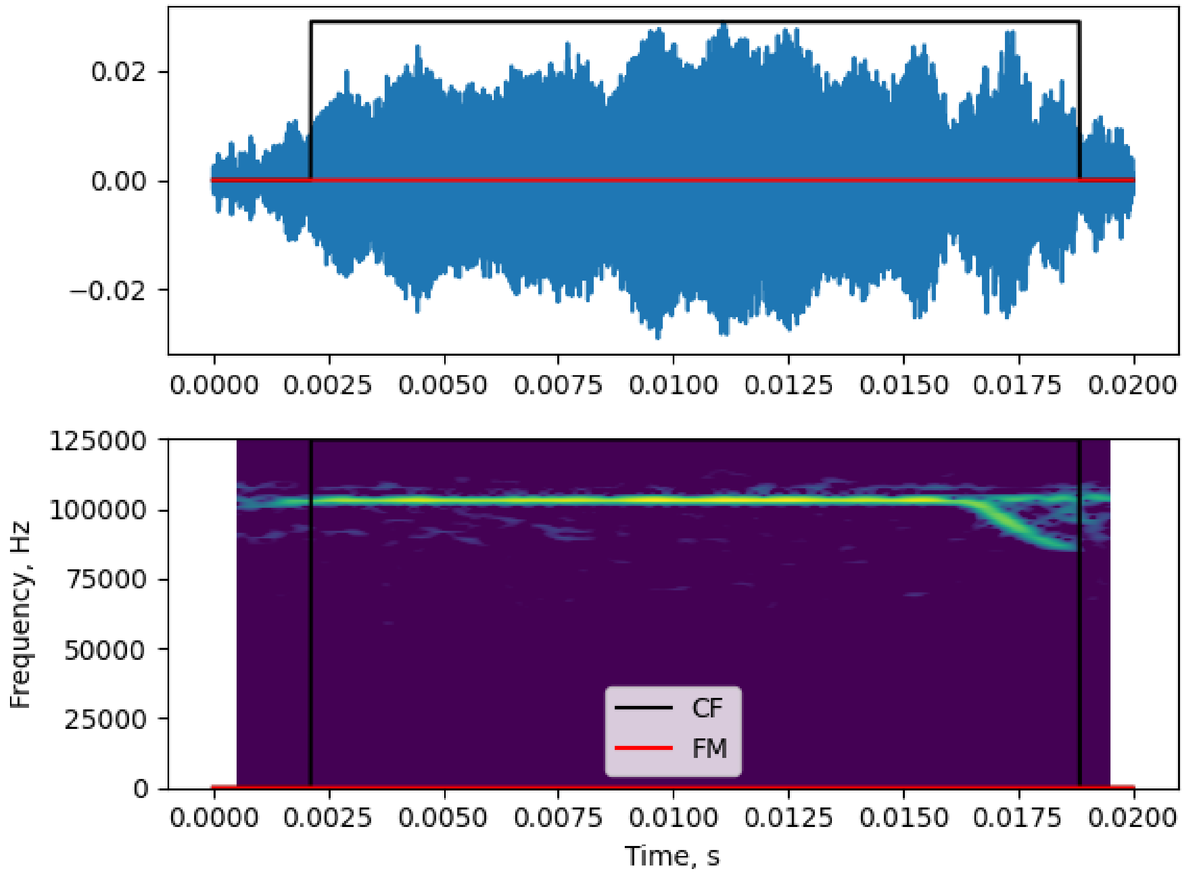


Out:

```
(<AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

Let’s check the output as it is right now

```
output_inspector.visualise_cffm_segmentation()
```

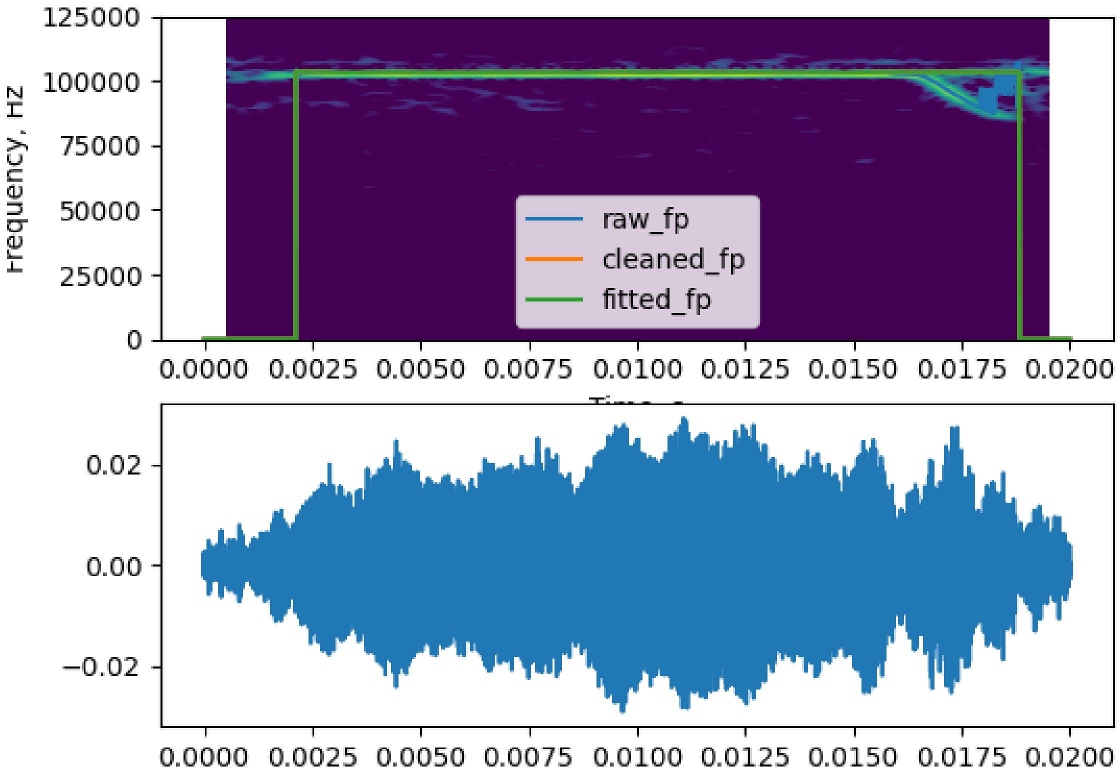


Out:

```
(<AxesSubplot:>, <AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>)
```

Inspect initial outputs The CF-FM segmentation is *clearly* not correct. There's FM component recognised at all - how is this happening? The reason it's not happening is likely because the `fmrate` has been misspecified or the frequency profile wasn't estimated correctly. Let's view the frequency profile first.

```
output_inspector.visualise_frequency_profiles()
```

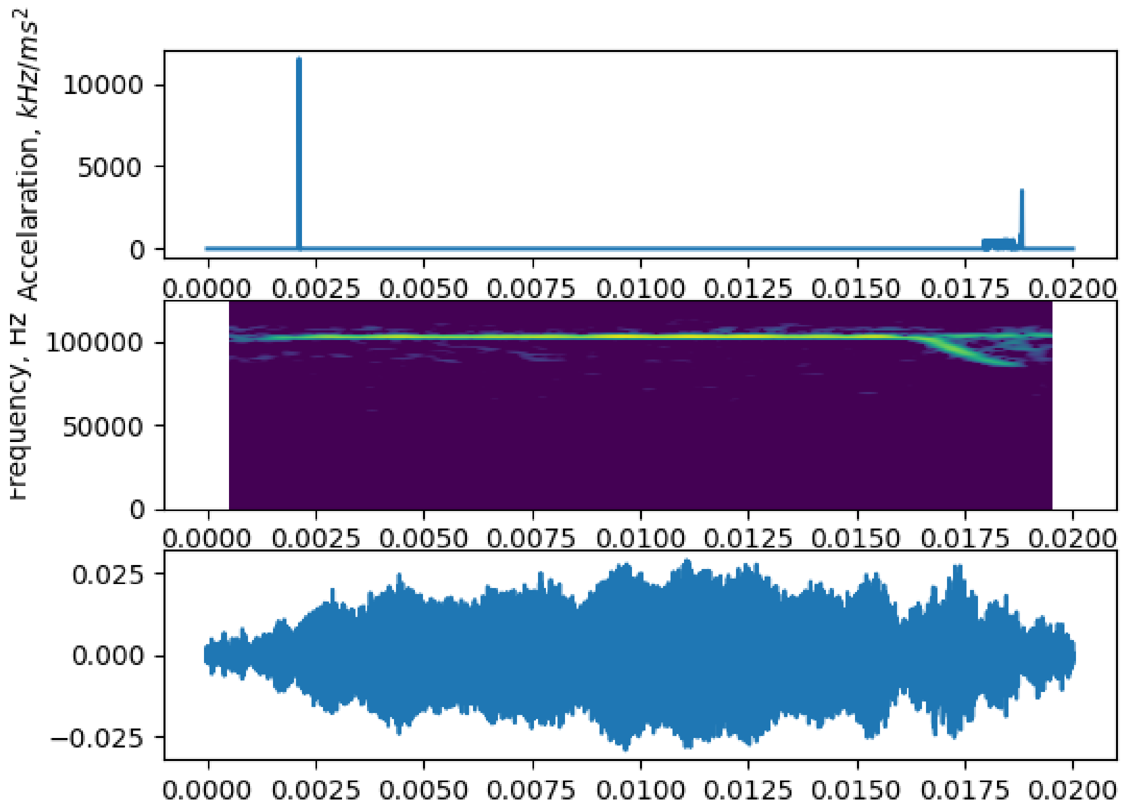


Out:

```
(<AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

The cleaned frequency profile seems to somehow 'ignore' the downward FM sweep in the call. Why is this happening? The 'flatness' in the cleaned frequency profile is likely coming from the spike detection. Spikes in the frequency profile are detected when the 'accelaration' of (the 2nd derivative) the frequency profile increases beyond a threshold. Let's check out the accelaration profile

```
output_inspector.visualise_accelaration()
```

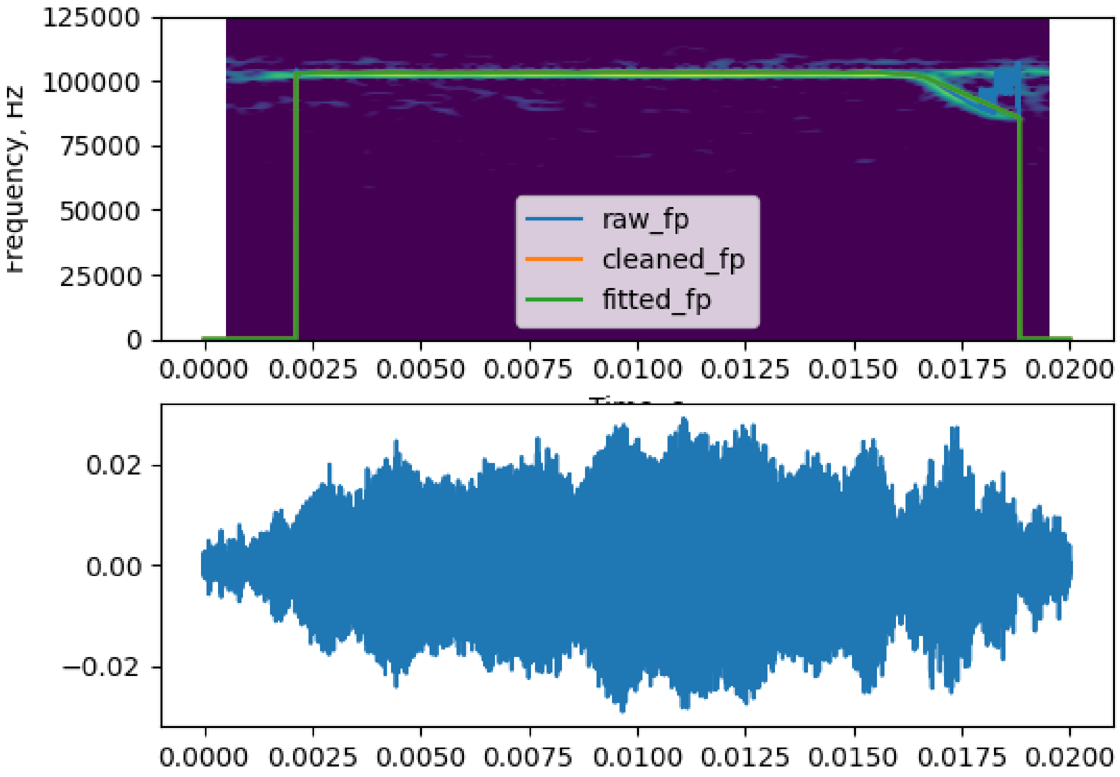


Out:

```
(<AxesSubplot:ylabel='Accelaration, $kHz/ms^{2}$'>, <AxesSubplot:xlabel='Time, s',␣
↪ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

The accelaration profile matches this suspicion. When a spikey region is encountered in the frequency profile in the *pwvd* frequency tracking - it backs up a bit and extrapolates the slope according to what's just behind the spikey region. The 'length' of this backing up in seconds is decided by the `extrap_window`, which is short for extrapolation window. Let's reduce the `extrap_window` and see if the frequency is tracked better.

```
keywords['extrap_window'] = 50*10**-6
outputs_refined = itsfm.segment_and_measure_call(audio, fs,**keywords)
out_refined_inspector = itsfm.itsFMInspector(outputs_refined, audio, fs)
out_refined_inspector.visualise_frequency_profiles()
```

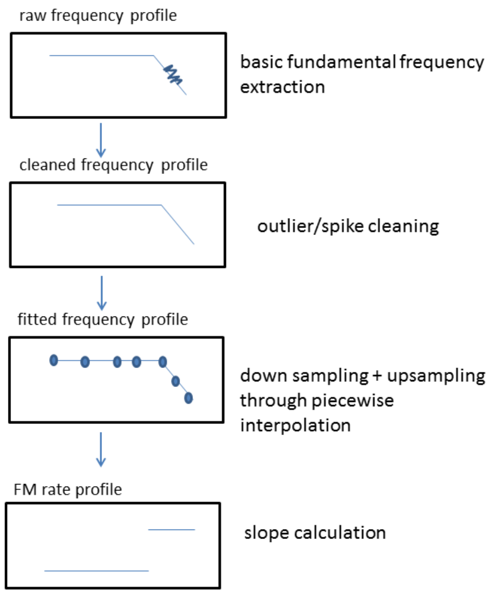


Out:

```
(<AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

So, we've managed to get a much better tracking by telling the algorithm not to 'backup' too much to infer the trend the frequency profile was heading in. It's not perfect, but it does recover the fact that there is an FM region. Remember this issue came up because of the weird reflection of the CF part that is of comparable intensity as the actual FM part itself.

### How the CF-FM segmentation works


raw frequency profile
basic fundamental frequency extraction
cleaned frequency profile
outlier/spike cleaning
fitted frequency profile
down sampling + upsampling through piecewise interpolation
FM rate profile
slope calculation

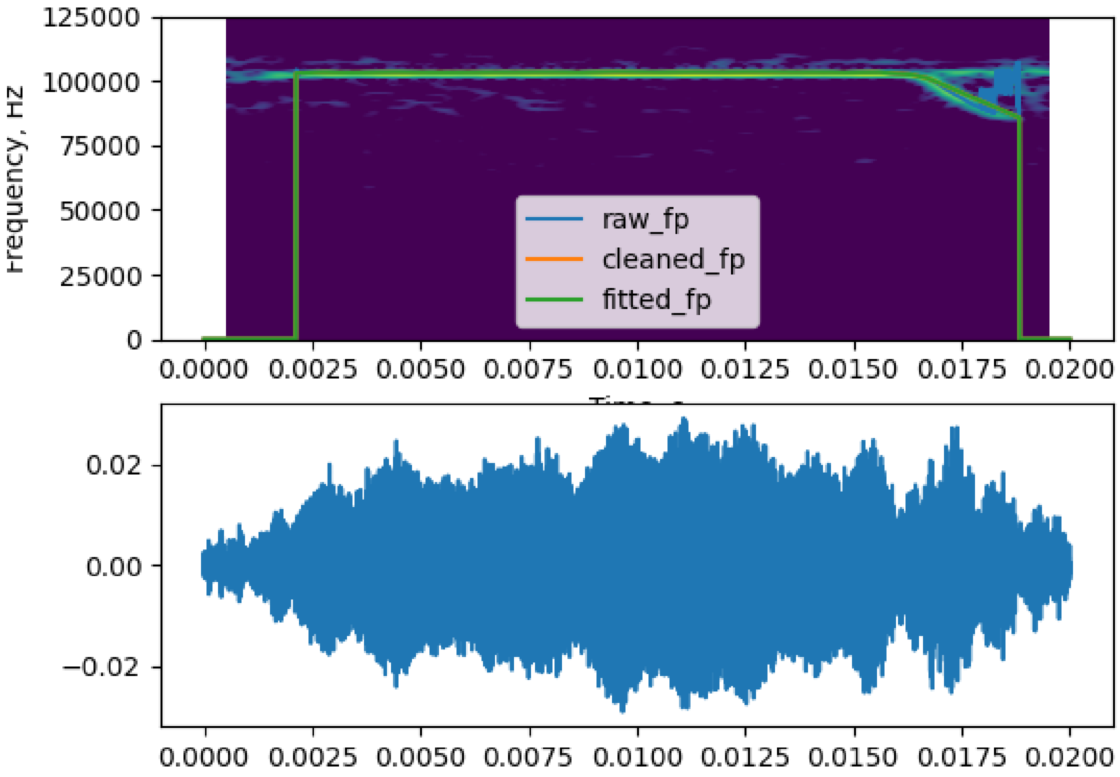
CF-FM segmentation occurs through a multi step process. First the instantaneous frequency of the signal is estimated at a sample-level resolution, the raw frequency profile - *raw_fp*. Then the *raw_fp* is refined as it can be quite noisy because of well, noise, or abrupt changes in signal level across the sound.

Minor jumps will be corrected to give rise to the cleaned frequency profile - *cleaned_fp*. The *cleaned_fp* however, is a very high-resolution look into the sound's frequency profile. Even though the temporal resolution is high, the spectral resolution is limited by the size of the *pwvd_window* (refer to the original docs here). This limited spectral resolution means each sample will not have a unique value. For instance if the frequency of sound is increasing linearly with time, the *cleaned_fp* may actually look like steps going up. These 'steps' cause issues while calculating the rate of frequency modulation - *fmrate*, and so , the *cleaned_fp* is actually downsampled and then upsampled by interpolation. This gives rise to the fitted frequency profile - *fitted_fp*.

The *fitted_fp* captures the local trends and doesn't have the step like nature of *cleaned_fp*. If we were to actually measure frequency modulation from *cleaned_fp* there'd be lots of 0 modulation regions and many very brief bursts of FM regions wherever a 'step' rose or dropped. Thanks to the sample-wise unique values in *fitted_fp* we can now calculate the local variation in frequency modulation across the sound.

Let's now check the frequency profiles once more

```
out_refined_inspector.visualise_frequency_profiles()
```



Out:

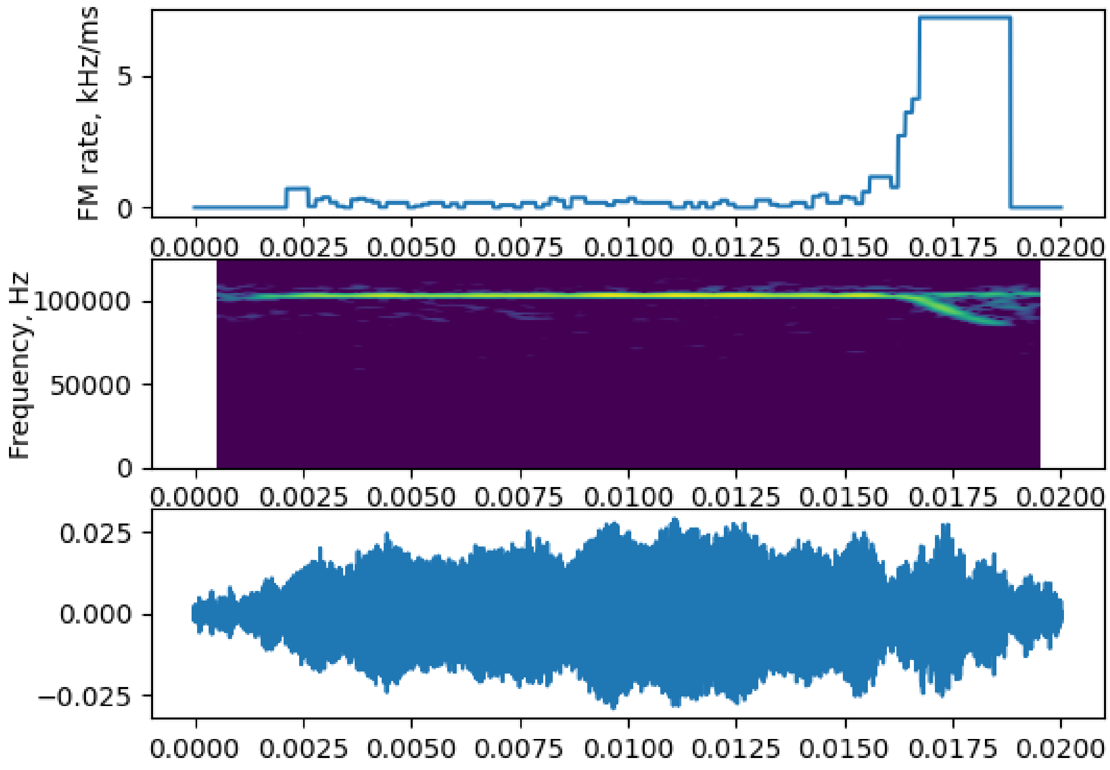
```
(<AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>, <AxesSubplot:>)
```

The raw and cleaned frequency profiles are very similar, though the 'cleanliness' in the *cleaned_fp* is visible especially because the frequency profile doesn' wildly jump around towards the end of the call. The *fitted_fp* also closely matches the *cleaned_fp* though it seems to rise later and drop faster. This is because of the downsampling that happens to estimate the *fmrate*. The rise time is a direct indicator of the downsampling factor, which samples the *cleaned_fp* at periodic intervals, and is thus called *sample_every*. The *sample_every* parameter defaults to 1% of the input signal duration. If the frequency profiles broadly match the actual call as seen coarsely on a spectrogram.

### Step 2: Check the *fmrate* profile

CF and FM parts of a call are segmented based on the rate of frequency modulation they show. The *fmrate* is a np.array with the estimated frequency modulation rate in **kHz/ms**. Yes, pay attention to the units, *it's not kHz/s, but kHz/ms*! Let's take a look at the FM rate profile for this sound.

```
out_refined_inspector.visualise_fmrate()
```

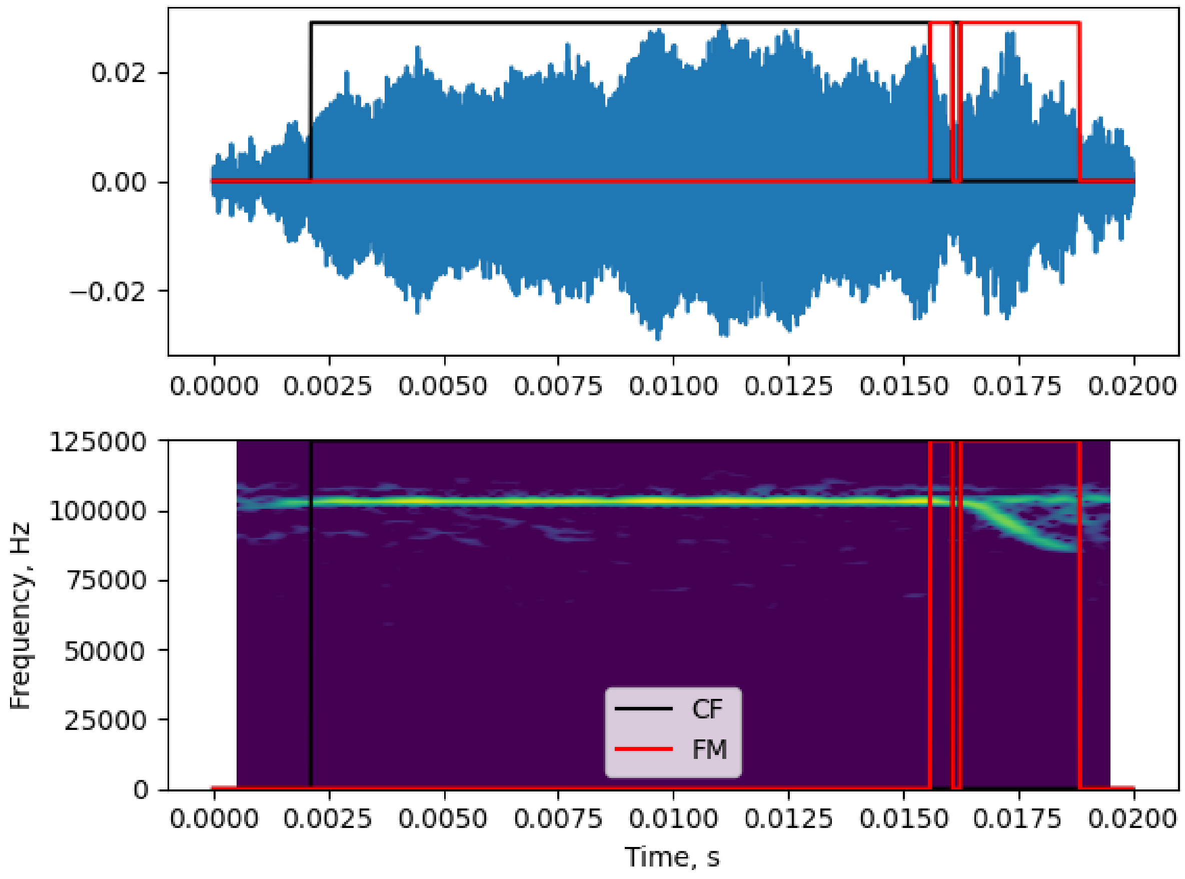


Out:

```
(<AxesSubplot:ylabel='FM rate, kHz/ms'>, <AxesSubplot:xlabel='Time, s', ylabel=
↪'Frequency, Hz'>, <AxesSubplot:>)
```

Let's compare this fmrate profile with the final CF-FM segmentation.

```
out_refined_inspector.visualise_cffm_segmentation()
```

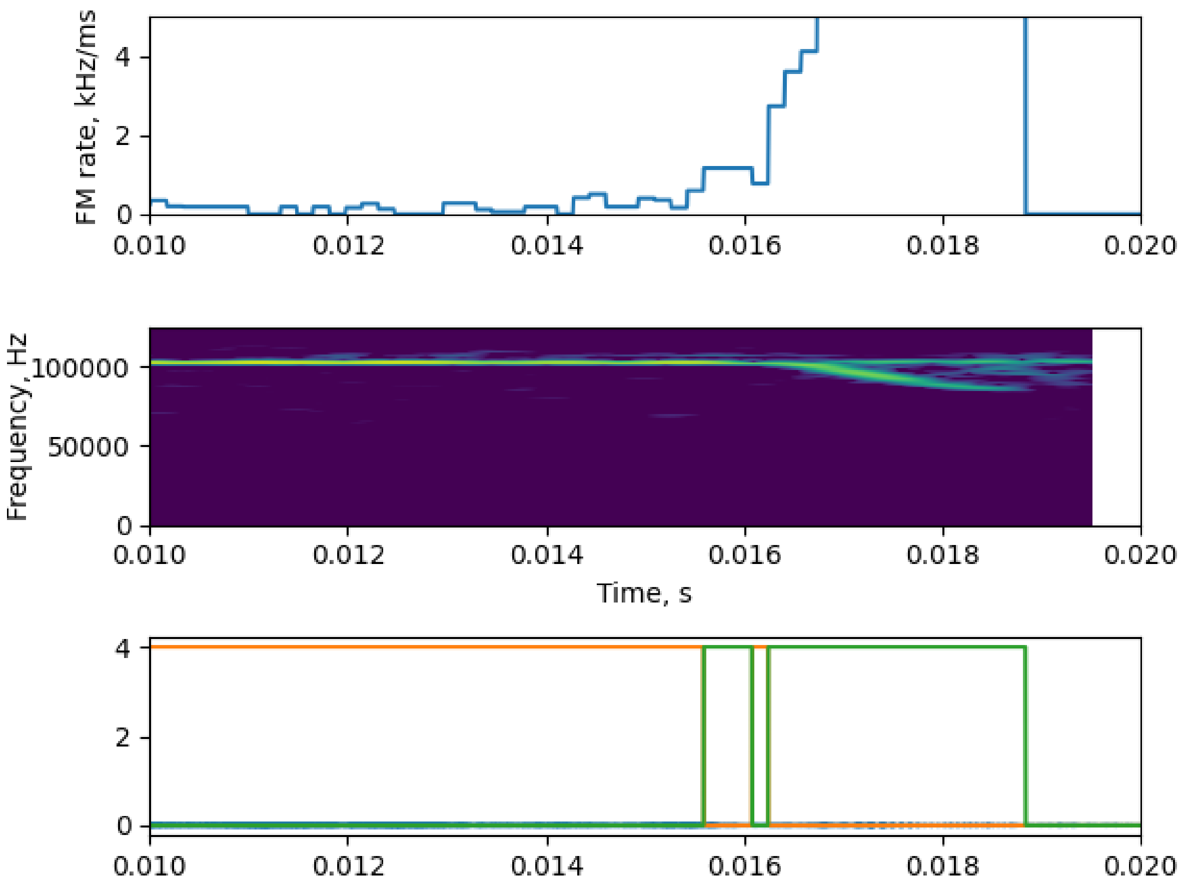


Out:

```
(<AxesSubplot:>, <AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>)
```

Something's odd – even though the FM rate seems to be close to zero near the actual FM parts, parts of it are still being classified as FM!! What's happening. Let's take a closer look at the FM rate profile, but zoom in so the y-axis is more limited. Let's also overlay the CF-FM segmentation results over this.

```
seg_out, call_parts, msmts = outputs_refined
cf, fm, info = seg_out

w,s,a = out_refined_inspector.visualise_fmrate()
w.set_ylim(0,5)
t_min, t_max = 0.01, 0.02
w.set_xlim(t_min, t_max)
s.set_xlim(t_min, t_max)
a.set_xlim(t_min, t_max)
w.plot()
itsfm.make_waveform(cf*4,fs)
itsfm.make_waveform(fm*4,fs)
plt.tight_layout()
```

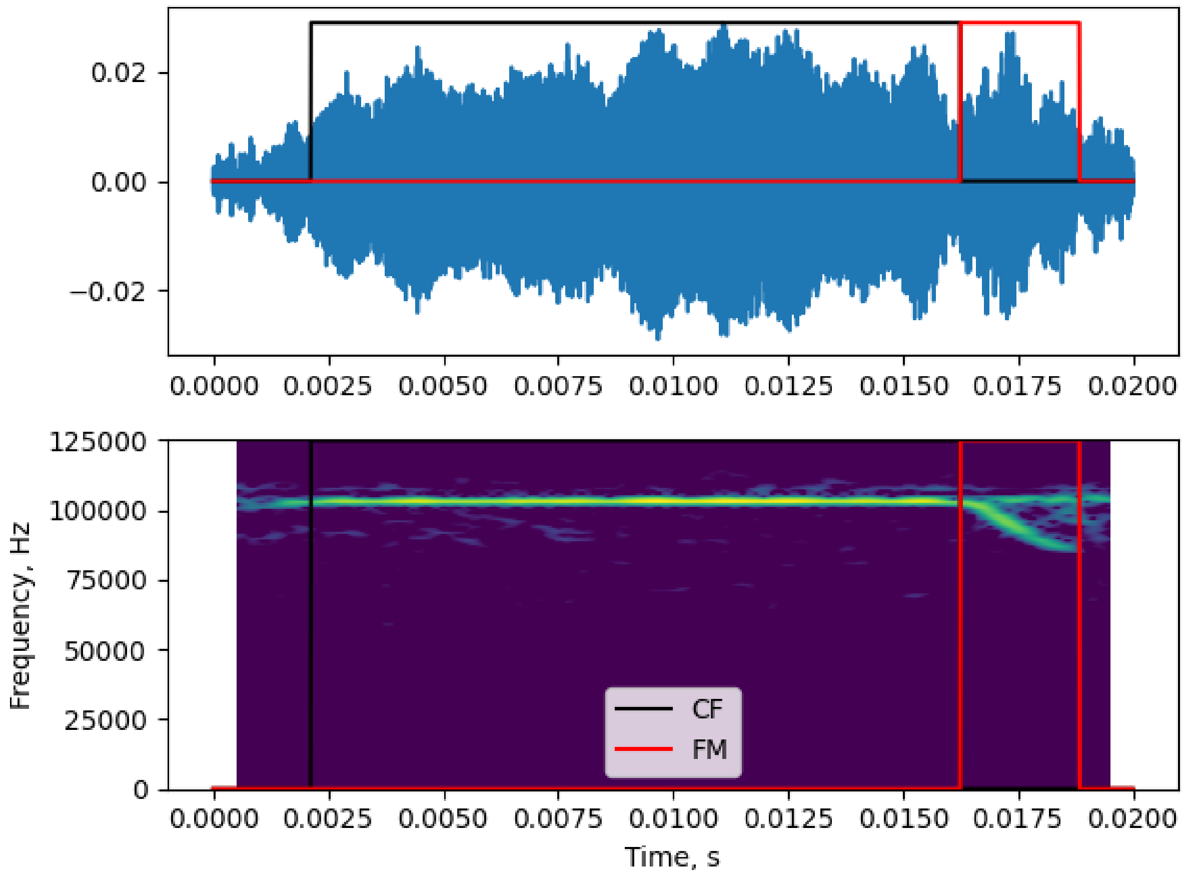


From this you can clearly see that the FM part correspond to tiny peaks in the *fmrate* which reach around 1 kHz/ms. It may of course be no surprise once you know the default *fmrate_threshold* is 1 kHz/ms. This rate doesn' make sense for bat call FM portions as they have much high frequency modulation rates. The easy way to estimate the relevant *fmrate_threshold* is to eyeball the start and end frequencies of a call part and calculate the fm rate!

### Step 3: Set a relevant *fmrate_threshold*

For this example call any FM rate above 0.5kHz/ms will allow a sensible segmentation of the CF and FM parts. Lets set it more conservatively at 2kHz/ms, this will reduce false positives. In general, for this particular call type, the FM sweep has an approximate rate of 5-6kHz/ms, and so we should definitely be able to pick up the FM region with a threshold of 2kHz/ms.

```
# add an additional keyword argument
keywords['fmrate_threshold'] = 2.0 # kHz/ms

output_newfmr = itsfm.segment_and_measure_call(audio, fs,**keywords)

out_newfmr_insp = itsfm.itsFMInspector(output_newfmr, audio, fs)
out_newfmr_insp.visualise_cffm_segmentation()
```



Out:

```
(<AxesSubplot:>, <AxesSubplot:xlabel='Time, s', ylabel='Frequency, Hz'>)
```

### Summary

This tutorial exposed some of the messy details behind the PWVD frequency tracking. In most cases, I hope you won't need to think so much about the parameter choices. However, some basic playing around will definitely be necessary each time you're handling a new type of sound or recording type. Hopefully, this has either allowed you to get a glimpse into the system. Do let me know if there's something (or everythin) is confusing, and not clear!

**Total running time of the script:** ( 0 minutes 14.127 seconds)

## 2.3 itsfm without coding

Not so comfortable coding in Python? There is an option to use *itsfm* by specifying the parameters you'd like to use for a segment and measure run. This means the sound will be segmented into FM and CF and measurements (custom/inbuilt) will be done on the detected sound parts.

### 2.3.1 Running a batch file analysis

Remember to install the package as outlined in the main page. Running a batch file analysis is as simple as typing the following command into the command line interface of your OS. *Remember to activate your conda/virtual environment if you're using one before!*

```
python -m itsfm -batchfile path_to_your_batchfile_here.csv
```

#### Outputs from a batch file analysis

1. Diagnostic plots : The batch file analysis will produce a pdf for each audio snippet processed. This pdf will have a series of diagnostic plots in each page for later inspection.
2. Measurement file : A common long-format measurement csv file will be output. Each row in the file corresponds to one segmented region and each column corresponds to the default/custom measurements run on the segments.

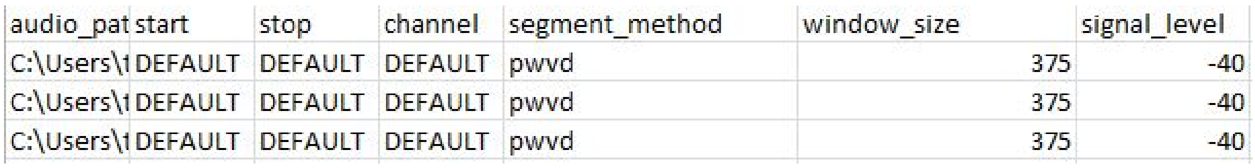
#### The batch file

The batch file is a .csv file with the following layout

The basic idea is to give the same inputs that you would use while calling the `itsfm.segment_and_measure_call` function. All non-default arguments can be input as columns in the batchfile. The names of the columns must match the keyword used in a function call.

#### A simple batch file

*Note* : if a keyword argument is not expicitly specified as a column with filled in values, the default value for this argument will be used.

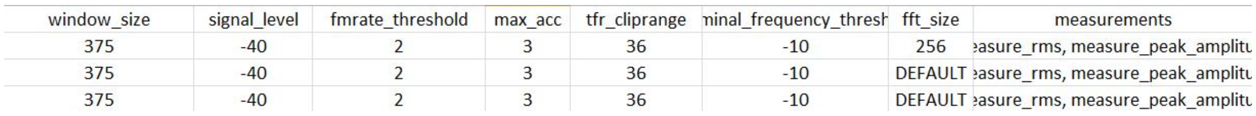


| audio_pat | start | stop | channel | segment_method | window_size | signal_level |
|---|---|---|---|---|---|---|
| C:\Users\1 | DEFAULT | DEFAULT | DEFAULT | pwvd | 375 | -40 |
| C:\Users\1 | DEFAULT | DEFAULT | DEFAULT | pwvd | 375 | -40 |
| C:\Users\1 | DEFAULT | DEFAULT | DEFAULT | pwvd | 375 | -40 |

1. `audio_path` : the path to the audio file
2. `start` : What time into the audio file should the sound be read? Defaults to 0.
3. `stop` : When does the relevant sound segment end? Defaults to the duration of the file.
4. `channel` : integer value. If file is a multichannel file, then choose relevant channel. *Note* : channel numbers start from 1 onwards.
5. `segment_method` : the CF-FM segmentation method to be used.
6. `window_size` : integer value. Number of samples to be used to calculate the moving dB rms window.
7. `signal_level` : float <=0. The value in dB rms (re 1) that defines a region of analysable signal.

### A batch file is extensible

Depending on the extent of control you'd like to have on the analysis, you can add more arguments to control the output. For instance, take a look at the batch file below. This shows an extension of the previous batch file. In this particular batch file there are a whole bunch of other



| window_size | signal_level | fmrate_threshold | max_acc | tfr_cliprange | ninal_frequency_thresh | fft_size | measurements |
|---|---|---|---|---|---|---|---|
| 375 | -40 | 2 | 3 | 36 | -10 | 256 | easure_rms, measure_peak_amplitu |
| 375 | -40 | 2 | 3 | 36 | -10 | DEFAULT | easure_rms, measure_peak_amplitu |
| 375 | -40 | 2 | 3 | 36 | -10 | DEFAULT | easure_rms, measure_peak_amplitu |

- `fmrate_threshold` : float>0. The fm rate above which a region is consdered FM in kHz/ms.
- `max_acc` : float>0. The maximum *acceleration* that is allowed in the frequency profile. The acceleration is a proxy for how rough or spiky the frequency profile in a particular region. Values closer to 0 are better.
- `tfr_cliprange` : float>0. The maximum dynamic range allowed in a time-frequency representation in dB. See *itsfm.frequency_tracking.generate_pwvd_frequency_profile*
- `fft_size` : int>0. The number of FFT samples used to generate spectrograms in the final visualisations.
- `measurements` : str. accepts a simple list with comma separated inbuilt function names. The supported inbuilt measurement functions can be be seen by typing `help(itsfm.measurement_functions)`

### Each row is independent

It is possible to use a combination of default and non-default values. Whether doing so is advisable or not is a situation-based call. For instance, in the extended batch file above, a non-default `fft_size` is used for the first file, and the other files have above have a default value.

### Skip a row

There may be times when the raw data is truly bad (or empty, or missing) and you want to skip a particular row in the batchfile. This can be done by add a 'skip' column, and adding `True` in that particular row. Remember to fill out the rest of the rows with `DEFAULT`.

### Run only a single row

To quickly test which parameters work best, you can also just run single examples by using the `one_row` argument. This approach allows you to troubleshoot a single problematic audio clip and quickly change the parameters for that file until it makes sense or works. The example below will run the 11th row in the batchfile.

```
$ python -m itsfm -batchfile template_batchfile.csv -one_row 10
```

### Running parts of a batchfile

Stuff happens and an analysis run can stop anytime as it runs throug the batchfile because some of the parameters don't make sense. To continue from a desired row or run only a selected set of rows you can use the `-from` and `-till` arguments.

```
$ python -m itsfm -batchfile template_batchfile.csv -from 10
```

The example above will run the analysis from the 11th row and proceed till the last row of the batchfile.

```
$ python -m itsfm -batchfile template_batchfile.csv -till 10
```

The example above will run the analysis from the 1st till 11th row and proceed till the last row of the batchfile.

```
$ python -m itsfm -batchfile template_batchfile.csv -from 5 -till 10
```

The example above runs *itsfm* analysis from the 6th-11th rows of a batchfile.

#### Measurement file already exists

It is very likely that you may get this error message on trying to run a batchfile after the first run:

```
$ ValueError: The file: measurements_basic_batchfile.csv already exists- please move␣
↪it elsewhere or rename it!
```

This is because only one measurement file is allowed to be there in the folder where batchfile processing is being done. This feature prevents the accidental overwriting of results! To prevent this error from appearing again, delete, rename or move the current measurements file.

### 2.3.2 Suppressing the '..already exists' error

It can be irritating to encounter the '. . . already exists' error while trying to maintain a fast back and forth between results and parameter values. To prevent this error from happening - just use the `-del_measurement` argument. Set it to `True` and any file starting with *measurement* will be deleted before the actual *itsfm* run.

*Warning* : use this being aware that this involves file deletion! It's fine if you plan to run the whole batchfile at one stretch later anyway.

```
$ python -m itsfm -batchfile template_batchfile.csv -batchfile yourbatchfilehere.csv -
↪del_measurement True
```

#### Which argument/s can be specified?

The exact arguments that can be specified depend on which level you'd like to apply control, and therefore the relevant functions need to be looked up. For instance, if I wanted to make sure the frequency profile of a sound was sampled every 1ms to generate the FM rate profile. I'd look up the `itsfm.segment.whole_audio_fmrate` source code to find the *sample_every* optional argument. A column names *sample_every* will allow the custom definition of a downsampling intensity for that row. In most cases the approach aligned above should work, especially if the parameter value is a float. Results may vary if the type of the csv file cell entry are mis-interpreted.

## 2.4 Accuracy Reports

This page has a collection of examples which illustrate the accuracy to which *itsfm* does the different things its supposed to do. As of now I've only added the accuracy report for CF-FM calls. Do make a pull request with accuracy reports for your sounds of interest!

### 2.4.1 CF-FM call segmentation accuracy

This page will illustrate the accuracy with which *itsfm* can segment CF-FM parts of a CF-FM call. To see what a CF-FM call looks like check out the bat-call example in the 'Basic Examples' page.

The synthetic data has already been generated and run with the `segment_and_measure` function, and now we'll compare the accuracy with which it has all happened.

A CF-FM bat call typically has three parts to it, 1) an 'up' FM, where the frequency of the call increases, 2) a 'CF' part, where the frequency is stable, and then 3) a 'down' FM, where the frequency drops. The synthetic data is basically a set of CF-FM calls with a combination of upFM, downFM and CF part durations, bandwidths,etc.

Here we will only be seeing if the durations of each of the segment parts have been picked up properly or not. We will *not* be performing any accuracy assessments on the exact parameters (eg. peak frequency, rms, etc) because it is assumed that if the call parts can be identified by their durations then the measurements will in turn be as expected.

There is no silence in the synthetic calls, and no noise too. This is the situation which should provide the highest accuracy.

#### What happened before

To see more on the details of the generation and running of the synthetic data see the modules *CF/FM call segmentation* and *Generating the CF-FM synthetic calls*

```
import itsfm
import matplotlib.pyplot as plt
plt.rcParams['agg.path.chunksize'] = 10000
import numpy as np
import pandas as pd
import seaborn as sns
import tqdm

obtained = pd.read_csv('obtained_pwvd_horseshoe_sim.csv')
synthesised = pd.read_csv('horseshoe_test_parameters.csv')
```

Let's look at the obtained regions and their durations

```
obtained
```

We can see the output has each CF/FM region labelled by the order in which they're found. Let's re-label these to match the names of the synthesised call parameter dataframe. 'upfm' is fm1, 'downfm' is fm2.

```
obtained.columns = ['call_number','cf_duration',
                    'upfm_duration', 'downfm_duration', 'other']
```

Let's look at the synthetic call parameters. There's a bunch of parameters that're not interesting for this accuracy exercise and so let's remove them

```
synthesised

synthesised.columns

synth_regions = synthesised.loc[:,['cf_duration', 'upfm_duration','downfm_duration']]
synth_regions['other'] = np.nan
synth_regions['call_number'] = obtained['call_number']
```

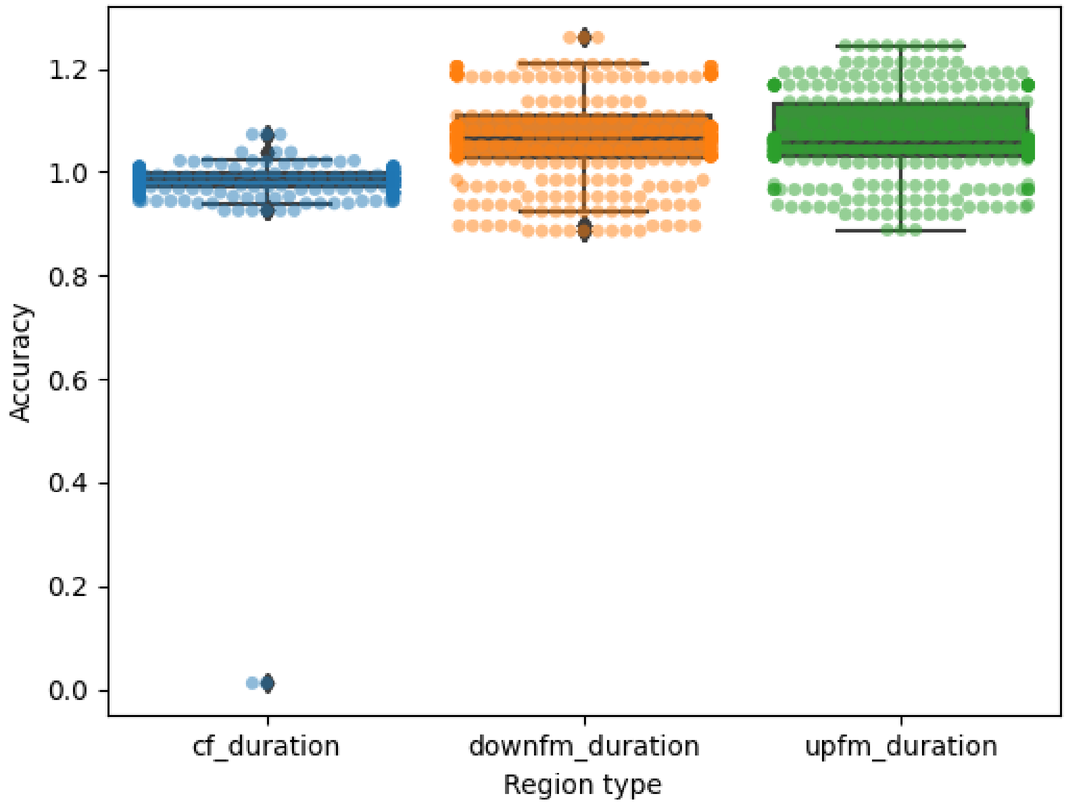
### Comparing the synthetic and the obtained results

We have the two datasets formatted properly, now let's compare the accuracy of *itsfm*.

```
accuracy = obtained/synth_regions
accuracy['call_number'] = obtained['call_number']
```

Overall accuracy of segmentation:

```
accuracy_reformat = accuracy.melt(id_vars=['call_number'],
                                            var_name='Region type',
                                            value_name='Accuracy')

accuracy_reformat = accuracy_reformat[accuracy_reformat['Region type']!='other']

plt.figure()

ax = sns.boxplot(x='Region type', y = 'Accuracy',
                         data=accuracy_reformat)

ax = sns.swarmplot(x='Region type', y = 'Accuracy',
                         data=accuracy_reformat,
                         alpha=0.5)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 75.3% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 46.0% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 40.7% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```

### Peak-percentage method accuracy

Now let's take a look at the peak percentage method's accuracy

```
obtained_pkpct = pd.read_csv('obtained_pkpct_horseshoe_sim.csv')

obtained_pkpct.head()
```

It can clearly be seen that there are some calls with *multiple* segments detected. This multiplicity of segments typically results from false positive detections, where the CF-FM ratio jumps above 0 spuriously for a few samples. Let's take a look at some of these situations.

```
def identify_valid_segmentations(df):
    '''
    Identifies if a segmentation output has valid (numeric)
    entries for cf1, fm1, fm2, and NaN for all other columns.

    Parameters
    ----------
    df : pd.DataFrame
        with at least the following column names, 'cf1','fm1','fm2'

    Returns
    -------
    valid_segmentation: bool.
        True, if the segmentation is valid.
    '''
    all_columns = df.columns
    target_columns = ['cf1','fm1','fm2']
    rest_columns = set(all_columns)-set(target_columns)
    rest_columns = rest_columns - set(['call_number'])

    valid_cf1fm1fm2 = lambda row, target_columns: np.all([ ~np.isnan(row[each]) for␣
↪each in target_columns])
    all_otherrows_nan = lambda row, rest_columns: np.all([ np.isnan(row[each]) for␣
↪each in rest_columns])

    all_valid_rows = np.zeros(df.shape[0],dtype=bool)
    for i, row in df.iterrows():
        all_valid_rows[i] = np.all([valid_cf1fm1fm2(row, target_columns),
                                    all_otherrows_nan(row, rest_columns)])
    return all_valid_rows
```

(continues on next page)

(continued from previous page)

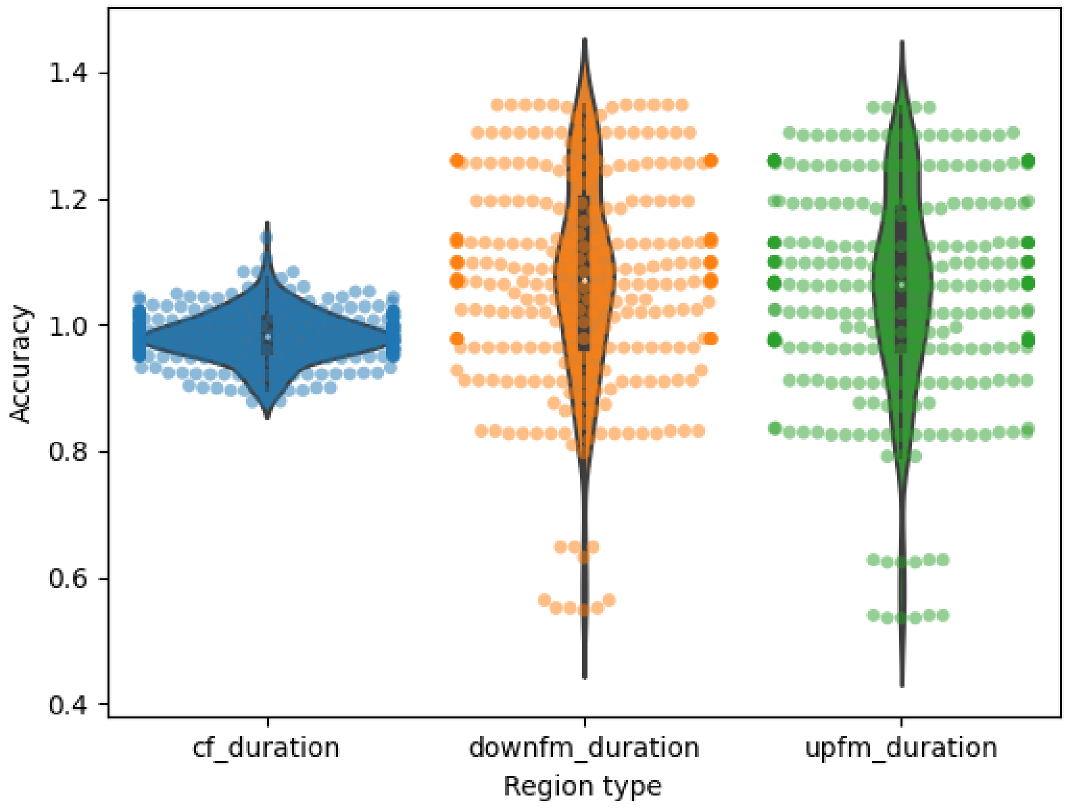
```
calls_w_3segs = identify_valid_segmentations(obtained_pkpct)

print(f'{np.sum(calls_w_3segs)/calls_w_3segs.size} % of calls have 3 segments')
```

Out:

```
0.9444444444444444 % of calls have 3 segments
```

6% of calls don't have 3 components - let's remove these poorly segmented calls and quantify their segmentation accuracy.

```
pkpct_well_segmented = obtained_pkpct.loc[calls_w_3segs,:]
pkpct_well_segmented = pkpct_well_segmented.drop(['cf2','fm3','fm4'],axis=1)

pkpct_well_segmented.columns = ['call_number','cf_duration',
                    'upfm_duration', 'downfm_duration', 'other']

pkpct_accuracy = pkpct_well_segmented/synth_regions.loc[calls_w_3segs,:]


# Overall accuracy of segmentation:
pkpct_accuracy_reformat = pkpct_accuracy.melt(id_vars=['call_number'],
                                            var_name='Region type',
                                            value_name='Accuracy')
pkpct_accuracy_reformat = pkpct_accuracy_reformat[pkpct_accuracy_reformat['Region type
↪']!='other']

plt.figure()
ax = sns.violinplot(x='Region type', y = 'Accuracy',
                         data=pkpct_accuracy_reformat)

ax = sns.swarmplot(x='Region type', y = 'Accuracy',
                         data=pkpct_accuracy_reformat,
                         alpha=0.5)
```

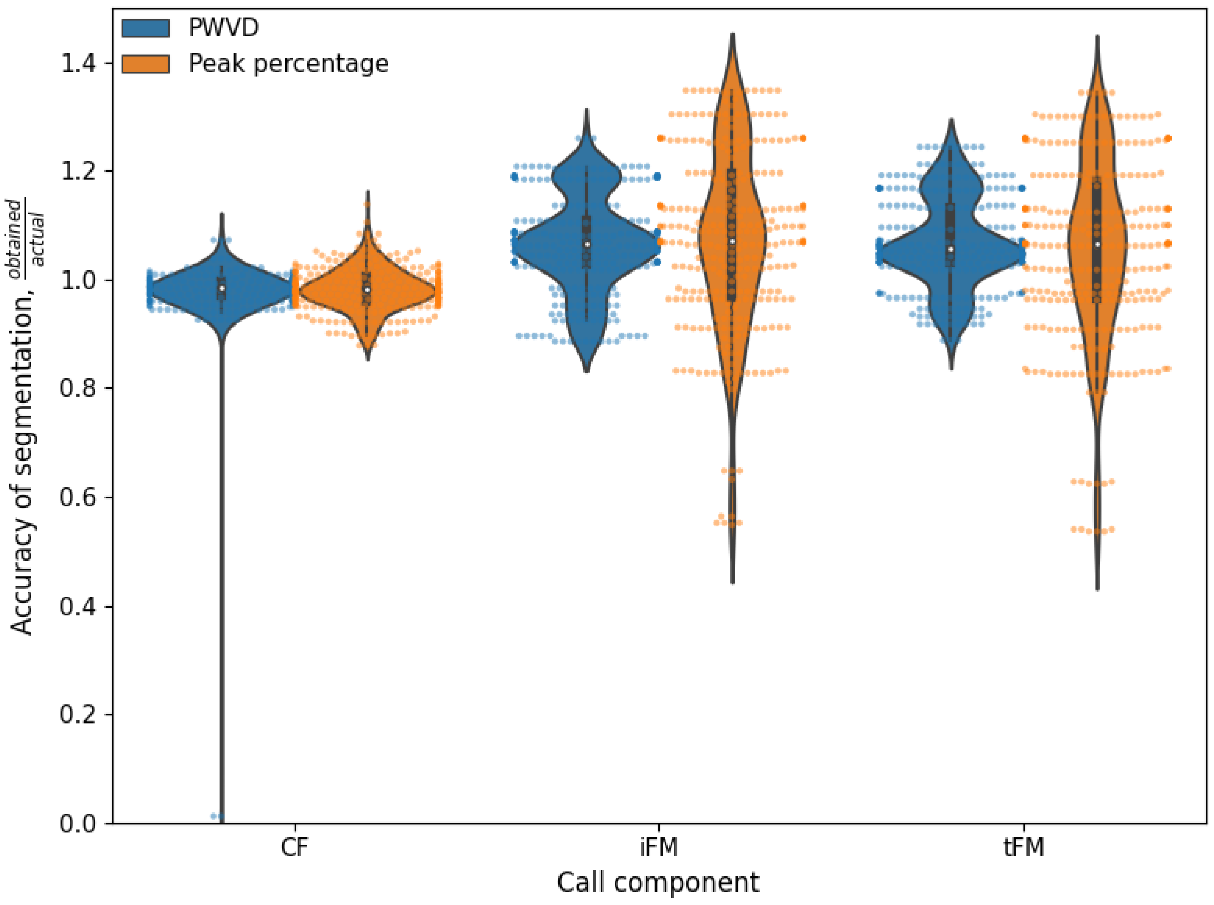


Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 51.6% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 19.0% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 26.8% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```

### Putting it all together: PWVD vs peak percentage

```
pwvd_accuracy = accuracy_reformat.copy()
pwvd_accuracy['method'] = 'pwvd'

pkpct_accuracy = pkpct_accuracy_reformat.copy()
pkpct_accuracy['method'] = 'pkpct'


both_accuracy = pd.concat([pwvd_accuracy, pkpct_accuracy])
both_accuracy['combined_id'] = both_accuracy['Region type']+both_accuracy['method']


grouped_accuracy = both_accuracy.groupby(['Region type','method'])

plt.figure(figsize=(8,6))
ax = sns.swarmplot(x='Region type', y = 'Accuracy',
                         data=both_accuracy, hue='method',hue_order=["pwvd", "pkpct"],
                         dodge=True,alpha=0.5, s=3)

ax2 = sns.violinplot(x='Region type', y = 'Accuracy',
                         data=both_accuracy, hue='method',hue_order=["pwvd", "pkpct"],
                         dodge=True,alpha=0.5, s=2.5)
ax2.legend_.remove()
handles, labels = ax2.get_legend_handles_labels() # thanks Ffisegydd@ https://
↪stackoverflow.com/a/35539098
l = plt.legend(handles[0:2], ['PWVD','Peak percentage'], loc=2, fontsize=11,
               borderaxespad=0., frameon=False)

plt.xticks([0,1,2],['CF','iFM','tFM'], fontsize=11)
plt.xlabel('Call component',fontsize=12);plt.ylabel('Accuracy of segmentation, $\\frac
↪{obtained}{actual}$',fontsize=12);
plt.yticks(fontsize=11)
plt.ylim(0,1.5)
plt.tight_layout()
plt.savefig('pwvd-pkpct-comparison.png')
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 60.5% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 37.3% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 32.1% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 11.1% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 31.2% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 19.6% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```

What are the 95%ile limits of the accuracy?

```
accuracy_ranges = grouped_accuracy.apply(lambda X: np.nanpercentile(X['Accuracy'],[2.
↪5,97.5]))
accuracy_ranges
```

Out:

```
Region type      method
cf_duration      pkpct                  [0.8994, 1.05945]
                 pwvd      [0.9392, 1.0218500000000001]
downfm_duration  pkpct                     [0.648, 1.348]
                 pwvd       [0.8867499999999999, 1.208]
upfm_duration    pkpct      [0.624, 1.2999999999999998]
                 pwvd      [0.9179999999999999, 1.2416]
dtype: object
```

### Troubleshooting the 'bad' fixes - what went wrong?

#### Some bad PWVD identifications

As we can see there are a few regions where the accuracy is very low, let's investigate which of these calls are doing badly.

```
poor_msmts = accuracy[accuracy['cf_duration']<0.5].index
```

Now, let's troubleshooot this particular set of poor measurements fully.

```
simcall_params = pd.read_csv('horseshoe_test_parameters.csv')
obtained_params = pd.read_csv('obtained_pwvd_horseshoe_sim.csv')

obtained_params.loc[poor_msmts,:]
```

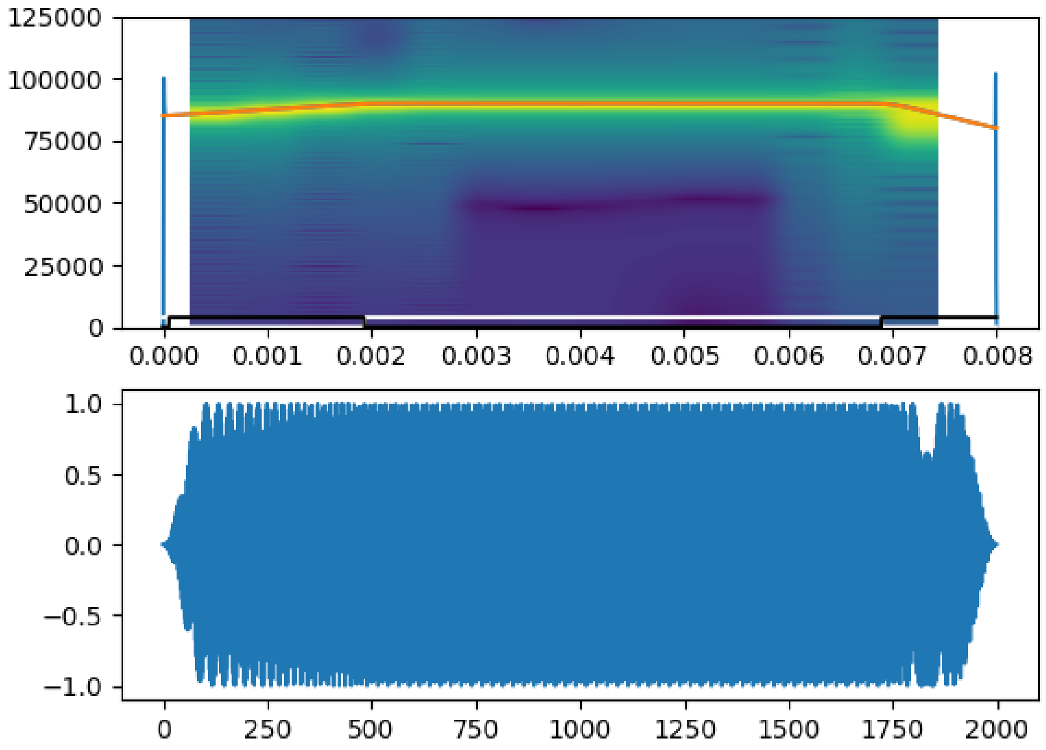
There are two CF regions being recognised, one of them is just extremely short. Where is this coming from? Let's take a look at the actual frequency tracking output, by re-running the `itsfm` routine once more:

```
import h5py


f = h5py.File('horseshoe_test.hdf5', 'r')

fs = float(f['fs'][:])

parameters = {}
parameters['segment_method'] = 'pwvd'
parameters['window_size'] = int(fs*0.001)
parameters['fmrate_threshold'] = 2.0
parameters['max_acc'] = 10
parameters['extrap_window'] = 75*10**-6

raw_audio = {}

for call_num in tqdm.tqdm(poor_msmts.to_list()):
    synthetic_call = f[str(call_num)][:]
    raw_audio[str(call_num)] = synthetic_call
    output = itsfm.segment_and_measure_call(synthetic_call, fs, **parameters)
```

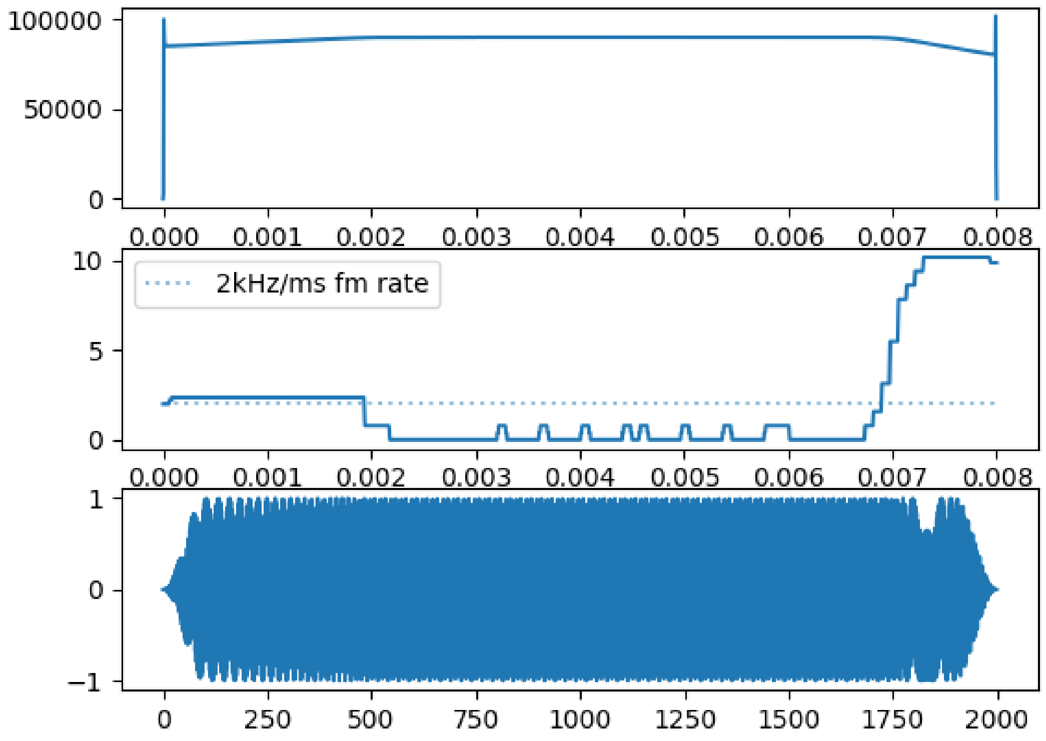
(continues on next page)

(continued from previous page)

```
    seg_output, call_parts, measurements= output

    # # save the long format output into a wide format output to
    # # allow comparison
    # sub = measurements[['region_id', 'duration']]
    # sub['call_number'] = call_num
    # region_durations = sub.pivot(index='call_number',
    #                               columns='region_id', values='duration')
    # obtained.append(region_durations)

f.close()

call_num = str(poor_msmts[0])

plt.figure()
plt.subplot(211)
plt.specgram(raw_audio[call_num], Fs=fs)
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[2]['raw_fp'])
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[2]['fitted_fp'])
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[0]*4000,'w')
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[1]*4000,'k')
plt.subplot(212)
plt.plot(raw_audio[call_num])

plt.figure()
plt.subplot(311)
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[2]['raw_fp'])
plt.subplot(312)
plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
         seg_output[2]['fmrate'])
#plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
#          seg_output[0]*5,'k',label='CF')
#plt.plot(np.linspace(0,raw_audio[call_num].size/fs,raw_audio[call_num].size),
#          seg_output[1]*5,'r', label='FM')
plt.hlines(2, 0, raw_audio[call_num].size/fs, linestyle='dotted', alpha=0.5,
           label='2kHz/ms fm rate')
plt.legend()
plt.subplot(313)
plt.plot(raw_audio[call_num])
```


125000
100000
75000
50000
25000
0
0.000
0.001
0.002
0.003
0.004
0.005
0.006
0.007
0.008
1.0
0.5
0.0
−0.5
−1.0
0
250
500
750
1000
1250
1500
1750
2000

•



•

Out:

```
  0%|          | 0/2 [00:00<?, ?it/s]
 50%|#####     | 1/2 [00:01<00:01,  1.01s/it]
100%|##########| 2/2 [00:02<00:00,  1.01s/it]
100%|##########| 2/2 [00:02<00:00,  1.01s/it]

[<matplotlib.lines.Line2D object at 0x7f40e96ac190>]
```

### Making some corrections to the PWVD output

Here, we can see that the 'error' is that the FM rate is very slightly below the 2 kHz/ms FM rate, and thus appears as a false CF region. This slight drop in FM rate is also because of edge effects. The frequency profile correction methods in place were able to recognise the odd spike in frequency profile and interpolate between two regions with reliable frequency profiles. This interpolation thus lead to a slight drop in the FM rate.

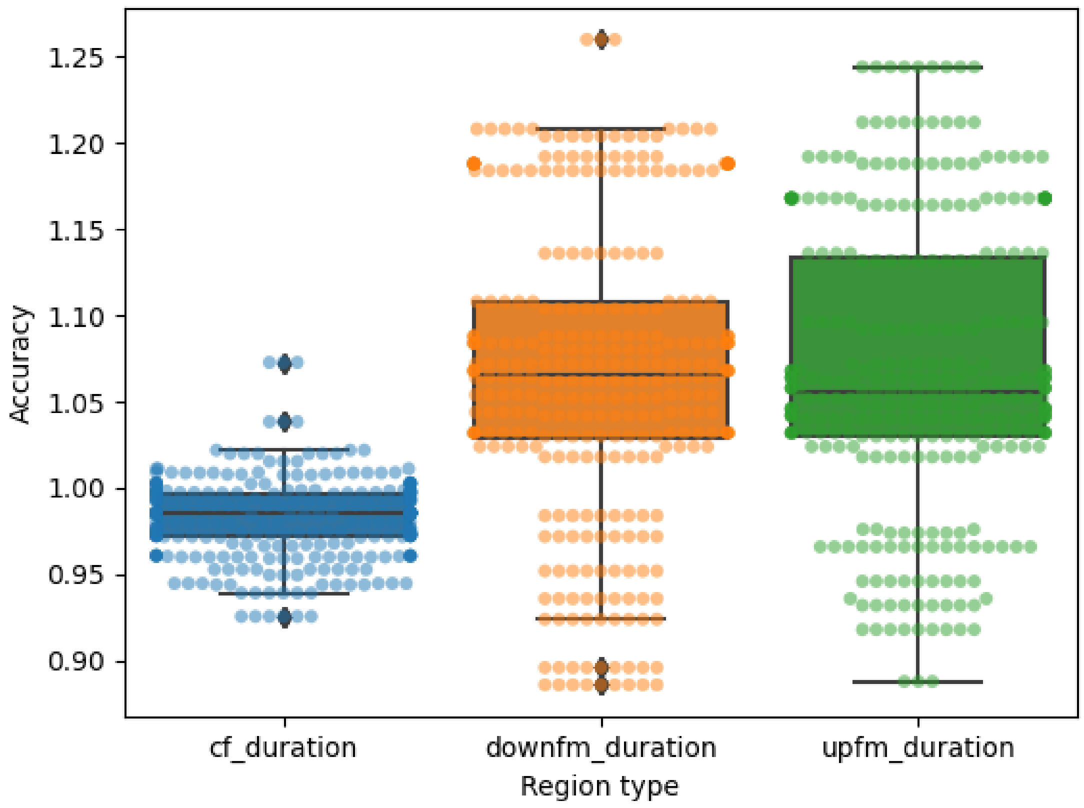
Considering that the CF measurement is actually there, but labelled as CF2, let's correct this labelling error and then see the final accuracy. We will not attempt to compensate for this error by adjusting the iFM duration here.

```
corrected_obtained = obtained_params.copy()
for each in poor_msmts:
    corrected_obtained.loc[each,'cf1'] = corrected_obtained.loc[each,'cf2']
    corrected_obtained.loc[each,'other'] = np.nan

corrected_obtained = corrected_obtained.loc[:,corrected_obtained.columns!='cf2']
```

(continues on next page)

(continued from previous page)

```
corrected_obtained.columns = ['call_number','cf_duration',
                    'upfm_duration', 'downfm_duration', 'other']

corrected_accuracy = corrected_obtained/synth_regions
corrected_accuracy['call_number'] = corrected_obtained['call_number']
corrected_accuracy_reformat = corrected_accuracy.melt(id_vars=['call_number'],
                                            var_name='Region type',
                                            value_name='Accuracy')
corrected_accuracy_reformat = corrected_accuracy_reformat.loc[corrected_accuracy_
↪reformat['Region type']!='other',:]

plt.figure()
ax = sns.boxplot(x='Region type', y = 'Accuracy',
                         data=corrected_accuracy_reformat)

ax = sns.swarmplot(x='Region type', y = 'Accuracy',
                         data=corrected_accuracy_reformat,
                         alpha=0.5)
```



Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 47.8% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
```

(continues on next page)

(continued from previous page)

```
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 14.5% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/python3.7/site-
↪packages/seaborn/categorical.py:1296: UserWarning: 19.8% of the points cannot be␣
↪placed; you may want to decrease the size of the markers or use stripplot.
  warnings.warn(msg, UserWarning)
```

**Total running time of the script:** ( 0 minutes 8.912 seconds)

## 2.4.2 Running CF-FM call segmentation

Here we will run the `segment_and_measure` function and store the results of how long each Cf/FM segment is.

### Dataset creation

The synthetic dataset has already been created in a separate module. See 'Generating the CF-FM synthetic calls' in the main page.

### It can take long

We're running a few hundred synthetic audio clips with a few seconds (1-10s) needed per iteration. This could mean, it might take a while(5,10 or more minutes)!

```
import h5py
import itsfm
import pandas as pd
from tqdm import tqdm
```

Now, let's load each synthetic call and proceed to save the results from the PWVD and peak-percentage based methods.

### FM rate based segmentation

```
obtained = []

f = h5py.File('horseshoe_test.hdf5', 'r')
synthesised = pd.read_csv('horseshoe_test_parameters.csv')


fs = float(f['fs'][:])

parameters = {}
parameters['segment_method'] = 'pwvd'
parameters['window_size'] = int(fs*0.001)
parameters['fmrate_threshold'] = 2.0
parameters['max_acc'] = 10
parameters['extrap_window'] = 75*10**-6

for call_num in tqdm(range(synthesised.shape[0])):
```

(continues on next page)

(continued from previous page)

```
    synthetic_call = f[str(call_num)][:]
    output = itsfm.segment_and_measure_call(synthetic_call, fs, **parameters)

    seg_output, call_parts, measurements= output
    # save the long format output into a wide format output to
    # allow comparison
    sub = measurements[['region_id', 'duration']]
    sub['call_number'] = call_num
    region_durations = sub.pivot(index='call_number',
                                 columns='region_id', values='duration')
    obtained.append(region_durations)

all_obtained = pd.concat(obtained)

all_obtained.to_csv('obtained_pwvd_horseshoe_sim.csv')
```

### Peak-percentage based segmentation

```
pkpctg_parameters = {}
pkpctg_parameters['segment_method'] = 'peak_percentage'
pkpctg_parameters['peak_percentage'] = 0.99
pkpctg_parameters['window_size'] = 125
pkpctg_parameters['double_pass'] = True

pkpct_obtained = []

for call_num in tqdm(range(synthesised.shape[0])):
    synthetic_call = f[str(call_num)][:]
    output = itsfm.segment_and_measure_call(synthetic_call, fs, **pkpctg_parameters)

    seg_output, call_parts, measurements= output
    # save the long format output into a wide format output to
    # allow comparison
    sub = measurements[['region_id', 'duration']]
    sub['call_number'] = call_num
    region_durations = sub.pivot(index='call_number',
                                 columns='region_id', values='duration')
    pkpct_obtained.append(region_durations)


f.close()

pk_pctage = pd.concat(pkpct_obtained)

pk_pctage.to_csv('obtained_pkpct_horseshoe_sim.csv')
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

### 2.4.3 Generating the CF-FM synthetic calls

Module that creates the data for accuracy testing horseshoe bat type calls

```
import h5py
from itsfm.simulate_calls import make_cffm_call
import numpy as np
import pandas as pd
import scipy.signal as signal
from tqdm import tqdm

cf_durations = [0.005, 0.010, 0.015]
cf_peakfreq = [40000, 60000, 90000]
fm_durations = [0.001, 0.002]
fm_bw = [5000, 10000, 20000]

all_combinations = np.array(np.meshgrid(cf_peakfreq, cf_durations,
                                        fm_bw,fm_durations,
                                        np.flip(fm_bw),np.flip(fm_durations)))
all_params = all_combinations.flatten().reshape(6,-1).T

col_names = ['cf_peak_frequency', 'cf_duration',
             'upfm_bw', 'upfm_duration',
             'downfm_bw', 'downfm_duration']

parameter_space = pd.DataFrame(all_params, columns=col_names)
parameter_space['upfm_terminal_frequency'] = parameter_space['cf_peak_frequency'] -␣
↪parameter_space['upfm_bw']
parameter_space['downfm_terminal_frequency'] = parameter_space['cf_peak_frequency'] -␣
↪parameter_space['downfm_bw']

parameter_columns = ['cf_peak_frequency', 'cf_duration',
                     'upfm_terminal_frequency', 'upfm_duration',
                     'downfm_terminal_frequency', 'downfm_duration']

all_calls = {}
for row_number, parameters in tqdm(parameter_space.iterrows(),
                                   total=parameter_space.shape[0]):

    cf_peak, cf_durn, upfm_terminal, upfm_durn, downfm_terminal, downfm_durn =␣
↪parameters[parameter_columns]
    call_parameters = {'cf':(cf_peak, cf_durn),
                        'upfm':(upfm_terminal, upfm_durn),
                        'downfm':(downfm_terminal, downfm_durn),
                        }

    fs = 250*10**3 # 500kHz sampling rate
    synthetic_call, _ = make_cffm_call(call_parameters, fs)
    synthetic_call *= signal.tukey(synthetic_call.size, 0.1)
    all_calls[row_number] = synthetic_call


# now save the data into an hdf5 file
with h5py.File('horseshoe_test.hdf5','w') as f:
    for index, audio in all_calls.items():
        f.create_dataset(str(index), data=audio)
    f.create_dataset('fs', data=np.array([fs]))
parameter_space.to_csv('horseshoe_test_parameters.csv')
```

**Total running time of the script:** ( 0 minutes 0.000 seconds)

CHAPTER

THREE

# WHAT THE PACKAGE *DOES*:

1. Identify sounds as being constant frequency or frequency modulated
2. The ‘pwvd’ segmentation method allows a sample-level frequency estimation, the ‘frequency profile’ of the sound
3. Generates an FM rate profile over the sound
4. Performs *basic* outlier detection

CHAPTER

FOUR

# WHAT THE PACKAGE *DOES NOT*:

1. Perform any kind of pattern detection/classification. The frequency profile of a sound is generated using a percentile based threshold on each slice of the underlying Pseudo Wigner-Ville distribution.
2. Handle complex and reverberant sounds. Sounds that are multi-component, ie, with multiple harmonics or with variation in intensity of harmonics across the recording won't fare very well.
3. Separate overlapping sounds

CHAPTER

FIVE

# INSTALLATION

This is a pre-PyPi version of the package. The easiest way to install the package is to head to this page, and download/clone the repository. Go into the downloaded folder and type `python setup.py install`.

CHAPTER

SIX

# WHAT THE PACKAGE COULD DO WITH (FUTURE FEATURE IDEAS):

1. A sensible way to deal with edges of the signals. Right now the instantaneous frequencies suffer from spikes caused by bad instantaneous frequency estimates at the edges in the pseudo-wigner ville distribution method.
2. Informed frequency tracking (eg. Viterbi path or similar) in multi-harmonic sounds. Right now the frequency profile of a sound is selected by independently choosing the first peak in the time-frequency slice. This prevents a sensible tracking of frequency because even slight variations in harmonic intensities over a sound can cause the peak frequency to jump almost an octave sometimes!
3. More time-frequency representation implementations and the signal cleaning methods associated with them.

CHAPTER

SEVEN

# WHY IS EVERYTHING IN THIS CODEBASE A FUNCTION? HAVE YOU HEARD OF CLASSES?

This is the author's first Python package, and the author admits it may not be the most elegant implementation. The author's previous experience (or lack thereof) working with classes may have left some bad memories :P.However the author also admits that many things in the package might have been less cumbersome with the use of classes, and plans to implement it in due time.

CHAPTER

EIGHT

# WHERE TO GET HELP

## 8.1 Common Errors

Here are the most common errors and the probable causes for them. When I use the word 'bad' here, I mean it in the sense of *bad* for that particular signal! Especially while analysing bioacoustic recordings, a parameter value that works for one recording may not necessarily work for another one!

### 8.1.1 1. Bad *signal_level*

```
$ ValueError: No regions above signal level found!
```

Easy, reduce the *signal_level* and try again.

### 8.1.2 2. Bad *signal_level*

```
$ IndexError: only integers, slices (`:`), ellipsis (`...`), numpy.newaxis (`None`)␣
↪and integer or boolean arrays are valid indices
```

This region is caused by a very small region of the signal being selected. The PWVD transform works by choosing a small window of samples on the left and right of the current sample. If the region above *signal_level* is very small, and not greater than this small window of samples this error is raised. By default, the *isfm* window size is set to the numebr of samples corresponding to 1ms.

Alter *signal_level* or *window_size* to get a more continuous moving dB rms profile. See below also.

### 8.1.3 3. Bad *signal_level* or *window_size*

```
$ ValueError: Shape of array too small to calculate a numerical gradient, at least␣
↪(edge_order + 1) elements are required
```

The actual signal in an audio file is detected by the segment of audio that's above a user-defined *signal_level*. When the *signal_level* is set poorly or results in very short chunks of audio (<3 samples), then typically this error is thrown:

This means there's a very short audio segment that's above the *signal_level*. This typically happens because the moving dB rms profile is too spiky, which means the signal level fluctuates very quickly above and below the threshold. The new *signal_level* is best re-set after inspecting the moving dB rms profile.

The two options to fix this error are:

1. increase *window_size* to get a smoother moving dB rms profile

2. set a new *signal_level* which will make sure the moving dB rms profile is above it and matches the duration of the original signal

### 8.1.4 4. Bad *signal_level* or *window_size*

The FM rate profile of a sound is calculated by down-sampling the cleaned frequency profile. The down-sampling is done by taking a sample every now and then as defined byt the inter-sample duration. The inter-sample duration typically defaults to 1 percent of the frequency profiles length. When a bad signal level is given, there can be very short audio segments that are detected, and thus when the FM rate needs to be calculated, things break because 1% of an already very short sound may be less than the inter-sample duration itself – and therefore this message.

```
$ ValueError: The suggested duration 3.16e-06 is less than ␣
↪the inter-sample distance (1/fs): 4e-06
```

Alter the *signal_level* or *window_size* to get a more continuous dB rms profile of the sound.

### 8.1.5 Anomaly spans whole array

```
$ ValueError: The anomaly spans the whole array - please check again
```

“Anomalies” in the *itsfm* package are regions in the frequency profile which are particularly rough. This means the accelaration of the frequency profile has gone beyond the *max_acc* threshold value. Most of the time anomalies are small parts of the original signal. However, there may be times when an anomalous region spans the whole signal – and thus this warning.

A closer inspection of this particular audio file may reveal more.

1. Reduce the *signal_level* for this particular audio. When the *signal_level* is set too high, the frequency profile of irrelevant parts may be getting analysed, leading to odd and rough frequency profiles.

Hopefully this web page has enough information. Use the search bar to check if the error/issue you’re encountering has already been documented. Also do check the examples to see if the same error messages have been explained. If something’s not clear or there’s something not covered do write to me : thejasvib@gmail.com. I’ll try to answer within a week.

CHAPTER

NINE

# I FOUND A BUG AND/OR HAVE FIXED SOMETHING

Please raise an issue or pull request on Github

CHAPTER

TEN

# ACKNOWLEDGEMENTS

The PWVD transforms in itsFM rely on the tftb package by Jaidev Deshpande. I'd like to thank the Neetash MR, Aditya Krishna and Holger R. Goerlitz for helpful discussions that eventually lead down this path, and Diana Schoeppler for discussions that inspired the peak percentage method in this package. I'd also like to thank all the people who happily sent me example data and gave feedback whenever asked!

# CHAPTER ELEVEN

# LICENSE

## 11.1 API : The user interface

User-friendly top-level functions which allow the user to handle

1. Call-background segmentation
2. CF-FM call part segmentation
3. Measurement of CF-FM audio parts

Let’s take a look at an example where we [TO BE COMPLETED!!!]

```python
import scipy.signal as signal
from itsfm.user_interface import segment_and_measure_call
from itsfm.view_horseshoebat_call import *
from itsfm.simulate_calls import make_cffm_call

# create synthetic call
call_parameters = {'cf':(100000, 0.01),
                    'upfm':(80000, 0.002),
                    'downfm':(60000, 0.003),
                    }

fs = 500*10**3 # 500kHz sampling rate
synthetic_call, freq_profile = make_cffm_call(call_parameters, fs)
```

(continues on next page)

(continued from previous page)

```
# window and reduce overall signal level
synthetic_call *= signal.tukey(synthetic_call.size, 0.1)
synthetic_call *= 0.75

# measuring a well-selected call (without silent background)



# measuing a call with a silent background

# and add 2ms of additional background_noise of ~ -60dBrms
samples_1ms = int(fs*0.001)
final_size = synthetic_call.size + samples_1ms*2
call_with_noise = np.random.normal(0,10**(-60/20.0),final_size)
call_with_noise[samples_1ms:-samples_1ms] += synthetic_call

#

seg_and_msmts = segment_and_measure_call(call_with_noise, fs,
                                         segment_from_background=True)
call_segmentation, call_parts, measurements, backg_segment = seg_and_msmts
```

`itsfm.user_interface.`**`segment_and_measure_call`**`(`*main_call*, *fs*, *segment_from_background=False*, ***kwargs*`)`

Segments the CF and FM parts of a call and then proceeds to measure their characteristics. If required, also segments call from background.

**Parameters**

- **`main_call`** (*`np.array`*) –
- **`fs`** (*`float>0`*) – sampling rate in Hz
- **`segment_from_background`** (*`boolean`*) – Whether to segment the call in the main_call audio. Defaults to False.

**Keyword Arguments**

- **`further keyword arguments see segment_call_from_background,`** (*`For`*) –
- **`and measure_hbc_call`** (`segment_call_into_cf_fm`) –

**Returns**

- **segmentation_outputs** (*tuple*) – The outputs of segment_call_into_cf_fm in a tuple
- **call_parts_audio** (*dictionary*) – Dictionary with numbered entries. If a sound has the following order of Cf and FM: FM-CF-FM, then the keys will be ‘fm1’,’cf1’,’fm2’. The numbering is according to the chronological order.
- **measurements** (*pd.DataFrame*) – All the measurements from the FM and CF parts.

### Example

Let's simulate a call to demonstrate how the measurement+segmentation works.

```
>>> import scipy.signal  as signal
>>> from itsfm.simulate_calls import make_cffm_call
>>> call_properties = {'cf':(80000, 0.01), 'upfm':(70000, 0.002),
                       'downfm':(50000, 0.002)}
>>> fs = 500000
>>> call, profile = make_cffm_call(call_properties, fs)
>>> call *= signal.tukey(call.size, 0.1)
>>> plt.figure()
>>> plot1 = plt.subplot(211)
>>> plt.plot(profile)
>>> #segment the CF and FM parts with the default 'peak percentage' method.
>>> segm_out, call_parts, measures, _ = segment_and_measure_call(call,
                                                                 fs,
                                                                 segment_method=
↪'peak_percentage',
                                                                 peak_percentage=0.
↪999,
                                                                 window_
↪size=int(fs*0.5*10**-3))
>>> print(measures)
```

Now segment with frequency tracking implemented with the Pseudo Wigner Ville Distribution, and the set the fmrate threshold to 10 kHz/ms

```
>>> segm_out, call_parts, measures, _ = segment_and_measure_call(call,
                                                                  fs,
                                                                  segment_method=
↪'pwvd',
                                                                  fmrate_
↪threshold=10,
                                                                  medianfilter_
↪length=0.5*10**-3,
                                                                  )
>>> plt.subplot(212, sharex=plot1)
>>> plt.plot(segm_out[-1]['fmrate'])
>>> print(measures)
```

**See also:**

*itsfm.measure()*

itsfm.user_interface.**save_overview_graphs**(*all_subplots*, *analysis_name*, *file_name*, *index*, ***kwargs*)

Saves overview graphs.

**Parameters**

- **all_subplots** (*list*) – List with plt.subplot objects in them. For each figure to be saved, one subplot object is enough.
- **analysis_name** (*str*) – The name of the analysis. If this funciton is called through a batchfile, then it becomes the name of the batchfile
- **file_name** (*str*) –
- **index** (*int, optional*) – A numeric identifier for each graph. This is especially relevant for analyses driven by batch files as there may be cases where the calls are selected

from the same audio file but in different parts.

**Returns**

**Return type** None

### Notes

This function has the main side effect of saving all the input figures into a pdf file with >1 pages (one page per plot) for the user to inspect the results.

### Example

import numpy as np

# 1st plot plt.figure() a = plt.subplot(211) plt.plot([1,2,3]) b = plt.subplot(212) plt.plot([5,4,3])

#2nd plot plt.figure() c = plt.subplot(121) plt.plot(np.random.normal(0,1,100)) d = plt.subplot(122) plt.plot(np.random.normal(0,1,10))

save_overview_graphs([a,c], 'example_plots', 'example_file',0)

## 11.2 API : Segmenting sounds into CF and FM

Module that segments the horseshoebat call into FM and CF parts The primary logic of this

`itsfm.segment.`**`segment_call_into_cf_fm`**(*call*, *fs*, ***kwargs*)

Function which identifies regions into CF and FM based on the following process.

1. Candidate regions of CF and FM are first produced based on the segmentation method chosen'.

2. These candidate regions are then refined based on the user's requirements (minimum length of region, maximum number of CF/FM regions in the sound)

3. The finalised CF and FM regions are output as Boolean arrays.

**Parameters**

- **`call`** (*`np.array`*) – Audio with horseshoe bat call
- **`fs`** (*`float>0`*) – Frequency of sampling in Hz.
- **`segment_method`** (*`str, optional`*) – One of ['peak_percentage', 'pwvd', 'inst_freq']. Checkout 'See Also' for more information. Defaults to 'peak_percentage'
- **`refinement_method`** (*`function, str, optional`*) – The method used to refine the initial CF and FM candidate regions according to the different constraints and rules set by the user.

  Defaults to 'do_nothing'

**Returns**

- **cf_samples, fm_samples** (*np.array*) – Boolean numpy array showing which of the samples belong to the cf and the fm respectively.
- **info** (*dictionary*) – Post-processing information depending on the methods used.

### Example

Create a chirp in the middle of a somewhat silent recording

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> from itsfm.simulate_calls import make_fm_chirp, make_tone
>>> from itsfm.view_horseshoebat_call import plot_movingdbrms
>>> from itsfm.view_horseshoebat_call import visualise_call, make_x_time
>>> from itsfm.view_horseshoebat_call import plot_cffm_segmentation
>>> fs = 44100
>>> start_f, end_f = 1000, 10000
>>> chirp = make_fm_chirp(start_f, end_f, 0.01, fs)
>>> tone_freq = 11000
>>> tone = make_tone(tone_freq, 0.01, fs)
>>> tone_start = 30000; tone_end = tone_start+tone.size
>>> rec = np.random.normal(0,10**(-50/20), 44100)
>>> chirp_start, chirp_end = 10000, 10000 + chirp.size
>>> rec[chirp_start:chirp_end] += chirp
>>> rec[tone_start:tone_end] += tone
>>> rec /= np.max(abs(rec))
>>> actual_fp = np.zeros(rec.size)
>>> actual_fp[chirp_start:chirp_end] = np.linspace(start_f, end_f, chirp.size)
>>> actual_fp[tone_start:tone_end] = np.tile(tone_freq, tone.size)
```

Track the frequency of the recording and segment it according to frequency modulation

```
>>> cf, fm, info = segment_call_into_cf_fm(rec, fs, signal_level=-10,
                                              segment_method='pwvd',)
```

View the output and plot the segmentation results over it: >>> plot_cffm_segmentation(cf, fm, rec, fs)

**See also:**

*segment_by_peak_percentage()*, *segment_by_pwvd()*, segment_by_inst_frequency(), itsfm.refine_cfm_regions(), *refine_cf_fm_candidates()*

### Notes

The post-processing information in the object *info* depends on the method used.

**peak_percentage** [the two keys 'fm_re_cf' and 'cf_re_fm' which are the] relative dBrms profiles of FM with relation to the CF portion and vice versa

pwvd :

itsfm.segment.**refine_cf_fm_candidates**(*refinement_method*, *cf_fm_candidates*, *fs*, *info*, ***kwargs*)

Parses the refinement method, checks if its string or function and calls the relevant objects.

**Parameters**

- **refinement_method** (*str/function*) – A string from the list of inbuilt functions in the module *refine_cfm_regions* or a user-defined function. Defaults to *do_nothing*, an inbuilt function which doesn't returns the candidate Cf-fm regions without alteration.
- **cf_fm_candidates** (*list with 2 np.arrays*) – Both np.arrays need to be Boolean and of the same size as the original audio.
- **fs** (*float>0*) –

- **info** (*dictionary*) –

**Returns** **cf, fm** – Boolean arrays wher True indicates the sample is of the corresponding region.

**Return type** np.array

itsfm.segment.**segment_by_peak_percentage**(*call*, *fs*, ***kwargs*)

This is ideal for calls with one clear CF section with the CF portion being the highest frequency in the call: bat/bird CF-FM calls which have on CF and one/two sweep section.

Calculates the peak frequency of the whole call and performs low+high pass filtering at a frequency slightly lower than the peak frequency.

**Parameters**

- **call** (*np.array*) –
- **fs** (*float>0*) –
- **peak_percentage** (*0<float<1, optional*) – This is the fraction of the peak at which low and high-pass filtering happens. Defaults to 0.98.

**Returns**

- **cf_samples, fm_samples** (*np.array*) – Boolean array with True indicating that sample has been categorised as being CF and/or FM.
- **info** (*dictionary*) – With keys 'fm_re_cf' and 'cf_re_fm' indicating the relative dBrms profiles of the candidate FM regions relative to Cf and vice versa.

## Notes

This method unsuited for audio with non-uniform call envelopes. When there is high variation over the call envelope, the peak frequency is likely to be miscalculated, and thus lead to wrong segmentation.

This method is somewhat inspired by the protocol in Schoeppler et al. 2018. However, it differs in the important aspect of being done entirely in the time domain. Schoeppler et al. 2018 use a spectrogram based method to segment the CF and FM segments of H. armiger calls.

## References

[1] **Schoeppler, D., Schnitzler, H. U., & Denzinger, A. (2018).** Precise Doppler shift compensation in the hipposiderid bat, Hipposideros armiger. Scientific Reports, 8(1), 1-11.

**See also:**

*itsfm.segment.pre_process_for_segmentation()*

itsfm.segment.**segment_by_pwvd**(*call*, *fs*, ***kwargs*)

This method is technically more accurate in segmenting CF and FM portions of a sound. The Pseudo-Wigner-Ville Distribution of the input signal is generated.

**Parameters**

- **call** (*np.array*) –
- **fs** (*float>0*) –
- **fmrate_threshold** (*float >=0*) – The threshold rate of frequency modulation in kHz/ms. Beyond this value a segment of audio is considered a frequency modulated region. Defaults to 1.0 kHz/ms

**Returns**

- **cf_samples, fm_samples** (*np.array*) – Boolean array of same size as call indicating candidate CF and FM regions.
- **info** (*dictionary*) – See get_pwvd_frequency_profile for the keys it outputs in the *info* dictioanry. In addition, another key 'fmrate' is also calculated which has an np. array with the rate of frequency modulation across the signal in kHz/ms.

### Notes

This method may takes some time to run. It is computationally intensive. This method may not work very well in the presence of multiple harmonics or noise. Some basic tweaking of the optional parameters may be required.

**See also:**

`get_pwvd_frequency_profile()`

### Example

Let's create a two component call with a CF and an FM part in it >>> from itsfm.simulate_calls import make_tone, make_fm_chirp, silence >>> from itsfm.view_horseshoebat_call import plot_cffm_segmentation >>> from itsfm.view_horseshoebat_call import make_x_time >>> fs = 22100 >>> tone = make_tone(5000, 0.01, fs) >>> sweep = make_fm_chirp(1000, 6000, 0.005, fs) >>> gap = silence(0.005, fs) >>> full_call = np.concatenate((tone, gap, sweep)) >>> # reduce rms calculation window size because of low sampling rate! >>> cf, fm, info = segment_by_pwvd(full_call,

**fs,** window_size=10, signal_level=-12, sample_every=1*10**-3, extrap_length=0.1*10**-3)

```
>>> w,s = plot_cffm_segmentation(cf, fm, full_call, fs)
>>> s.plot(make_x_time(cf,fs), info['fitted_fp'])
```

`itsfm.segment.`**`whole_audio_fmrate`**(*whole_freq_profile*, *fs*, ***kwargs*)

When a recording has multiple components to it, there are silences in between. These silences/background noise portions are assigned a value of 0 Hz.

When a 'whole audio' fm rate is naively calculated by taking the diff of the whole frequency profile, there will be sudden jumps in the fm-rate due to the silent parts with 0Hz and the sound segments with non-zero segments. Despite these spikes being very short, they then propagate their influence due to the median filtering that is later down downstream. This essentially causes an increase of false positive FM segments because of the apparent high fmrate.

To overcome the issues caused by the sudden zero to non-zero transitions in frequency values, this function handles each non-zero sound segment separately, and calculates the fmrate over each sound segment independently.

**Parameters**

- **`whole_freq_profile`** (*`np.array`*) – Array with sample-level frequency values of the same size as the audio.
- **`fs`** (*`float>0`*) –

**Returns**

- **fmrate** (*np.array*) – The rate of frequency modulation in kHz/ms. Same size as *whole_freq_profile* Regions in *whole_freq_profile* with 0 frequency are set to 0kHz/ms.

- **fitted_frequency_profile** (*np.aray*) – The downsampled, smoothed version of *whole_freq_profile*, of the same size.

**Attention:** The *fmrate must* be processed further downstream! In the whole-audio *fmrate* array, all samples that were 0 frequency in the original *whole_freq_profile* are set to 0 kHz/ms!!!

**See also:**

*calculate_fm_rate()*

## Example

Let's make a synthetic multi-component sound with 2 FMs and 1 CF component.

```
>>> fs = 22100
>>> onems = int(0.001*fs)
>>> sweep1 = np.linspace(1000,2000,onems) # fmrate of 1kHz/ms
>>> tone = np.tile(3000, 2*onems) # CF part
>>> sweep2 = np.linspace(4000,10000,3*onems) # 2kHz/ms
>>> gap = np.zeros(10)
>>> freq_profile = np.concatenate((sweep1, gap, tone, gap, sweep2))
>>> fmrate, fit_freq_profile = whole_audio_fmrate(freq_profile, fs)
```

itsfm.segment.**calculate_fm_rate**(*frequency_profile*, *fs*, ***kwargs*)

A frequency profile is generally oversampled. This means that there will be many repeated values and sometimes minor drops in frequency over time. This leads to a higher FM rate than is actually there when a sample-wise diff is performed.

This method downsamples the frequency profile, fits a polynomial to it and then gets the smoothened frequency profile with unique values.

The sample-level FM rate can now be calculated reliably.

**Parameters**

- **frequency_profile** (*np.array*) – Array of same size as the original audio. Each sample has the estimated instantaneous frequency in Hz.
- **fs** (*float>0*) – Sampling rate in Hz
- **medianfilter_length** (*float>0, optional*) – The median filter kernel size which is used to filter out the noise in the frequency profile.
- **sample_every** (*float, optional*) – For default see fit_polynomial_on_downsampled_version

**Returns** **fm_rate** – Same size as frequency_profile. The rate of frequency modulation in kHz/ms

**Return type** np.array

**See also:**

*fit_polynomial_on_downsampled_version()*

itsfm.segment.**fit_polynomial_on_downsampled_version**(*frequency_profile*, *fs*, ***kwargs*)

Chooses a subset of all points in the input frequency_profile and fits a piecewise polynomial on it. The start and end of the frequency profile are not altered, and chosen as they are.

**Parameters**

- **frequency_profile** (*np.array*) – The estimated instantaneous frequency in Hz at each sample.
- **fs** (*float>0*) –
- **sample_every** (*float>0, optional*) – The time gap between consecutive points. Defaults to a calculated value which corresponds to 1% of the frequency profiles duration.
- **interpolation_kind** (*int, optional*) – The polynomial order to use while fitting the points. Defaults to 1, which is a piecewise linear fit.

**Returns** **fitted** – Same size as frequency_profile.

**Return type** np.array

itsfm.segment.**fraction_duration**(*input_array*, *fs*, *fraction*)
calculates the duration that matches the required fraction of the input array's duration.

The fraction must be 0 < fraction < 1

itsfm.segment.**check_relevant_duration**(*duration*, *fs*)
checks that the duration is more than the inter-sample duration.

itsfm.segment.**refine_candidate_regions**()
Takes in candidate CF and FM regions and tries to satisfy the constraints set by the user.

itsfm.segment.**check_segment_cf_and_fm**(*cf_samples*, *fm_samples*, *fs*, ***kwargs*)

itsfm.segment.**get_cf_region**(*cf_samples*, *fs*, ***kwargs*)
TODO : generalise to multiple CF regions

**Parameters**

- **cf_samples** (*np.array*) – Boolean with True indicating a Cf region.
- **fs** (*float*) –

**Returns** **cf_region** – The longest continuous stretch

**Return type** np.array

itsfm.segment.**get_fm_regions**(*fm_samples*, *fs*, ***kwargs*)
TODO : generalise to multiple FM regions :param fm_samples: Boolean numpy array with candidate FM samples. :type fm_samples: np.array :param fs: :type fs: float>0 :param min_fm_duration: minimum fm duration expected in seconds. Any fm segment lower than this

duration is considered to be a bad read and discarded. Defaults to 0.5 milliseconds.

**Returns** **valid_fm** – Boolean numpy array with the corrected fm samples.

**Return type** np.array

itsfm.segment.**segment_call_from_background**(*audio*, *fs*, ***kwargs*)
Performs a wavelet transform to track the signal within the relevant portion of the bandwidth.

This methods broadly works by summing up all the signal content above the `` `lowest_relevant_frequency` `` using a continuous wavelet transform.

If the call-background segmentation doesn't work well it's probably due to one of these things:

1. Incorrect `background_threshold` : Play around with different `background_threshold values`.

2. Incorrect `lowest_relevant_frequency` : If the lowest relevant frequency is set outside of the signal's actual frequency range, then the segmentation will fail. Try lower this parameter till you're sure all of the signal's spectral range is above it.
3. Low signal spectral range : This method uses a continuous wavelet transform to localise the relevant signal. Wavelet transforms have high temporal resolution in for high frequencies, but lower temporal resolutions for lower frequencies. If your signal is dominantly low-frequency, try resampling it to a lower sampling rate and see if this works?

If the above tricks don't work, then try bandpassing your signal - may be it's an issue with the in-band signal to noise ratio.

**Parameters**

- **`audio`** (*`np.array`*) –
- **`fs`** (*`float>0`*) – Frequency of sampling in Hertz.
- **`lowest_relevant_freq`** (*`float>0, optional`*) – The lowest frequency band in Hz whose coefficients will be tracked. The coefficients of all frequencies in the signal >= the lowest relevant frequency are tracked. This is the lowest possible frequency the signal can take. It is best to give a few kHz of berth. Defaults to 35kHz.
- **`background_threshold`** (*`float<0, optional`*) – The relative threshold which is used to define the background. The segmentation is performed by selecting the region that is above background_threshold dB relative to the max dB rms value in the audio. Defaults to -20 dB
- **`wavelet_type`** (*`str, optional`*) – The type of wavelet which will be used for the continuous wavelet transform. Run *pywt.wavelist(kind='continuous')* for all possible types in case the default doesn't seem to work. Defaults to mexican hat, 'mexh'
- **`scales`** (*`array-like, optional`*) – The scales to be used for the continuous wavelet transform. Defaults to np.arange(1,10).

**Returns**

- **potential_region** (*np.array*) – A boolean numpy array where True corresponds to the regions which are call samples, and False are the background samples. The single longest continuous region is output.
- **dbrms_profile** (*np.array*) – The dB rms profile of the summed up wavelet transform for all centre frequencies >= lowest_relevant_frequency.s

**Raises**

- **`ValueError`** – When lowest_relevant_frequency is too high or not included in the centre frequencies of the default/input scales for wavelet transforms.
- ***`IncorrectThreshold`*** – When the dynamic range of the relevant part of the signal is smaller or equal to the background_threshold.

`itsfm.segment.`**`identify_valid_regions`**(*condition_satisfied*, *num_expected_regions=1*)

**Parameters**

- **`condition_satisfied`** (*`np.array`*) – Boolean numpy array with samples either being True or False. The array may have multiple regions which satisfy a conditions (True) separated by smaller regions which don't (False).
- **`num_expected_regions`** (*`int > 0`*) – The number of expected regions which satisfy a condition. If >2, then the first two longest continuous regions will be returned, and the smaller regions will be suppressed/eliminated. Defaults to 1.

**Returns** **valid_regions** – Boolean array which identifies the regions with the longest contiguous lengths.

**Return type** np.array

`itsfm.segment.`**`identify_maximum_contiguous_regions`**(*condition_satisfied*, *number_regions_of_interest=1*)

Given a Boolean array - this function identifies regions of contiguous samples that are true and labels each with its own region_number.

**Parameters**

- **`condition_satisfied`** (*`np.array`*) – Numpy array with Boolean (True/False) entries for each sample.
- **`number_regions_of_interest`** (*`integer > 1`*) – Number of contiguous regions which are to be detected. The region ids are output in descending order (longest–>shortest). Defaults to 1.

**Returns**

- **region_numbers** (*list*) – List with numeric IDs given to each contiguous region which is True.
- **region_id_and_samples** (*np.array*) – Two columns numpy array. Column 0 has the region_number, and Column 1 has the individual samples that belong to each region_number.

:raises ValueError : This happens if the condition_satisfied array has no entries that are True.:

`itsfm.segment.`**`pre_process_for_segmentation`**(*call*, *fs*, ***kwargs*)

Performs a series of steps on a raw cf call before passing it for temporal segmentation into cf and fm. Step 1: find peak frequency Step 2: lowpass (fm_audio) and highpass (cf_audio) below

a fixed percentage of the peak frequency

Step 3: calculate the moving dB of the fm and cf audio

**Parameters**

- **`call`** (*`np.array`*) –
- **`fs`** (*`int.`*) – Frequency of sampling in Hertz
- **`peak_percentage`** (*`0<float<1, optional`*) – This is the fraction of the peak at which low and high-pass filtering happens. Defaults to 0.98.
- **`lowpass`** (*`optional`*) – Custom lowpass filtering coefficients. See low_and_highpass_around_threshold
- **`highpass`** – Custom highpass filtering coefficients. See low_and_highpass_around_threshold
- **`window_size`** (*`integer, optional`*) – The window size in samples over which the moving rms of the low+high passed signals will be calculated. For default value see documentation of moving_rms

**Returns** **cf_dbrms, fm_dbrms** – The dB rms profile of the high + low passed versions of the input audio.

**Return type** np.arrays

**See also:**

*`itsfm.segment.low_and_highpass_around_threshold()`*

itsfm.segment.**low_and_highpass_around_threshold**(*audio*, *fs*, *threshold_frequency*, ***kwargs*)

Make two version of an audio clip: the low pass and high pass versions.

**Parameters**

- **audio** (*np.array*) –
- **fs** (*float>0*) – Frequency of sampling in Hz
- **threshold_frequency** (*float>0*) – The frequency at which the lowpass and highpass operations are be done.
- **lowpass,highpass** (*ndarrays, optional*) – The b & a polynomials of an IIR filter which define the lowpass and highpass filters. Defaults to a second order elliptical filter with rp of 3dB and rs of 10 dB. See signal.ellip for more details of rp and rs.
- **pad_duration** (*float>0, optional*) – Zero-padding duration in seconds before low+high pass filtering. Defaults to 0.1 seconds.
- **double_pass** (*bool, optional*) – Low/high pass filter the audio twice. This has been noticed to help with segmentation accuracy, especially for calls with short CF/FM segments where edge effects are particularly noticeable. Defaults to False

**Returns** **lp_audio, hp_audio** – The low and high pass filtered versions of the input audio.

**Return type** np.arrays

itsfm.segment.**get_thresholds_re_max**(*cf_dbrms*, *fm_dbrms*)

itsfm.segment.**calc_proper_kernel_size**(*durn*, *fs*)

scipy.signal.medfilt requires an odd number of samples as kernel_size. This function calculates the number of samples for a given duration which is odd and is close to the required duration.

**Parameters**

- **durn** (*float*) – Duration in seconds.
- **fs** (*float*) – Sampling rate in Hz

**Returns** **samples** – Number of odd samples that is equal to or little less (by one sample) than the input duration.

**Return type** int

itsfm.segment.**resize_by_adding_one_sample**(*input_signal*, *original_signal*, ***kwargs*)

Resizes the input_signal to the same size as the original signal by repeating one sample value. The sample value can either the last or the first sample of the input_signal.

itsfm.segment.**median_filter**(*input_signal*, *fs*, ***kwargs*)

Median filters a signal according to a user-settable window size.

**Parameters**

- **input_signal** (*np.array*) –
- **fs** (*float*) – Sampling rate in Hz.
- **medianfilter_size** (*float, optional*) – The window size in seconds. Defaults to 0.001 seconds.

**Returns** **med_filtered** – Median filtered version of the input_signal.

**Return type** np.array

`itsfm.segment.`**`identify_cf_ish_regions`**(*frequency_profile*, *fs*, ***kwargs*)

Identifies CF regions by comparing the rate of frequency modulation across the signal. If the frequency modulation within a region of the signal is less than the limit then it is considered a CF region.

**Parameters**

- **`frequency_profile`** (*`np.array`*) – The instantaneous frequency of the signal over time in Hz.
- **`fm_limit`** (*`float, optional`*) – The maximum rate of frequency modulation in Hz/s. Defaults to 1000 Hz/s
- **`medianfilter_size`** (*`float, optional`*) –

**Returns**

- **cfish_regions** (*np.array*) – Boolean array where True indicates a low FM rate region. The output may still need to be cleaned before final use.
- *clean_fmrate_resized*

### Notes

If you're used to reading FM modulation rates in kHz/ms then just follow this relation to get the required modulation rate in Hz/s:

X kHz/ms = (X Hz/s)* 10^-6

OR

X Hz/s = (X kHz/ms) * 10^6

**See also:**

*`median_filter()`*

`itsfm.segment.`**`segment_cf_regions`**(*audio*, *fs*, ***kwargs*)

**`exception`** `itsfm.segment.`**`CFIdentificationError`**

**`exception`** `itsfm.segment.`**`IncorrectThreshold`**

## 11.3 API: Measuring sounds

Module that measures each continuous CF and FM segment with either inbuilt or user-defined functions.

`itsfm.measure.`**`measure_hbc_call`**(*call*, *fs*, *cf*, *fm*, ***kwargs*)

Performs common or unique measurements on each of the Cf and FM segments detected.

**Parameters**

- **`audio`** (*`np.array`*) –
- **`fs`** (*`float>0.`*) – Frequency of sampling in Hz.
- **`cf`** (*`np.array`*) – Boolean array with True indicating samples that define the CF
- **`fm`** (*`np.array`*) – Boolean array with True indicating samples that define the FM
- **`measurements`** (*`list, optional`*) – List with measurement functions

**Returns** **measurement_values** – A wide format dataframe with one row corresponding to all the measured values for a CF or FM segment

**Return type** pd.DataFrame

**See also:**

*itsfm.measurement_functions()*

## Example

Create a call with fs and make fake CF and FM segments

```
>>> fs = 1.0
>>> call = np.random.normal(0,1,100)
>>> cf = np.concatenate((np.tile(0, 50), np.tile(1,50))).astype('bool')
>>> fm = np.invert(cf)
```

Get the default measurements by not specifying any measurements explicitly.

```
>>> sound_segments, measures = measure_hbc_call(call, fs,
                                                cf, fm )
>>> print(measures)
```

And here's an example with some custom functions.The default measurements will appear in addition to the custom measurements.

```
>>> from itsfm.measurement_functions import measure_peak_amplitude, measure_peak_
↪frequency
>>> custom_measures = [peak_frequency, measure_peak_amplitude]
>>> sound_segments, measures = measure_hbc_call(call, fs,
                                                cf, fm,
                                                measurements=custom_measures)
```

itsfm.measure.**parse_cffm_segments**(*cf*, *fm*)

Recognises continuous stretches of Cf and FM segments, organises them into separate 'objects' and orders them in time.

**Parameters** **fm** (*cf,*) – Boolean arrays indicating which samples are CF/FM.

**Returns** **cffm_regions_numbered** – Each tuple corresponds to one CF or FM region in the audio. The tuple has two entries 1) the region identifier, eg. 'fm1' and 2) the indices that correspond to the region eg. slice(1,50)

**Return type** np.array with tuples.

## Example

# an example sound with two cfs and an fm in the middle

```
>>> cf = np.array([0,1,1,0,0,0,1,1,0]).astype('bool')
>>> fm = np.array([0,0,0,1,1,1,0,0,0]).astype('bool')
>>> ordered_regions = parse_cffm_segments(cf, fm)
>>> print(ordered_regions)
[['cf1', slice(1, 3, None)], ['fm1', slice(3, 6, None)],
 ['cf2', slice(6, 8, None)]]
```

itsfm.measure.**perform_segment_measurements**(*full_sound*, *fs*, *segment*, *functions_to_apply*, ***kwargs*)

Performs one or more measurements on a specific segment of a full audio clip.

**Parameters**

- **full_sound** (*np.array*) –
- **fs** (*float>0*) –
- **segment** (*tuple*) – First object is a string with the segment's id, eg. 'fm1' or 'cf2' Second object is a slice with the indices of the segment, eg. slice(0,100)
- **functions_to_apply** (*list of functions*) – Each function must be a 'measurement function'. A measurement function is one that accepts a strict set of inputs. check See Also for more details.

**Returns** **results** – A single row with all the measurements results. The first column is always the 'regionid', the rest of the columns are measurement function dependent.

**Return type** pd.DataFrame

### Example

Here we'll create a short segment and take the rms and the peak value of the segment. The *relevant_region* is not an FM region, it is only labelled so here to show how it works with the rest of the package!

```
>>> np.random.seed(909)
>>> audio = np.random.normal(0,1,100)
>>> relevant_region = ('fm1',slice(10,30))
```

The sampling rate doesn't matter for the custom functions defined below, but, it may be important for some other functions.

```
>>> fs = 1 # Hz
>>> from itsfm.measurement_functions import measure_rms, measure_peak
>>> results = perform_segment_measurements(audio, fs, relevant_region,
                                            [measure_rms, measure_peak])
```

itsfm.measure.**find_regions**(*X*)

itsfm.measure.**combine_and_order_regions**(*cf_slices*, *fm_slices*)

itsfm.measure.**assign_cffm_regionids**(*cffm*, *cf_regions*, *fm_regions*)

itsfm.measure.**common_measurements**()
Loads the default common measurement set for any region.

## 11.3.1 Measurement functions

This is a set of *measurement functions* which are used to measure various things about a part of an audio. A *measurement function* is a specific kind of

function which accepts three arguments and outputs a dictionary.

### What is a *measurement function*:

A *measurement function* is a specific kind of function which accepts three arguments and outputs a dictionary. User-defined functions can be used to perform custom measurements on the segment of interest.

### Measurement function parameters

1. the full audio, a np.array
2. the sampling rate, a float>0
3. the *segment*, a slice object which defines the span of the segment. For instance ('fm1', slice(0,100))

### What needs to be returned:

A measurement function must return a dictionary with >1 keys that are strings and items that can be easily incorporated into a Pandas DataFrame and viewed on a csv file with ease. Ideal item types include strings, floats, or tuples.

See the source code of the built-in measurement functions below for an example of how to satisfy the measurement function pattern.

**Attention:** Remember to name the output of the measurement function properly. If the output key of one measurement function is the same as the other, it will get overwritten in the final dictionary!

`itsfm.measurement_functions.`**`measure_rms`**(*audio*, *fs*, *segment*, ***kwargs*)

**See also:**

*`itsfm.signal_processing.rms()`*

`itsfm.measurement_functions.`**`measure_peak_amplitude`**(*audio*, *fs*, *segment*, ***kwargs*)

`itsfm.measurement_functions.`**`start`**(*audio*, *fs*, *segment*, ***kwargs*)

`itsfm.measurement_functions.`**`stop`**(*audio*, *fs*, *segment*, ***kwargs*)

`itsfm.measurement_functions.`**`duration`**(*audio*, *fs*, *segment*, ***kwargs*)

`itsfm.measurement_functions.`**`measure_peak_frequency`**(*audio*, *fs*, *segment*, ***kwargs*)

**See also:**

*`itsfm.signal_processing.get_peak_frequency()`*

`itsfm.measurement_functions.`**`measure_terminal_frequency`**(*audio*, *fs*, *segment*, ***kwargs*)

**See also:**

`itsfm.get_terminal_frequency()`

## 11.4 API : Viewing sounds, parameters and results

Bunch of functions which help in visualising data and results

There is a common pattern in the naming of viewing functions.

1. functions starting with 'visualise' include an overlay of a particular output attribute on top of or with the the original signal. For example *visualise_sound*
2. functions starting with 'plot' are bare bones plots with just the attribute on the y and time on the x.

**class** `itsfm.view.`**`itsFMInspector`**(*segmeasure_out*, *whole_audio*, *fs*, ***kwargs*)

Handles the output from measure_and_segment calls, and allows plotting of the outputs.

**Parameters**

- **`segmeasure_out`** (*`tuple`*) – Tuple object containing three other objects which are the output from segment_and_measure_call 1. segmentation_output : tuple

  Tuple with the *cf* boolean array, *fm* boolean array and *info* dictioanry

  2. **audio_parts** [dictionary] Dictionary with call part labels and values as selected audio parts as np.arrays
  3. **measurements** [pd.DataFrame] A wide-formate dataframe with one row referring to meaurements done on one call part eg. if a call has 3 parts (fm1, cf1, fm2), then there will be three columns and N columns, if N measurements have been done.

- **`whole_audio`** (*`np.array`*) – The audio that was analysed.
- **`fs`** (*`float>0`*) – Sampling rate in Hz.

### Notes

- Not all *visualise* methods may be supported. It depends on the segmentation method at hand.
- All *visualise* methods return one/multiple subplots that could be used and embellished further for your own custom laying over.

**`visualise_fmrate`**()

Plots the spectrogram + FM rate profile in a 2 row plot

**`visualise_accelaration`**()

Plots the spectrogram + accelaration of the frequency profile in a 2 row plot

**`visualise_cffm_segmentation`**()

**`visualise_frequency_profiles`**(*fp_type='all'*)

Visualises either one or all of the frequency profiles that are present in the info dictionary. The function relies on picking up all keys in the info dictionary that end with '<>_fp' pattern.

**Parameters** **`fp_type`** (*`str/list with str's`*) – Needs to correspond to a key found in the info dictionary

**`visualise_pkpctage_profiles`**()

**`visualise_geq_signallevel`**()

Some tracking/segmentation methods rely on using only regions that are above a threshold, the *signal_level* . A moving dB rms window is pass

ed, and only regions above it are

`itsfm.view.`**`check_call_background_segmentation`**(*whole_call*, *fs*, *main_call_mask*, ***kwargs*)

Visualises the main call selection

**Parameters**

- **`whole_call`** (*`np.array`*) – Call audio
- **`fs`** (*`float>0`*) – Sampling rate in Hz
- **`main_call_mask`** (*`np.array`*) – Boolean array where True indicates the sample is part of the main call, and False that it is not.

**Returns** **waveform, spec**

**Return type** pyplot.subplots

## Notes

The appearance of the two subplots can be further changes by varying the keyword arguments. For available keyword arguments see the visualise_sound function.

`itsfm.view.`**`show_all_call_parts`**(*only_call*, *call_parts*, *fs*, ***kwargs*)

**Parameters**

- **`only_call`** (*`np.array`*) –
- **`call_parts`** (*`dictionary`*) – Dictionary with keys 'cf' and 'fm' The entry for 'cf' should only have one audio segment. The entry for 'fm' can have multiple audio segments.
- **`fs`** (*`float>0`*) – Sampling rate in Hz.

**Returns**

**Return type** None

## Notes

For further keyword arguments to customise the spectrograms see documentation for make_specgram This function does not return any output, it only produces a figure with subplots.

`itsfm.view.`**`visualise_fmrate_profile`**(*X*, *freq_profile*, *fs*)

`itsfm.view.`**`plot_accelaration_profile`**(*X*, *fs*)

Plots the frequency acclearation profile of a frequency profile

**Parameters**

- **`X`** (*`np.array`*) – The frequency profile with sample-level estimates of frequency in Hz.
- **`fs`** (*`float>0`*) –

**Returns**

- *A plt.plot which can be used as an independent figure ot*
- *a subplot.*

`itsfm.view.`**`plot_movingdbrms`**(*X*, *fs*, ***kwargs*)

`itsfm.view.`**`visualise_sound`**(*audio*, *fs*, ***kwargs*)

**Parameters**

- **audio** –
- **fs** –
- **fft_size** (*integer>0, optional*) –

**Returns** a0, a1

**Return type** subplots

`itsfm.view.`**`make_specgram`**(*audio*, *fs*, ***kwargs*)

`itsfm.view.`**`get_fftsize`**(*fs*, ***kwargs*)

`itsfm.view.`**`make_overview_figure`**(*call*, *fs*, *measurements*, ***kwargs*)

`itsfm.view.`**`plot_dbrms_cffmprofiles`**(*seg_details*, *fs*)

Makes a plot with CF anf FM dB rms profiles. This method only works for peak-percentage based segmentation.

**Parameters**

- **seg_details** (*tuple*) – Tuple with 3 entries. The third entry needs to be a dictionary with at least the following keys : 'cf_re_fm' and 'fm_re_cf'
- **fs** (*float>0*) – Sample rate in Hz

**Returns**

**Return type** matplotlib plot

## 11.5 API: support modules

### 11.5.1 Frequency tracking

Even though the spectrogram is one of the most dominant time-frequency representation, there are whole class of alternate representations. This module has the code which tracks the dominant frequency in a sound using non-spectrogram methods.

#### The Pseudo Wigner Ville Distribution

The Pseudo Wigner Ville Distribution is an accurate but not so well known method to represent a signal on the time-frequency axis[1]. This time-frequency representation is implemented in the *get_pwvd_frequency_profile*.

References

[1] Cohen, L. (1995). Time-frequency analysis (Vol. 778). Prentice hall.

`itsfm.frequency_tracking.`**`get_pwvd_frequency_profile`**(*input_signal*, *fs*, ***kwargs*)

Generates a clean frequency profile through the PWVD. The order of frequency profile processing is as follows:

1. Split input signal into regions that are greater or equal to the *signal_level*. This speeds up the whole process of pwvd tracking multiple sounds, and ignores the fainter samples.
2. Generate PWVD for each above-noise region.
3. Set regions below background noise to 0Hz
4. Remove sudden spikes and set these regions to values decided by interpolation between adjacent non-spike regions.

**Parameters**

- **input_signal** (*np.array*) –
- **fs** (*float*) –

## Notes

The fact that each signal part is split into independent above-background segments and then frequency tracked can have implications for frequency resolution. Short sounds may end up with frequency profiles that have a lower resolution than longer sounds. Each sound is handled separately primarily for memory and speed considerations.

## Example

Create two chirps in the middle of a somewhat silent recording

```
>>> import matplotlib.pyplot as plt
>>> from itsfm.simulate_calls import make_fm_chirp
>>> from itsfm.view_horseshoebat_call import plot_movingdbrms
>>> from itsfm.view_horseshoebat_call import visualise_call, make_x_time
>>> fs = 44100
>>> start_f, end_f = 1000, 10000
>>> chirp = make_fm_chirp(start_f, end_f, 0.01, fs)
>>> rec = np.random.normal(0,10**(-50/20), 22100)
>>> chirp1_start, chirp1_end = 10000, 10000 + chirp.size
>>> chirp2_start, chirp2_end = np.array([chirp1_start, chirp1_end])+int(fs*0.05)
>>> rec[chirp_start:chirp_end] += chirp
>>> rec[chirp2_start:chirp2_end] += chirp
>>> rec /= np.max(abs(rec))
>>> actual_fp = np.zeros(rec.size)
>>> actual_fp[chirp1_start:chirp1_end] = np.linspace(start_f, end_f, chirp.size)
>>> actual_fp[chirp2_start:chirp2_end] = np.linspace(start_f, end_f, chirp.size)
```

Check out the dB rms profile of the recording to figure out where the noise floor is

```
>>> plot_movingdbrms(rec, fs)
```

```
>>> clean_fp, info = get_pwvd_frequency_profile(rec, fs,
                                                 signal_level=-9,
                                                 extrap_window=10**-3,
                                                 max_acc = 0.6)
>>> plt.plot(clean_fp, label='obtained')
>>> plt.plot(actual_fp, label='actual')
>>> plt.legend()
```

Now, let's overlay the obtained frequency profile onto a spectrogram to check once more how well the dominant frequency has been tracked.

```
>>> w,s = visualise_call(rec, fs, fft_size=128)
>>> s.plot(make_x_time(clean_fp, fs), clean_fp)
```

**See also:**

*itsfm.signal_cleaning.smooth_over_potholes()*, find_above_noise_regions()

itsfm.frequency_tracking.**find_geq_signallevel**(*X*, *fs*, ***kwargs*)
Find regions greater or equal to signal level

`itsfm.frequency_tracking.`**`clean_up_spikes`**(*whole_freqeuncy_profile*, *fs*, ***kwargs*)

**Applies smooth_over_potholes on each non-zero frequency segment** in the profile.

smooth_over_potholes

Let's create a case with an FM and CF tone

```
>>> from itsfm.simulate_calls import make_tone, make_fm_chirp, silence
  >>> fs = 22100
  >>> tone = make_tone(5000, 0.01, fs)
  >>> sweep = make_fm_chirp(1000, 6000, 0.005, fs)
  >>> gap = silence(0.005, fs)
  >>> full_call = np.concatenate((tone, gap, sweep))
```

The raw frequency profile, with very noisy frequency estimates needs to be further cleaned

```
>>> raw_fp, frequency_index = generate_pwvd_frequency_profile(full_call,
                                                               fs)
>>> noise_supp_fp = noise_supp_fp = suppress_background_noise(raw_fp,
                                              full_call,
                                              window_size=25,
                                              background_noise=-30)
```

Even after the noisy parts have been suppressed, there're still some spikes caused by the

```
>>>
```

`itsfm.frequency_tracking.`**`generate_pwvd_frequency_profile`**(*input_signal*, *fs*, ***kwargs*)

Generates the raw instantaneous frequency estimate at each sample. using the Pseudo Wigner Ville Distribution

**Parameters**

- **`input_signal`** (*`np.array`*) –
- **`fs`** (*`float`*) –
- **`pwvd_filter`** (*`Boolean, optional`*) – Whether to perform median filtering with a 2D kernel. Defaults to False
- **`pwvd_filter_size`** (*`int, optional`*) – The size of the square 2D kernel used to median filter the initial PWVD time-frequency representation.
- **`pwvd_window`** (*`float>0, optional`*) – The duration of the window used in the PWVD. See pwvd_transform for the default value.
- **`tfr_cliprange`** (*`float >0, optional`*) – The clip range in dB. Clips all values in the abs(pwvd) time-frequency representation to between max and max*10*(-tfr_cliprange/20.0). Defaults to None, which does not alter the pwvd transform in anyway.

**Returns** **raw_frequency_profile, frequency_indx** – Both outputs are the same size as input_signal. raw_frequency_profile is the inst. frequency in Hz. frequency_indx is the row index of the PWVD array.

**Return type** np.array

**See also:**

*`pwvd_transform()`*, *`track_peak_frequency_over_time()`*, *`itsfm.signal_cleaning.clip_tfr()`*

`itsfm.frequency_tracking.`**`pwvd_transform`**(*input_signal*, *fs*, ***kwargs*)

Converts the input signal into an analytical signal and then generates the PWVD of the analytical signal.

Uses the PseudoWignerVilleDistribution class from the tftb package [1].

**Parameters**

- **`input_signal`** (*`np.array`*) –
- **`fs`** (*`float`*) –
- **`pwvd_window_type`** (*`np.array, optional`*) – The window to be used for the pseudo wigner-ville distribution. If not given, then a hanning signal is used of the default length. The window given here supercedes the 'window_length' argument below.
- **`pwvd_window`** (*`float>0, optional`*) – The duration of the window used in the PWVD. Defaults to 0.001s

**Returns** **time_frequency_output** – Two dimensional array with dimensions of NsamplesxNsamples, where Nsamples is the number of samples in input_signal.

**Return type** np.array

## References

**[1] Jaidev Deshpande, tftb 0.1.1 ,Python module for time-frequency analysis,** https://pypi.org/project/tftb/

`itsfm.frequency_tracking.`**`track_peak_frequency_over_time`**(*input_signal*, *fs*, *time_freq_rep*, ***kwargs*)

Tracks the lowest possible peak frequency. This ensures that the lowest harmonic is being tracked in a multiharmonic signal with similar levels across the harmonics.

EAch 'column' of the 2D PWVD is inspected for the lowest peak that crosses a percentile threshold, and this is then taken as the peak frequency.

**Parameters**

- **`input_signal`** (*`np.array`*) –
- **`fs`** (*`float>0`*) –
- **`time_freq_rep`** (*`np.array`*) – 2D array with the PWVD representation.
- **`percentile`** (*`0<float<100, optional`*) –

**Returns** **peak_freqs, peak_inds** – Arrays with same size as the input_signal. peak_freqs is the frequencies in Hz, peak_inds is the row index.

**Return type** np.array

**See also:**

*`find_lowest_intense_harmonic_across_TFR()`*, *`get_most_intense_harmonic()`*

`itsfm.frequency_tracking.`**`find_lowest_intense_harmonic_across_TFR`**(*tf_representation*, ***kwargs*)

`itsfm.frequency_tracking.`**`get_most_intense_harmonic`**(*time_slice*, ***kwargs*)

Searches a single column in a 2D array for the first region which crosses the given percentile threshold.

`itsfm.frequency_tracking.`**`get_midpoint_of_a_region`**(*region_object*)

`itsfm.frequency_tracking.`**`accelaration`**(*X*, *fs*)

Calculates the absolute accelrateion of a frequency profile in kHz/ms^2 `itsfm.frequency_tracking.`**`speed`**(*X*, *fs*)

Calculates the abs speed of the frequency profile in kHz/ms

`itsfm.frequency_tracking.`**`get_first_region_above_threshold`**(*input_signal*, ***kwargs*)

Takes in a 1D signal expecting a few peaks in it above the percentil threshold. If all samples are of the same value, the region is restricted to the first two samples.

**Parameters**

- **`input_signal`** (*`np.array`*) –
- **`percentile`** (*`0<float<100, optional`*) – The percentile threshold used to set the threshold. Defaults to 99.5

**Returns** **region_location** – If there is at least one region above the threshold a tuple with the output from scipy.ndimage.find_objects. Otherwise None.

**Return type** tuple or None

`itsfm.frequency_tracking.`**`frequency_spike_detection`**(*X*, *fs*, ***kwargs*)

Detects spikes in the frequency profile by monitoring the accelration profile through the sound.

**Parameters**

- **`X`** (*`np.array`*) – A frequency profile with sample-level estimates of frequency in Hz
- **`fs`** (*`float>0`*) –
- **`max_acc`** (*`float>0, optional`*) – Maximum acceleration in the frequency profile. Defaults to 0.5kHz/ms^2

**Returns** **anomalous** – Boolean

**Return type** np.array

## 11.5.2 Signal processing

Functions which actually do the calculations on the raw input signal

Module with signal processing functions in it used by both measure and segment modules.

`itsfm.signal_processing.`**`dB`**(*X*)

Calculates the 20log of X

`itsfm.signal_processing.`**`rms`**(*X*)

Root mean square of a signal

`itsfm.signal_processing.`**`calc_energy`**(*X*)

Sum of all squared samples

`itsfm.signal_processing.`**`get_power_spectrum`**(*audio*, *fs=250000.0*)

Calculates an RFFT of the audio. :param audio: :type audio: np.array :param fs: Frequency of sampling in Hz :type fs: int

**Returns**

- **dB_power_spectrum** (*np.array*) – dB(power_spectrum)
- **freqs** (*np.array*) – Centre frequencies of the RFFT.

`itsfm.signal_processing.`**`calc_sound_borders`**(*audio*, *percentile=99*)

Gives the start and stop of a sound based on the percentile cumulative energy values.

**Parameters**

- **audio** (*np.array*) –
- **percentile** (*float, optional*) – Value between 100 and 0. The sound border is calcualted as the border which encapsulates the percentile of energy Defaults to 99.

**Returns** **start, end**

**Return type** int

`itsfm.signal_processing.`**`get_robust_peak_frequency`**(*audio*, ***kwargs*)

Makes a spectrogram from the audio and calcualtes the peak frequency by averaging each slice of the spectrogram's FFT's.

This 'smooths' out the structure of the power spectrum and allows a single and clear peak detection.

Thanks to Holger Goerlitz for the suggestion.

**Parameters**

- **audio** (*np.array*) –
- **fs** (*float*) – Frequency of sampling in Hz
- **seg_length** (*int, optional*) – The size of the FFt window used to calculate the moving FFT slices. DEfaults to 256
- **noverlap** (*int, optional*) – The number of samples overlapping between one FFT slice and the next. Defaults to seg_length -1

**Returns** **peak_frequency** – Frequency with highest power in the audio in Hz.

**Return type** float

`itsfm.signal_processing.`**`get_peak_frequency`**(*audio*, *fs*)

Gives peak frequency and frequency resolution with which the measurement is made

**Parameters**

- **audio** (*np.array*) –
- **fs** (*float>0*) – sampling rate in Hz

**Returns** **peak_freq, freq_resolution** – The peak frequency and frequency resolution of this peak frequency in Hz.

**Return type** float

`itsfm.signal_processing.`**`get_frequency_resolution`**(*audio*, *fs*)

**Parameters**

- **audio** (*np.array*) –
- **fs** (*float>0*) – sampling rate in Hz

**Returns** **resolution** – The frequency resolution in Hz.

**Return type** float

`itsfm.signal_processing.`**`moving_rms`**(*X*, ***kwargs*)

Calculates moving rms of a signal with given window size. Outputs np.array of *same* size as X. The rms of the last few samples <= window_size away from the end are assigned to last full-window rms calculated

**Parameters**

- **X** (*np.array*) – Signal of interest.
- **window_size** (*int, optional*) – Defaults to 125 samples.

**Returns** **all_rms** – Moving rms of the signal.

**Return type** np.array

`itsfm.signal_processing.`**`moving_rms_edge_robust`**(*X*, ***kwargs*)

Calculates moving rms of a signal with given window size. Outputs np.array of *same* size as X. This version is robust and doesn't suffer from edge effects as it calculates the moving rms in both forward and backward directions and calculates a consensus moving rms profile.

The consensus rms profile is basically achieved by taking the left half of the forward rms profile and concatenating it with the right hald of the backward passed rms profile.

**Parameters**

- **`X`** (*`np.array`*) – Signal of interest.
- **`window_size`** (*`int, optional`*) – Defaults to 125 samples.

**Returns** **all_rms** – Moving rms of the signal.

**Return type** np.array

## Notes

moving_rms_edge_robust may not be too accurate when the rms is expected to vary over short time scales in the centre of the signal!!

`itsfm.signal_processing.`**`form_consensus_moving_rms`**(*forward*, *backward*)

**Parameters**

- **`backward`** (*`forward,`*) – Two arrays of the same dimensions.
- **`and returns the consensus maximum value at each sample.`** (*`Compares`*) –

`itsfm.signal_processing.`**`median_filter`**(*input_signal*, *fs*, ***kwargs*)

Median filters a signal according to a user-settable window size.

**Parameters**

- **`input_signal`** (*`np.array`*) –
- **`fs`** (*`float`*) – Sampling rate in Hz.
- **`medianfilter_size`** (*`float, optional`*) – The window size in seconds. Defaults to 0.001 seconds.

**Returns** **med_filtered** – Median filtered version of the input_signal.

**Return type** np.array

`itsfm.signal_processing.`**`calc_proper_kernel_size`**(*durn*, *fs*)

scipy.signal.medfilt requires an odd number of samples as kernel_size. This function calculates the number of samples for a given duration which is odd and is close to the required duration.

**Parameters**

- **`durn`** (*`float`*) – Duration in seconds.
- **`fs`** (*`float`*) – Sampling rate in Hz

**Returns** **samples** – Number of odd samples that is equal to or little less (by one sample) than the input duration.

**Return type** int

itsfm.signal_processing.**resize_by_adding_one_sample**(*input_signal*, *original_signal*, ***kwargs*)

Resizes the input_signal to the same size as the original signal by repeating one sample value. The sample value can either the last or the first sample of the input_signal.

itsfm.signal_processing.**get_terminal_frequency**(*audio*, *fs*, ***kwargs*)

Gives the -XdB frequency from the peak.

The power spectrum is calculated and smoothened over 3 frequency bands to remove complex comb-like structures.

Then the lowest frequency below XdB from the peak is returned.

**Parameters**

- **audio** (*np.array*) –
- **fs** (*float>0*) – Sampling rate in Hz
- **terminal_frequency_threshold** (*float, optional*) – The terminal frequency is calculated based on finding the level of the peak frequency and choosing the lowest frequency which is -10 dB (20log10) below the peak level. Defaults to -10 dB

**Returns**

- *terminal_frequency*
- *threshold*

### Notes

Careful about setting threshold too low - it might lead to output of terminal frequencies that are actually in the noise, and not part of the signal itself.

### 11.5.3 Signal cleaning

Functions which refine, clean and detect outliers.

This module handles the identification and cleaning of noise in signals. A 'noisy' signal is one that has spikes in it or sudden variations in a continuous looking function. Most of these functions are built to detect and handle sudden spikes in the frequency profile estimates of a sound.

itsfm.signal_cleaning.**exterpolate_over_anomalies**(*X*, *fs*, *anomalous*, ***kwargs*)

Ex(tra)+(in)ter-polates –> Exterpolates over anomalous regions. Anomalous regions are either 'edge' or 'island' types. The 'edge' anomalies are those which are at the extreme ends of the signal. The 'island' anomalies are regions with non-anomalous regions on the left and right.

An 'edge' anomalous region is handled by running a linear regression on the neighbouring non-anomalous region, and using the slope to extrapolate over the edge anomaly.

An 'island' anomaly is handled by interpolating between the end values of the neighbouring non-anomalous regions.

**Parameters**

- **X** (*np.array*) –
- **fs** (*float>0*) – Sampling rate in Hz
- **anomalous** (*np.array*) – Boolean array of same size as X True indicates an anomalous sample.

- **extrap_window** (*float>0, optional*) – The duration of the extrapolation window in seconds. Defaults to 0.1ms

**Returns** **smooth_X** – Same size as X, with the anomalous regions

**Return type** np.array

## Notes

Only extrapolation by linear regression is supported currently. The *extrap_window* parameter is important especially if there is a high rate of frequency modulation towards the edges of the sound. When there is a high freq. mod. at the edges it is better to set the *extrap_window* small. However, setting it too small also means that the extrapolation may not be as nice anymore.

## Example

*not up to date!!!*

**See also:**

`find_closest_normal_region()`

`itsfm.signal_cleaning.`**`fix_island_anomaly`**(*X*, *fs*, *anomaly*, *ref_region_length*, ***kwargs*)

First tries to interpolate between the edges of the anomaly at hand. If the interpolation leads to a very drastic slope, a 'sensible' extrapolation is attempted using parts of the non-anomalous signal.

**Parameters**

- **X** (*np.array*) –
- **fs** (*float>0*) –
- **anomaly** (*tuple slice*) – scipy.ndimage.find_objects output (slice(start,stop,None),)
- **ref_region_length** (*int>0*) – The number of samples to be used as a reference region in case of extrapolation
- **max_fmrate** (*float>0, optional*) – The maximum fm rate to be tolerated while interpolating in kHz/ms Defaults to 100 kHz/ms.

**Returns** **interpolated** – Array of same size as anomaly.

**Return type** np.array

`itsfm.signal_cleaning.`**`extrapolate_sensibly`**(*X*, *fs*, *anomaly*, *ref_region_length*, ***kwargs*)

Function called when *fix_island_anomaly* detects direct interpolation will lead to unrealistic slopes. This function is called when there's a big difference in values across an anomalous region and an extrapolation must be performed which will not alter the signal drastically.

**The method tries out the following:**

1. Look left and right of the anomaly to see which region has higher frequency content.
2. Extrapolate in the high-to-low frequency direction.

This basically means that if the local inspection window around anomaly has a sweep between 20-10kHZ on the left and a 0Hz region on the right - the anomaly will be extrapolated with the slope from the sweep region because it has higher frequency content.

### Example

```
>>> freq_profile = [np.zeros(10), np.arange(15,30,5)*1000]
>>> fs = 1.0
>>> x = np.concatenate(freq_profile)[::-1]
>>> anom = (slice(2, 5, None),)
>>>
>>> plt.plot(x, label='noisy frequency profile')
>>> anom_x = np.zeros(x.size, dtype='bool')
>>> anom_x[anom[0]] = True
>>> plt.plot(anom_x*8000, label='identified anomaly')
>>> extrap_out = extrapolate_sensibly(x, fs, anom, 4)
>>> sensibly_extrap = x.copy()
>>> sensibly_extrap[anom_x] = extrap_out
>>> plt.plot(sensibly_extrap, label='extrapolated')
>>> plt.legend()
```

`itsfm.signal_cleaning.`**`get_neighbouring_regions`**(*X*, *target*, *region_size*)

Takes out samples of *region_size* on either size of the target.

**Parameters**

- **X** (*np.array*) –
- **target** (*slice*) – ndimage.find_objects type slice
- **region_size** (*int >0*) –

**Returns** **left_and_right**

**Return type** list

`itsfm.signal_cleaning.`**`calc_coarse_fmrate`**(*X*, *fs*, ***kwargs*)

Calculates slope by subtracting the difference between 1st and last sample and dividing it by the length of the array. The output is then converted to units of kHz/ms.

**Parameters**

- **X** (*np.array*) – Frequency profile with values in Hz.
- **fs** (*float>0*) –

`itsfm.signal_cleaning.`**`anomaly_extrapolation`**(*region*, *X*, *num_samples*, ***kwargs*)

Takes X values next to the region and fits a linear regression into the region. This is only suitable for cases where the anomalous region is at an 'edge' - either one of its samples is 0 or the last sample of X.

**Parameters**

- **region** (*object tuple*) – A slice type object which is the output from scipy.ndimage.find_objects This is a slice inside a list/tuple.
- **X** (*np.array*) – The original array over which the extrapolation is to be performed
- **num_samples** (*int>0*) – The number of samples next to the region to be used to fit the data for extrapolation into the region.

**Returns** **extrapolated** – The values corresponding to the extrapolated region.

**Return type** np.array

### Notes

1. This function covers 90% of cases… if there is an anomaly right next to an edge anomaly with <num_samples distance – of course things will go whack.

> **Warning:** A mod on this function also allows extrapolation to occur if there are < num_samples next to the anomaly - this might make the function a bit lax in terms of the extrapolations it produces.

`itsfm.signal_cleaning.`**`anomaly_interpolation`**(*region*, *X*, ***kwargs*)

Interpolates X values using values of X adjacent to the region.

**Parameters**

- **region** (*object tuple*) – Output from scipy.ndimage.find_objects
- **X** (*np.array*) –

**Returns** **full_span** – The values of interpolated X, of same size as the region length.

**Return type** np.array

`itsfm.signal_cleaning.`**`smooth_over_potholes`**(*X*, *fs*, ***kwargs*)

A signal can show drastic changes in its value because of measurement errors. These drastic variations in signal are called potholes (uneven parts of a road). This method tries to 'level' out the pothole by re-setting the samples of the pothole. A linear interpolation is done from the start of a pothole till its end using the closest non-pothole samples.

A pothole is identified by a region of the signal with drastic changes in slope. A moving window calculates N slopes between the focal sample and the Nth sample after it to estimate if the Nth sample could be part of a pothole or not.

**Parameters**

- **X** (*np.array*) –
- **fs** (*float>0*) –
- **max_stepsize** (*float>0, optional*) – The maximum absolute difference between adjacent samples. Defaults to 50.
- **pothole_inspection_window** (*float>0, optional*) – The length of the moving window that's used to discover potholes. See identify_pothole_samples for default value.

**Returns**

- *pothole_covered*
- *pothole_regions*

**See also:**

*identify_pothole_samples()*, pothole_inspection_window()

`itsfm.signal_cleaning.`**`identify_pothole_samples`**(*X*, *fs*, ***kwargs*)

Moves a sliding window and checks the values of samples in the sliding window. If the jump of values between samples is not linearly propotional to the expected max_stepsize, then it is labelled a pothole sample.

A pothole sample is one which represents a sudden jump in the values - indicating a noisy tracking of the frequency. The jump in values in a non-noisy signal is expected to be proportional to the distance between the samples.

For instance, if :

```
>>> a = np.array([10, 2, 6, 10, 12])
```

If the max step size is 2, then because abs(10-2)>2, it causes a pothole to appear on 2. There is no pothole label on the 2nd index because abs(10-6) is not >4. Because 10 and 6 are two samples apart, the maximum allowed jump in value is max_stepsize*2, which is 4.

For optimal pothole detection the 'look-ahead' span of the pothole_inspection_window should at least the size of the longest expected potholes. Smaller window sizes will lead to false negatives.

**Parameters**

- **X** (*np.array*) –
- **fs** (*float>0*) –
- **max_stepsize** (*float>0*) – The max absolute difference between the values of one sample to the next.
- **pothole_inspection_window** (*float>0, optional*) – Defaults to 0.25ms

**Returns** **pothole_candidates** – Boolean array with same size as X. Sample that are True represent pothole candidates.

**Return type** np.array

**See also:**

*detect_local_potholes()*

itsfm.signal_cleaning.**onepass_identify_potholes**(*X*, *fs*, *max_stepsize*, ***kwargs*)

itsfm.signal_cleaning.**detect_local_potholes**(*X*, *max_step_size*)

accepts a 1D array and checks the absolute difference between the first sample and all other samples.

The samples with difference greater than the linearly expected increase from max_step_sizes are labelled candidate potholes.

**Parameters**

- **X** (*np.array*) –
- **max_step_size** (*float>=0*) –

**Returns** **candidate_potholes** – Boolean array of same size as X

**Return type** np.array

itsfm.signal_cleaning.**get_all_spikeish_indices**(*regions*)

itsfm.signal_cleaning.**find_non_forbidden_index**(*candidate*, *forbidden_indices*, *search_direction*, *X*)

itsfm.signal_cleaning.**remove_bursts**(*X*, *fs*, ***kwargs*)

Bursts are brief but large jumps in the signal above zero. Even though they satisfy most of the other conditions of beginning above the noise floor and of being above 0 value, they still are too short to be relevant signals.

**Parameters**

- **X** (*np.array*) – The noisy signal to be handled
- **fs** (*float>0*) – Sampling rate in Hz.
- **min_element_length** (*float>0, optional*) – The minimum length a section of the signal must be to be kept in seconds. Defaults to 5 inter-sample-intervals.

**Returns** **X_nonspikey** – Same size as X, and without very short segments.

**Return type** np.array

**See also:**

`segments_above_min_duration()`

## Notes

An inter-sample-interval is defined as 1/fs

`itsfm.signal_cleaning.`**`segments_above_min_duration`**(*satisfies_condition*, *min_samples*)

Accepts a boolean array and looks for continuous chunks that are above a minimum length.

**Parameters**

- **`satisfies_condition`** (*`np.array`*) – Boolean array where samples with True satisfy a condition.
- **`min_samples`** (*`int >0`*) – The minimum number of samples a continuous region of True must be to be kept.

**Returns** **above_min_duration** – Same size as satisfies_condition, with only the continuous chunks that are above min_samples.

**Return type** np.array

`itsfm.signal_cleaning.`**`suppress_background_noise`**(*main_signal*, *input_audio*, ***kwargs*)

`itsfm.signal_cleaning.`**`suppress_frequency_spikes`**(*noisy_profile*, *input_audio*, *fs*, ***kwargs*)

`itsfm.signal_cleaning.`**`suppress_to_zero`**(*target_signal*, *basis_signal*, *threshold*, *mode='below'*)

Sets the values of the target signal to zero if the samples in the basis_signal are $geq$ or $leq$ the threshold

**Parameters**

- **`basis_signal`** (*`target_signal,`*) –
- **`threshold`** (*`float`*) –
- **`mode`** (*`['below', 'above'], str`*) –

**Returns** **cleaned_signal** – A copy of the target signal with the values that are below/above the threshold set to zero

**Return type** np.array

## Example

# create a basis signal with a ‘weak’ left half and a ‘loud’ right hald # we want to suppress the we >>> basis = np.concatenate((np.arange(10), np.arange(100,200))) >>> target_signal = np.random.normal(0,1,basis.size) >>> cleaned_target = suppress_to_zero(basis, target_signal, 100, mode=’above’)

`itsfm.signal_cleaning.`**`clip_tfr`**(*tfr*, ***kwargs*)

**Parameters**

- **`tfr`** (*`np.array`*) – 2D array with the time-frequency representation of choice (pwvd, fft etc). The tfr must have real-valued non-negative values as the clip range is defined in dB.

- **`tfr_cliprange`** (*`float >0, optional`*) – The maximum dynamic range in dB which will be used to track the instantaneous frequency. Defaults to None. See *Notes* for more details

**Returns** **clipped_tfr** – A 2d array of same shape as *tfr*, with values clipped between [max, max x 10^(tfr_range/20)]

**Return type** np.array

## Notes

The *tfr_cliprange* is used to remove the presence of background noise, faint harmonics or revernberations/echoes in the audio. This of course all assumes that the main signal itself is sufficiently intense in the first place.

After the PWVD time-frequency represenation is made, values below X dB of the maximum value are 'clipped' to the same minimum value. eg. if the pwvd had values of [0.1, 0.9, 0.3, 1, 0.001, 0.0006] and the tfr_cliprange is set to 6dB, then the output of the clipping will be [0.5, 0.9, 0.3, 1, 0.5, 0.5]. This step essentially eliminates any variation in the array, thus allowing a clear tracking of the highest component in it.

`itsfm.signal_cleaning.`**`conditionally_set_to`**(*X*, *conditional*, *bool_state*)

Inverts the samples in X where the conditional is True. :param X: Boolean :type X: np.array :param conditional: Boolean :type conditional: np.array :param bool_state: :type bool_state: [True, False]

**Returns** **cond_set_X** – conditionally set X

**Return type** np.array

## Notes

this function is useful if you want to 'suppress' a few samples conditionally based ont he values of the same samples on another array.

## Example

```
>>> x = np.array([True, True, False, False, True])
>>> y = np.array([0,0,10,10,10])
Imagine x is some kind of detection array, while y is the
signal-to-noise ratio at each of the sample points. Of course,
you'd like to discard all the predictions from low SNR measurements.
Let's say you want to keep only those entries in X where y is >1.
>>> x_cond = conditionally_set_to(x, y<10, False)
>>> x_cond
```

np.array([False, False, False, False, True ])

### 11.5.4 Batch processing

Runs the batch processing option. The main outputs are the call measurements and the visualisations. (See __main__.py)

```
$ python -m itsfm -batchfile template_batchfile.csv
```

Also allows the user to run only one specific row of the whole batch file

```
$ python -m itsfm -batchfile template_batchfile.csv -one_row 10
```

The line above loads the 11th row (0-based indexing!!) of the template_batchfile

`itsfm.batch_processing.`**`run_from_batchfile`**(*batchfile_path*, ***kwargs*)

**Parameters** **`batchfile_path`** (*`str/path`*) – Path to a batchfile

**Keyword Arguments**

- **`one_row`** (*`int, optional`*) – A specific row to be loaded from the whole batchfile The first row starts with 0. Defaults to None
- **`_from`** (*`int, optional`*) – Row to start the batchfile processing from. Defaults to None
- **`_till`** (*`int, optional`*) – Row to end the batchfile processing. Defaults to None

`itsfm.batch_processing.`**`subset_batch_data`**(*batch_data*, ***kwargs*)

**Parameters** **`batch_data`** (*`pd.DataFrame`*) –

**Keyword Arguments**

- **`one_row`** (*`int, optional`*) – Defaults to None
- **`_from`** (*`int, optional`*) – Defaults to None
- **`_till`** (*`int, optional`*) – The row number the analysis should run till, including the end point. Remember the row numbering starts from 0! Defaults to None

**Returns** **subset_batch_data** – Either a copy of batch_data or a part of batch_data

**Return type** pd.DataFrame

#### Example

# let's get only one row from the fake batch data file >>> batch = pd.DataFrame(data={'a':range(10), 'b':range(10)}) >>> onerow = subset_batch_data(batch, one_row=5) >>> print(onerow) # get a limited range of the dataframe >>> part = subset_batch_data(batch, _from=3, _till=8) >>> print(part)

`itsfm.batch_processing.`**`measurement_file_action`**(***kwargs*)

Either lets the measurement file remain, or deletes it if present

**Keyword Arguments** **`del_measurement`** (*`boolean`*) – True means all files starting with 'measurement' are deleted

`itsfm.batch_processing.`**`onerow_used_properly`**(***kwargs*)

Checks that the -one_row argument is not used in conjunction with -from or -till

`itsfm.batch_processing.`**`save_measurements_to_file`**(*output_filepath*, *audio_file_name*, *previous_rows*, *measurements*)

Continously saves a row to a csv file and updates it.

Thanks to tmss @ https://stackoverflow.com/a/46775108

**Parameters**

- **`output_filepath`** (*`str/path`*) –
- **`audio_file_name`** (*`str.`*) –
- **`previous_rows`** (*`pd.DataFrame`*) – All the previous measurements. Can also just have a single row.
- **`measurements`** (*`pd.DataFrame`*) – Current measurements to be incorporated

**Returns**

**Return type** None, previous rows

## Notes

Main side effect is to write an updated version of the output file.

`itsfm.batch_processing.`**`load_raw_audio`**(*kwargs*)

Takes a dictioanry input. All the parameter names need to be keys in the input dictionary.

**Parameters**

- **`audio_path`** (*`str/path`*) – Path to audio file
- **`channel`** (*`int, optional`*) – Channel number to be loaded - starting from 1! Defaults to 1.
- **`start,stop`** (*`float, optional`*) –

**Returns** **raw_audio** – The audio corresponding to the start and stop times and the required channel.

**Return type** np.array

`itsfm.batch_processing.`**`to_separate_from_background`**(*arguments*)

`itsfm.batch_processing.`**`to_list_w_funcs`**(*X*, *source_module=<module 'itsfm.measurement_functions' from '/home/docs/checkouts/readthedocs.org/user_builds/itsfm/envs/latest/lib/pytho packages/itsfm-0.0.1-py3.7.egg/itsfm/measurement_functions.py'>*, ***kwargs*)

**Parameters**

- **`X`** (*`str`*) – String defining a list with commas as separators eg. "[func_name1, func_name2] "
- **`source_module`** (*`str, optional`*) – Defaults to itsfm.measurement_functions
- **`signs_to_remove`** (*`list w str`*) – Any special signs to remove from each str in the list of comma separated strings. Defaults to None.

**Returns** list with functions belonging to the source module

**Return type** list_w_funcs

### Example

```
>>> x = "[measure_rms, measure_peak_amplitude]"
>>> list_w_funcs = to_list_w_funcs(x)
```

`itsfm.batch_processing.`**`remove_punctuations`**(*full_str*, ***kwargs*)

Removes spaces, ], and [ in a string. Additional signs can be removed too

**Parameters**

- **`full_str`** (*`str`*) – A long string with multiple punctuation marks to be removed (space, comma, ])
- **`signs_to_remove`** (*`list w str', optional`*) – Additional specific punctuation/s to be removed Defaults to None

**Returns** clean_str

**Return type** str

`itsfm.batch_processing.`**`parse_batchfile_row`**(*one_row*)

checks for all user-given arguments and removes any columns with DEFAULT in them.

**Parameters** **`one_row`** (*`pd.DataFrame`*) – A single row with multiple column names, corresponding to compulsory required arguments and the optional ones

**Returns** **arguments** – Simple dictioanry with one entry for each key.

**Return type** dictionary

`itsfm.batch_processing.`**`make_to_oned_dataframe`**(*oned_series*)

**Parameters** **`oned_series`** (*`pd.Series`*) – One dimensional pd.Series with columns and values

**Returns**

**Return type** oned_df

**`exception`** `itsfm.batch_processing.`**`ImproperArguments`**

# PYTHON MODULE INDEX

# INDEX