(GIS) OSmOSE

Open Science meets
Ocean Sound Explorers

Pushing the standards forward in Underwater Passive Acoustics processing for both theory and code.

OSmOSE Working Report

Authorship This document was drafted by

- Paul Nguyen Hong Duc[1)]
- Dorian Cazau[2)]

belonging to the following institutes (at the time of their contribution): 1) Institut Jean le Rond d'Alembert, Sorbonne Universités, 2) Lab-STICC, ENSTA Bretagne.

Document Review Though the views in this document are those of the authors, it was reviewed by a panel of acousticians before publication. This enabled a degree of consensus to be developed with regard to the contents, although complete unanimity of opinion is inevitably difficult to achieve. Note that the members of the review panel and their employing organisations have no liability for the contents of this document.

The Review Panel consisted of the following experts (listed in alphabetical order):

- Ronan Fablet[1)]

belonging to the following organisms / research institutes (at the time of their contribution): 1) Lab-STICC, IMT Atlantique.

Last date of modifications June 11, 2020

Recommended citation Nguyen, P. et al. "Achieving intensive computation of low-level descriptors "at scale - with speed" in Underwater Passive Acoustics", OSmOSE Product Presentation (version dating from June 11, 2020, distributed openly on `https://osmose.xyz/`)

Future revisions Revisions to this document will be considered at any time, as well as suggestions for additional material or modifications to existing material, and should be communicated to Dorian Cazau (dorian.cazau@ensta-bretagne.fr).

Acknowledgements We thank the Ple de Calcul et de Donnes pour la Mer[1] from IFREMER for the provision of their infrastructure DATARMOR and associated services. We also would like to thank our main sponsors in this work: CominLabs[2] through the innovation action Tech4Whales, DREC Agence Française de la Biodiversité[3] and ISblue[4]. The authors also would like to acknowledge the assistance of the review panel, and the many people who volunteered valuable comments on the draft at the consultation phase.

[1] `https://wwz.ifremer.fr/Recherche/Infrastructures-de-recherche/Infrastructures-numeriques/Pole-de-Calcul-et-de-Donnees-pour-la-Mer`

[2] `https://www.cominlabs.u-bretagneloire.fr/`

[3] `https://www.afbiodiversite.fr/`

[4] `https://www.isblue.fr/about-us/`

Contents

Abstract

In the Big Data era, the community of PAM faces strong challenges, including the need for more standardized processing tools across its different applications in oceanography, and for more scalable and high-performance computing systems to process more efficiently the everly growing datasets. In this work we address conjointly both issues by first proposing a detailed theory-plus-code document of a classical analysis workflow to describe the content of PAM data, which hopefully will be reviewed and adopted by a maximum of PAM experts to make it standardized. Second, we transposed this workflow into the Scala language within the Spark/Hadoop frameworks so it can be directly scaled out on several node cluster.

Chapter 1

Introduction

1.1 Context

Measured noise levels in Passive Acoustic Monitoring (PAM) are sometimes difficult to compare because different measurement methodologies or acoustic metrics are used, and results can take on different meanings for each different application, leading to a risk of misunderstandings between scientists from different PAM disciplines. For reasons of comparability, and since it is cumbersome to define each term every time it is used, some common definitions are needed for acoustic metrics.

In the hope of boosting standardization and interoperability, numerous efforts have already been made to outline some best practices regarding PAM both as an ocean observing measure and as a STIC discipline. Robinson et al. (2014) provided a full technical report of best practices, reviewed by a comitee of experts. Merchant et al. (2015) provided a comprehensive overview of PAM methods to characterize acoustic habitats, and released an open-source toolbox both in R and Matlab with a theoretical document.

1.2 Contributions

In the same vein, our work addresses the need for a common approach, and the desire to promote best practices for processing the data, and for reporting the measurements using appropriate metrics.

We release a new open source end-to-end analytical workflow for description and interpretation of underwater soundscapes, along with the present document. We outline the following contributions

- this workflow has been implemented in **three different computer languages**: Matlab, Python and Scala. These three implementations perfectly match in regards to the unitary tests done on core functions, with rms error below 10^{-16}, and to the data processing operations and end-user functionalities and results. Note also that in these implementations we try at best to fit with "the best practices in programming" from the DCLDE community in Passive Acoustic Monitoring, for the Matlab implementation, and with the web community and data scientists, for the Scala implementation. These different versions of the workflow have been released on github under a GNU licence;
- in this document, we aligned the lines of codes with their corresponding theoretical signal processing definitions, so as to **fill at best the gap between theory and code**;
- the Scala implementation of the workflow allows for a **direct and transparent scaling out of data processing** over a CPU cluster using the Hadoop/Spark frameworks, allowing for significant computational gain.

As stated in the preamble, this workflow has been collaboratively elaborated, co-developed and reviewed by a research team gathering more than 2 PAM experts over 2 different institutes. Thus, it should provide a reliable value of standardization. Also, during all our work, we built at best on similar works in order to avoid replicating previous efforts. In table 1.1, we list the different source codes on which we have relied to implement our workflow. In reference to these sources, we systematically highlighted agreements and

disagreements with their implementations (and theoretical explanations when present) in the paragraphs named "Discussion", discussed them in regards to each of these different sources and thus justified the choices made for our own implementation.

Eventually, note that reported codes in this document are not representative of their real implementation structure (e.g. in terms of functions), but we rather focus on reporting the essential code lines that implement litteraly each equation and theoretical points.

Code source	Language	Main functions used	References
Package scipy v-1.0.0	Python	stft.py / spectrogram.py / welch.py	`https://www.scipy.org/`
Matlab 2014a	Matlab	spectrogram.m / pwelch.m	MathWorks
pamGuide	R / Matlab	PAMGuide.R	Merchant et al. (2015)

Table 1.1: Details of codes reviewed.

1.3 Overview

As shown in figure 1.1, our workflow is composed of the following blocks

- pre-processing (Sec. 2);
- segmentation (Sec. 3);
- feature computation and integration (Sec. 4);

Note that we have two different time scales for data analysis:

- first scale (see Section 3): for feature computation in short-term analysis windows of length "windowSize";
- second scale (see Section 4): for feature integration in longer time segments, applied when $segmentSize > windowSize$.

Note that when $segmentSize <= windowSize$, these time scales are similar and only one segmentation is performed.

The implemented acoustic metrics are (selected among the list in (Robinson et al., 2014, Sec. 2.1.2))

- **PSD** Power Spectral Density;
- **TOL** Third-Octave Levels;
- **SPL** Sound Pressure Level

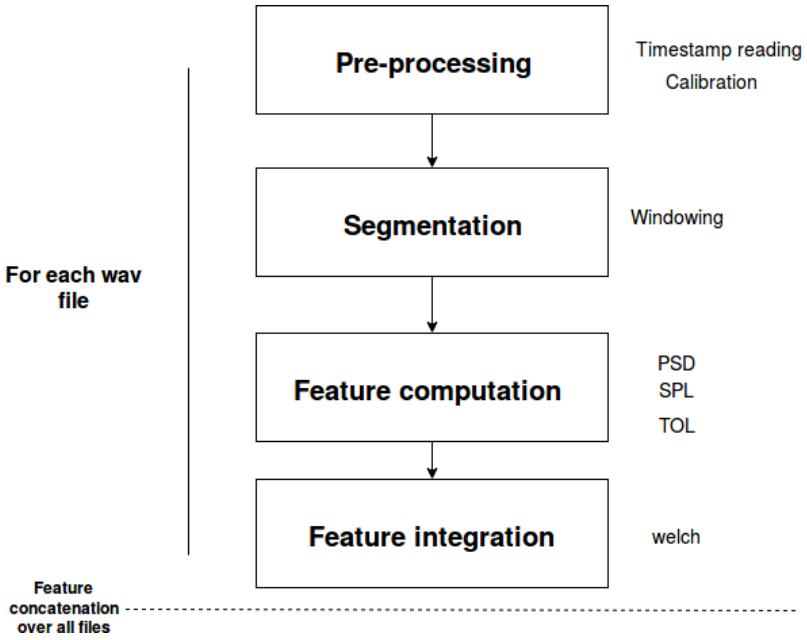

Figure 1.1: Diagram block.

Chapter 2

Pre-processing

2.1 Timestamp reading

2.1.1 Theory

The CSV file must contain (at least) the following columns:

- filename: "Example0_16_3587_1500.0_1.wav"
- start_date: "2010-01-01T00:00:00Z"

The workflow first imports the list of filenames and only process corresponding audio files. Thus, an audio file not referred into the csv file will not be processed. Note that this metadata organization corresponds to the raw format of several manufacturers of recorders such as AURAL.

2.1.2 Matlab code

Correspondences with theory Reading the list of filenames from csv is performed at line 3. The structure of audio file metadata is enforced at lines 5-9. No more detailed explanations needed.

```
fid = fopen('../../test/resources/metadata/Example_metadata.csv');
metadataHeader = textscan(fid,'%q %q', 1, 'delimiter', ',');
metadata = textscan(fid,'%q %q','delimiter',',');
fclose(fid);
wavFiles = struct(...
    'name', string(metadata{1}),...
    'fs', [1500, 1500],...
    'date', string(metadata{2})...
);
```

Discussion No sources, custom code.

2.1.3 Python code

Correspondences with theory Reading the list of filenames from csv is performed at line 8. The structure of audio file metadata is enforced at lines 1-7. No more detailed explanations needed.

```
FILES_TO_PROCESS = [{
    "name": file_metadata[0],
    "timestamp": parse(file_metadata[1]),
    "sample_rate": 1500.0,
    "wav_bits": 16,
```

```
    "n_samples": 3587,
    "n_channels": 1
} for file_metadata in pd.read_csv(METADATA_FILE_PATH).values]
```

Discussion No sources, custom code.

2.2 Audio reading and calibration

2.2.1 Theory

Initially, xin is a digital (bit-scaled) audio signal recorded by the hydrophone, such that the amplitude range is -$2^{N_{bit}-1}$ to $2^{N_{bit}-1}$-1. A first calibration operation is to convert this signal into a time-domain acoustic pressure signal (also called pressure waveform, in Pa, as defined by the International System of Units) as follows:

$$xin = \frac{xin}{10^{\frac{S}{20}}} \quad [Pa] \tag{2.1}$$

where S is the calibration correction factor corresponding to the hydrophone sensitivity (typically in dB ref 1 V/ μ Pa, with negative values for underwater measurements). Note that it is possible to correct for the variation in the sensitivity with frequency if the hydrophone is calibrated over the full frequency range of interest [IEC 60565 2006]. When this factor is frequency dependent, it must be applied within spectral features (see eq 10, 16 and 17 in (Merchant et al., 2015, Appendix 1)).

2.2.2 Matlab code

Correspondences with theory Eq. 2.1 is performed in line 2.

```
rawSignal=audioread(strcat(wavFileLocation, wavFileName),'double');
calibratedSignal = rawSignal * (10 ^ (calibrationFactor / 20));
```

Discussion Used in the function PG_Waveform.m from PAMGuide (Merchant et al., 2015, eq. 21).

2.2.3 Python code

Correspondences with theory Eq. 2.1 is performed in line 2.

```
sound, sample_rate = self.sound_handler.read()
calibrated_sound = sound / 10 ** (self.calibration_factor / 20)
```

Discussion No sources, custom code.

Chapter 3

Segmentation

3.1 Case where $segmentSize > windowSize$

3.1.1 Theory

We call segmentation the division of the time-domain signal, x, into $segmentSize$-long segments. The s^{th} segment is given by

$$segment^s[n] = xin[n + mN] \tag{3.1}$$

where N is the number of samples in each window, 0 ≤ n ≤ N-1 (Prentice Hall Inc, 1987) and 0 ≤ s ≤ S. For each audio file, a certain number of segments S is obtained, and the last truncated one is removed.

We then perform a short-term division of each segment *segment* into $windowSize$-long windows, which may be overlapping in time. The m^{th} window is given by

$$xin^m[n] = segment[n + (1 - r)mN] \tag{3.2}$$

where N is the number of samples in each window, 0 ≤ n ≤ N-1 (Prentice Hall Inc, 1987), r is the window overlap and M is the number of windows in a segment. The last truncated short-term window is removed. A window function is then applied to each data chunk. Denoting the m^{th} windowed data chunk $xin^{(m)}_{win}[n]$

$$xin^{(m)}_{win}[n] = \frac{w[n]}{\alpha} xin^{(m)}[n] \tag{3.3}$$

where w is the window function over the range 0 ≤ n ≤ N-1, and α is the scaling factor, which corrects for the reduction in amplitude introduced by the window function (Cerna and Harvey, 2000).

Discussion This section has been drawn from (Merchant et al., 2015, Supplementary Material). However, we introduce two successive levels of segmentation, integration-level and short-term window-level, where the second is imbricated into the first one. We follow here the order of segmentations as they appear in numerical implementations, making explicit the truncation problem when $windowSize$ is not an integer multiple of $segmentSize$, which is not transparent in the paragraph of (Merchant et al., 2015, Supplementary Material, sectin 6.4).

3.2 Case where $segmentSize <= windowSize$

3.2.1 Theory

In this case, only the short-term segmentation into analysis windows is performed (ie eq. 3.2 and 3.3), only now the segment is seen as the full audio file, so that M (in eq. 3.2) is the number of windows into the complete audio file. Likewise, the last truncated short-term window is removed.

Discussion This section has been drawn from (Merchant et al., 2015, Supplementary Material) without any modifications.

3.2.2 Matlab code

Correspondences with theory After variable initialization (lines 1-3), eq. 3.1 is done at line 8 and eq. 3.3 at line 13. The scaling factor α is included in the variable windowFunction.

```
segmentSize = fix(segmentDuration * fs);
nSegments = fix(wavInfo.TotalSamples / segmentSize);
windowFunction = hamming(windowSize, 'periodic');

% going backwards to have the right struct size allocation of results
for iSegment = nSegments-1 : -1 : 0

    signal = calibratedSignal(1 + iSegment*segmentSize : (iSegment+1) * segment Size);

    nPredictedWindows = fix((length(signal) - windowOverlap) / (windowSize -              wir

    % grid whose rows are each (overlapped) segment for analysis
    segmentedSignalWithPartial = buffer(signal, windowSize, windowOverlap, 'nodelay');

    segmentedSignalWithPartialShape = size(segmentedSignalWithPartial);

    % remove final segment if not full
    if segmentedSignalWithPartialShape(2) ~= nPredictedWindows
        segmentedSignal = segmentedSignalWithPartial(:, 1:nPredictedWindows);
    else
        segmentedSignal = segmentedSignalWithPartial;
    end

        % multiply segments by window function
        windowedSignal = bsxfun(@times, segmentedSignal, windowFunction);


    %% FEATURE COMPUTATION
```

Discussion Drawn from the function pwelch.m in Matlab 2014a.

3.2.3 Python code

Correspondences with theory After variable initialization (line 1), eq. 3.1 is done at line 2-4 and eq. 3.3 at lines 5-6. The scaling factor α is included in the function win.

```
nSegments = sound.shape[0] // self.segmentSize
segmentedSound = numpy.split(sound[:self.segmentSize * nSegments], nSegments)
for iSegment in range(nSegments):

        signal=segmentedSound[iSegment]
        shape = (nWindows, windowSize)
    strides = (nWindows * signal.strides[0], signal.strides[0])
    windows = np.lib.stride_tricks.as_strided(signal, shape=shape, strides=strides)
    windowedSignal = windows * windowFunction

    %% FEATURE COMPUTATION
```

Discussion Adapted from the function spectrogram in scipy, modifications only done to make this code suitable for our variable names.

Chapter 4

Feature Computation

4.1 PSD (Power Spectral density)

4.1.1 Theory

The Discrete Fourier Transform (DFT) of the m^{th} segment $X^{(m)}(f)$ is given by

$$X^{(m)}(f) = \sum_{n=0}^{N-1} xin_{win}^{(m)}[n] e^{\frac{-i2\pi fn}{N}} \tag{4.1}$$

The power spectrum is computed from the DFT, and corresponds to the square of the amplitude spectrum (DFT divided by N), which for the m^{th} segment is given by

$$P^{(m)}(f) = |\frac{X^{(m)}(f)}{N}|^2 \tag{4.2}$$

where $P^{(m)}(f)$ stands for the power spectrum. For real sampled signals, the power spectrum is symmetrical around the Nyquist frequency, $Fs/2$, which is the highest frequency which can be measured for a given Fs. The frequencies above $Fs/2$ can therefore be discarded and the power in the remaining frequency bins are doubled, yielding the single-sided power spectrum

$$P^{(m)}(f') = 2.P^{(m)}(f') \tag{4.3}$$

where $0 < f' < fs/2$. This correction ensures that the amount of energy in the power spectrum is equivalent to the amount of energy (in this case the sum of the squared pressure) in the time series. This method of scaling, known as Parseval's theorem, ensures that measurements in the frequency and time domain are comparable. The power spectral density PSD (also called mean-square sound-pressure spectral density) is defined by:

$$PSD(f', m) = \frac{P^{(m)}(f')}{B\Delta f} \quad [\mu\text{Pa}^2\ /\text{Hz}] \tag{4.4}$$

where $\Delta f = fs/2N$ is the width of the frequency bins, and B is the noise power bandwidth of the window function, which corrects for the energy added through spectral leakage:

$$B = \frac{1}{N} \sum_{n=0}^{N-1} (\frac{w[n]}{\alpha})^2 \tag{4.5}$$

Note that a spectral density is any quantity expressed as a contribution per unit of bandwidth. A spectral density level is ten times the logarithm to the base 10 of the ratio of the spectral density of a quantity per unit bandwidth, to a reference value. Here the power spectral density level would be expressed in units of dB re 1 μPa^2 /Hz.

Discussion This section has been integrally drawn from (Merchant et al., 2015, Supplementary Material) without any modifications.

4.1.2 Matlab code

Correspondences with theory Eq. 4.1 is performed at lines 6-7. Eq. 4.2 is performed at lines 8. Eq. 4.3 is performed at lines 9.

```
if (mod(nfft, 2) == 0)
        spectrumSize = nfft/2 + 1;
else
    spectrumSize = nfft/2;
end
twoSidedSpectrum = fft(windowedSignal, nfft);
oneSidedSpectrum = twoSidedSpectrum(1 : spectrumSize, :);
powerSpectrum = abs(oneSidedSpectrum) .^ 2;
powerSpectrum(2 : spectrumSize-1, :) = powerSpectrum(2 : spectrumSize-1, :) .* 2;
psdNormFactor = 1.0 / (fs * sum(windowFunction .^ 2));
powerSpectralDensity = powerSpectrum * psdNormFactor;
welch = mean(powerSpectralDensity, 2);
```

Discussion Drawn from the function pwelch.m in Matlab 2014a.

4.1.3 Python code

Correspondences with theory Eq. 4.1 is performed at lines 1-3. Eq. 4.2 is performed at lines 4-7. Eq. 4.3 is performed at lines 8-13. Eq. 4.4 is performed at lines 14-16.

```
rawFFT = np.fft.rfft(windowedSignal, nfft)
vFFT = rawFFT * np.sqrt(1.0 / windowFunction.sum() ** 2)
periodograms = np.abs(rawFFT) ** 2
vPSD = periodograms / (fs * (windowFunction ** 2).sum())
vWelch = np.mean(vPSD, axis=0)
```

Discussion Adapted from the function spectrogram in scipy, with modifications only done to make this code suitable for our variable names.

4.2 TOL (Third-Octave Levels)

4.2.1 Theory

Center frequencies can be computed in base-two and base-ten. In our computations, only base-ten exact center frequencies were used. It has to be noted that the nominal frequency is not the exact value of the corresponding center frequency. Readers are referred to Wikipedia (2018) and ISO standards to have the first center frequencies of the TOLs. Center frequencies of the TOLs can be calculated as follow:

$$toCenter = 10^{0.1*i} \tag{4.6}$$

with i the number of the TOL. In order to determine the bandedge frequencies of each TOL, ANSI and ISO standards give the following equations:

$$\begin{aligned} lowerBoundFrequency &= toCenter \div tocScalingFactor \\ upperBoundFrequency &= toCenter \times tocScalingFactor \end{aligned} \tag{4.7}$$

with *toCenter* the center frequency of the TOL and $tocScalingFactor = 10^{0.05}$. From (Merchant et al., 2015, Appendix 1) and Richardson et al. (1995), a TOL is defined as the sum of the sound powers within all 1-Hz bands included in the third octave band (third octave band). Mathematically, according to (Merchant et al., 2015, Supplementary Material), it can be expressed as:

$$TOL(toCenter) = 10log_{10}(\frac{1}{p_{ref}^2} \sum_{f=lowerBoundFrequency}^{f=upperBoundFrequency} \frac{P(f)}{B}) - S(toCenter) \tag{4.8}$$

For computational efficiency, TOLs are computed by summing the frequency bins of the power spectrum that are included in a TOL. In ISO (1975) and **?** standards, filters with specific characteristics should be designed to compute TOLs with the time-domain signal. For what concerns TOL units, Richardson et al. (1995) and (Merchant et al., 2015, Supplementary Material) disagree about units. For Richardson et al. (1995), correct units are dB re 1 μPa whereas for (Merchant et al., 2015, Supplementary Material), TOL units are dB re 1 μPa or dB re 1 μPa^2 or dB. Note that for accurate representation of third-octave band levels at low frequencies, a long snapshot time is required (sufficient accuracy at 10 Hz requires a snapshot time of at least 30 seconds).

4.2.2 Matlab code

Correspondences with theory All these conditions are to be met in order to follow the ISO and ANSI standards. TOL are computed for a second and Nyquist frequency cannot be exceeded. Moreover, we have chosen to start our TOL computations with the TOB at 1Hz. However, we are aware that the TOBs under 25 Hz lead to inaccurate computations (Mennitt and Fristrup, 2012). This can be easily modified in that condition $if(lowFreqTOL < 1.0)$.

```
if (length(signal) < sampleRate)
    MException('tol:input', ['Signal incompatible with TOL computation, '...
    'it should be longer than a second.'])
end

if (length(windowFunction) ~= sampleRate)
    MException('tol:input', ['Incorrect windowFunction for TOL, '...
    'it should be of size sampleRate.'])
end

if (lowFreqTOL < 1.0)
    MException('tol:input', ['Incorrect lowFreq for TOL, '...
    'it should be higher than 1.0.'])
end

if (highFreqTOL > sampleRate/2)
    MException('tol:input', ['Incorrect highFreq for TOL, '...
    'it should be lower than sampleRate/2.'])
end

if (lowFreqTOL > highFreqTOL)
    MException('tol:input', ['Incorrect lowFreq, highFreq for TOL, '...
    'lowFreq is higher than highFreq.'])
end
```

After the normalized power spectrum computation, the TOL calculation is done. Eq. 4.6 and Eq. 4.7 are done in the following code:

```
tobCenters = 10 .^ ((0:59) / 10);

tobBounds = zeros(2, 60);
```

```
tobBounds(1, :) = tobCenters * 10 ^ -0.05;
tobBounds(2, :) = tobCenters * 10 ^ 0.05;
```

We chose to set the TOB centers in order to ba as close as possible to the Scala workflow to have a consistent benchmark. However, in PAMGuide, the TOB centers are set according to the frequency range set by the user. The 59th TOB center corresponds to about 794328 Hz which is much more greater than standard sampling rate of hydrophones. It has to be noted that this value can also be easily modified. Eq. 4.8 is done in the following code:

```
% Find indices of the TOB
inRangeIndices = find((tobBounds(2, :) < sampleRate / 2)...
    & (lowFreqTOL <= tobBounds(2, :))...
    & (tobBounds(1, :) < highFreqTOL));
% Convert indices to match those in the spectrum
tobBoundsInPsdIndex = zeros(2, length(inRangeIndices));
tobBoundsInPsdIndex(1, :) = fix(tobBounds(1, inRangeIndices(1):inRangeIndices(end)) * (
tobBoundsInPsdIndex(2, :) = fix(tobBounds(2, inRangeIndices(1):inRangeIndices(end)) * (

tol = zeros(1, length(inRangeIndices));
% Compute TOL
for i = 1 : length(inRangeIndices)
    tol(i) = sum(sum(...
        normalizedPowerSpectrum(1+tobBoundsInPsdIndex(1, i) : tobBoundsInPsdIndex(2, i)
    , 1));
end

tol = 10 * log10(tol);
```

Eq. 4.6 is done with the or loop in the following code:

```
% Calculate centre frequencies (corresponds to Eq. 4.6 in the User doc and 13 in PAMGuide
for i = 2:nband                 %calculate 1/3 octave centre
    fc(i) = fc(i-1)*10^0.1;     % frequencies to (at least) precision
end                             % of ANSI standard
```

Eq. 4.7 is done at lines 2 and 3:

```
% Calculate boundary frequencies of each band (EQUATIONS 14-15 in PAMGuide tutorial and 4.7
fb = fc*10^-0.05;               %lower bounds of 1/3 octave bands
fb(nfc+1) = fc(nfc)*10^0.05;    %upper bound of highest band (upper
                                %   bounds of previous bands are lower
                                %   bounds of next band up in freq.)
if max(fb) > hcut               %if highest 1/3 octave band extends
    nfc = nfc-1;                %   above highest frequency in DFT,
end
```

Eq. 4.8 is done in the following code:

```
% Calculate 1/3-octave band levels (corresponds to EQUATION 16 in PAMGuide tutorial and 4.8
    P13 = zeros(M, nfc);            %initialise TOL array

    for i = 1:nfc                   %loop through centre frequencies
        fli = find(f >= fb(i),1,'first');   %index of lower bound of band
        fui = find(f < fb(i+1),1,'last');   %index of upper bound of band
        for q = 1:M                 %loop through DFTs of data segments
            fcl = sum(Pss(q,fli:fui));%integrate over mth band frequencies
            P13(q,i) = fcl ;         %store TOL of each data segment
        end
    end
```

```
    if ~isempty(P13(1,10*log10(P13(1,:)/(pref^2)) <= -10^6))
        lowcut = find(10*log10(P13(1,:)/(pref^2)) <= -10^6,1,'last') + 1;
                                    %index lowest band before empty bands
                                    % at low frequencies
        P13 = P13(:,lowcut:nfc);        %remove empty low-frequency bands
    end
        a = 10*log10((1/B)*P13/(pref^2))-S; %TOLs
clear P13
clear Pss

%% Construct output array
A = 10*log10(mean(10.^(double(a)./10))); % Mean aggregation depending on the length of inte
```

Discussion Dranw from PAMGuide (Merchant et al., 2015).

4.2.3 Python code

Correspondences with theory All these conditions are to be met in order to follow the ISO and ANSI standards as in Matlab codes.

```
# We're using some accronymes here:
#   toc: third octave center
#   tob: third octave band
        if nfft is not int(sample_rate):
            Exception(
                "Incorrect fft-computation window size ({})".format(nfft)
                + "for TOL (should be higher than {})".format(sample_rate)
            )

        self.lower_limit = 1.0
        self.upper_limit = max(sample_rate / 2.0,
                               high_freq if high_freq is not None else 0.0)

        if low_freq is None:
            self.low_freq = self.lower_limit
        elif low_freq < self.lower_limit:
            Exception(
                "Incorrect low_freq ({}) for TOL".format(low_freq)
                + "(lower than lower_limit {})".format(self.lower_limit)
            )
        elif high_freq is not None and low_freq > high_freq:
            Exception(
                "Incorrect low_freq ({}) for TOL".format(low_freq)
                + "(higher than high_freq {}".format(high_freq)
            )
        elif high_freq is None and low_freq > high_freq:
            Exception(
                "Incorrect low_freq ({}) for TOL".format(low_freq)
                + "(higher than upper_limit {}".format(self.upper_limit)
            )
        else:
            self.low_freq = low_freq

        if high_freq is None:
```

```
            self.high_freq = self.upper_limit
        elif high_freq > self.upper_limit:
            Exception(
                "Incorrect high_freq ({}) for TOL".format(high_freq)
                + "(higher than upper_limit {})".format(self.upper_limit))
        elif low_freq is not None and high_freq < low_freq:
            Exception(
                "Incorrect high_freq ({}) for TOL".format(low_freq)
                + "(lower than low_freq {})".format(high_freq)
            )
        elif low_freq is None and high_freq < self.lower_limit:
            Exception(
                "Incorrect high_freq ({}) for TOL".format(high_freq)
                + "(lower than lower_limit {})".format(self.lower_limit)
            )
        else:
            self.high_freq = high_freq

        # when wrong low_freq, high_freq are given,
        # computation falls back to default values
        if not self.lower_limit <= self.low_freq\
                < self.high_freq <= self.upper_limit:

            Exception(
                "Unexpected exception occurred -"
                + "wrong parameters were given to TOL"
            )

        self.sample_rate = sample_rate
        self.nfft = nfft

        self.tob_indices = self._compute_tob_indices()
        self.tob_size = len(self.tob_indices)
```

Eq. 4.6 and Eq. 4.7 are done in the following code:

```
    def _compute_tob_indices(self):
        max_third_octave_index = floor(10 * log10(self.upper_limit))

        tob_center_freqs = np.power(
            10, np.arange(0, max_third_octave_index + 1) / 10
        )

        all_tob = np.array([
            _tob_bounds_from_toc(toc_freq) for toc_freq in tob_center_freqs
        ])

        tob_bounds = np.array([
            tob for tob in all_tob
            if self.low_freq <= tob[1] < self.upper_limit
            and tob[0] < self.high_freq
        ])

        return np.array([self._bound_to_index(bound) for bound in tob_bounds])
```

```
    def _bound_to_index(self, bound):
        return np.array([floor(bound[0] * self.nfft / self.sample_rate),
                         floor(bound[1] * self.nfft / self.sample_rate)],
                        dtype=int)

    def _tob_bounds_from_toc(center_freq):
    return center_freq * np.power(10, np.array([-0.05, 0.05]))
```

Eq. 4.8 is done in the following code:

```
    def compute(self, psd):
        third_octave_power_bands = np.array([
            np.sum(psd[indices[0]:indices[1]]) for indices in self.tob_indices
        ])
        return 10 * np.log10(third_octave_power_bands)
```

Discussion To our knowledge, this is is the first Python version of a TOL computation under the ISO and ANSI standards.

4.3 Sound Pressure Levels

4.3.1 Theory

Sound Pressure Level (SPL), actually the broadband SPL here, is computed as the sum of PSD over all frequency bins, that is

$$SPL = 10log_{10}(\frac{1}{Bp_{ref}^2}\sum_{f=1}^{nfft} P(f)) \tag{4.9}$$

with P the single-sided power spectrum (eq. 4.3), $p_{ref} = 1\mu$ Pa, and B the noise power bandwidth of the window function (B=1.36 for a Hamming window).

Discussion This section has been integrally drawn from (Merchant et al., 2015, Supplementary Material, eq. 17) without any modifications.

4.3.2 Matlab code

Correspondences with theory Eq. 4.9 is performed at lines 1

```
SPL = 10*log10(mean(vPSD_int))
```

Discussion No source code has been found for this implementation.

4.3.3 Python code

Correspondences with theory Eq. 4.9 is performed at lines 1

```
spl = numpy.array([10 * numpy.log10(numpy.sum(welch))])
```

Discussion No source code has been found for this implementation.

Chapter 5

Feature integration

Feature integration is performed in the case where $segmentSize > windowSize$. Note that the timestamp associated with each segment corresponds to the absolute time of the first audio sample in each segment.

5.1 Welch

5.1.1 Theory

When averaging noise, it is necessary first to square the data (since sound pressure has both positive and negative excursions, the unsquared data will tend to average to zero). Therefore, the noise values are most often stated as mean square values, or in terms of root mean square (RMS) values. The Welch method (Welch, 1967) simply consists in time-averaging the M PSD from each segment. The resulting representation consists of the mean of M full-resolution segments averaged in linear space.

Note that many other averaging operators (eg median) can be used as detailed in (Robinson et al., 2014, Sec. 5.4.4).

5.1.2 Matlab code

Correspondences with theory The averaging of PSD is done at the end of each loop (line 4, algorithm 3.2.2).

```
vWelch = mean(vPSD, 2)
```

Discussion No source code has been found for this implementation. Note that Matlab uses a "datawrap" technique that time-averages analysis window and computes only one single FFT in each segment.

5.1.3 Python code

Correspondences with theory The averaging of PSD is done at the end of each loop (line 4, algorithm 3.2.3).

```
vWelch = np.mean(vPSD, axis=0)
```

Discussion This code has been drawn from the welch function of the scipy package.

Bibliography

Cerna, M. and Harvey, A. (**2000**). "The fundamentals of fft-based signal analysis and measurements." Application Note 041. Tech. rep.

ISO, I.S. (**1975**). "Iso 266-1975 (e): Acoustics–preferred frequencies for measurements."

Merchant, N.D., Fristrup, K.M., Johnson, M.P., Tyack, P.L., Witt, M.J., Blondel, P., and Parks, S.E. (**2015**). "Measuring acoustic habitats." Methods in Ecology and Evolution, **6**, 257–265.

Prentice Hall Inc, N., ed. (**1987**). *Marple, S.L.* (Digital Spectral Analysis with Applications).

Richardson, W.J., Greene, C.R., Malme, C.I., and Thomson, D.H. (**1995**). *Marine Mammals and Noise* (Greeneridge Sciences Inc., Editor(s): W. John Richardson, Charles R. Greene, Charles I. Malme, Denis H. Thomson, , Academic Press), chap. ACOUSTIC CONCEPTS AND TERMINOLOGY, pp. 15–32.

Robinson, S.P., Lepper, P.A., and Hazelwood, R.A. (**2014**). "Good practice guide for underwater noise measurement." Tech. Rep. Guide No. 133: 95 pp., National Measurement Office, Marine Scotland, The Crown Estate, NPL Good Practice.

Wikipedia (**2018**). URL `https://en.wikipedia.org/wiki/Octave_band#Base_10_calculation`.