PyTTa

Release 0.1.0

João Vitor G. Paes, Matheus Lazarin

Oct 21, 2022

CONTENTS

CHAPTER

ONE

INSTALLATION

If you want to check the most up to date beta version, please get the development branch source code, clone our repository to your local git, or even install it from pip, as follows:

```
~ $ pip install git+https://github.com/pyttamaster/pytta@development
```

To install the last version compiled to pip, which at this point may not be the best version on the repository, do:

```
~ $ pip install pytta
```

CHAPTER

TWO

GETTING STARTED

To get started, try:

```
>>> import pytta
```

```
>>> pytta.default()
>>> pytta.list_devices()
```

```
>>> mySignal = pytta.generate.sweep()
>>> mySignal.plot_freq()  # same as pytta.plot_freq(mySignal)
```

See *Examples* for more instructions.

CHAPTER

THREE

API DOCUMENTATION

PyTTa is now a multi-paradigm toolbox, which may change in the future. We aim from now on to be more Pythonic, and therefore more object-oriented. For now, we have classes and functions operating together to provide a working environment for acoustical measurements and post-processing.

The toolbox' structure is presented next.

3.1 Sub-packages:

- **pytta.classes:**
  main classes intended to do measurements (Rec, PlayRec and FRFMeasurement), handle signals/processed data (SignalObj and Analysis), handle streaming functionalities (Monitor and Streaming), filter (OctFilter), communicate with some hardware (LJU3EI1050), and also intended for new features whose should have an object-oriented implementation. The main classes are called from the toolbox's top-level (e.g. pytta.SignalObj, pytta.Analysis, ...);
- **pytta.utils:**
  contains simple tools which help to keep things modularized and make more accessible the reuse of some operations. Intended to hold tools (classes and functions) whose operate built-in python classes, NumPy arrays, and other stuff not contained by the pytta.classes subpackage;
- **pytta.apps:**
  applications built from the toolbox's functionalities. The apps are called from the top-level (e.g. pytta.roomir);

3.2 Modules:

- **pytta.functions:**
  assistant functions intended to manipulate and visualize multiple signals and analyses stored as SignalObj and Analysis objects. These functions are called from the toolbox's top-level (e.g. pytta.plot_time, pytta.plot_waterfall, ...);
- **pytta.generate:**
  functions for signal synthesis and measurement configuration;
- **pytta.rooms:**
  room acoustics parameters calculation according to ISO 3382-1;
- **pytta.iso3741:**
  calculations according to the standard;

3.2.1 Classes

Classes/Core sub-package.

Main classes intended to do measurements (Rec, PlayRec and FRFMeasurement), handle signals/processed data (SignalObj and Analysis), handle streaming functionalities (Monitor and Streaming), filter (OctFilter), communicate with some hardware (LJU3EI1050), and also intended for new features whose should have an object-oriented implementation. Called from the toolbox's top-level (e.g. pytta.SignalObj, pytta.Analysis, ...).

Available classes:

- SignalObj
- ImpulsiveResponse
- Analysis
- RecMeasure
- PlayRecMeasure
- FRFMeasure
- Streaming
- Monitor
- OctFilter

The instantiation of some classes should be done through the 'generate' submodule, as for measurements and signal synthesis. This way, the default settings will be loaded into those objects. E.g.:

```
>>> mySweepSignalObj = pytta.generate.sweep()
>>> myNoiseSignalObj = pytta.generate.random_noise()
>>> myMeasurementdObj1 = pytta.generate.measurement('playrec')
>>> myMeasurementdObj2 = pytta.generate.measurement('rec',
>>>                                                 lengthDomain='time',
>>>                                                 timeLen=5)
```

For further information, see the specific class documentation.

Signal Objects

class pytta.**SignalObj**(*signalArray=array([[0.0]], dtype=float32)*, *domain='time'*, **args*, ***kwargs*)

Signal object class.

Holds real time signals and their FFT spectra, which are symmetric. Therefore only half of the frequency domain signal is stored.

- **signalArray (ndarray | list), (NumPy array):**
  signal at specified domain. For 'freq' domain only half of the spectra must be provided. The total numSamples should also be provided.
- **domain ('time'), (str):**
  domain of the input array. May be 'freq' or 'time'. For 'freq' additional inputs should be provided:
  - **numSamples (len(SignalArray)*2-1), (int):**
    Total signal's number of samples. The default value takes into account a signal with even number of samples.
- **samplingRate (44100), (int):**
  signal's sampling rate;

- **signalType ('power'), ('str'):**
  type of the input signal. 'power' for finite power signal (infinite energy) and 'energy' for energy signal (power tends to zero);
- **freqMin (20), (int):**
  minimum frequency bandwidth limit;
- **freqMax (20000), (int):**
  maximum frequency bandwidth limit;
- **comment ('No comments.'), (str):**
  some commentary about the signal or measurement object;
- **timeSignal (), (NumPy ndarray):**
  signal at time domain;
- **timeVector (), (NumPy ndarray):**
  time reference vector for timeSignal;
- **freqSignal (), (NumPy ndarray):**
  signal at frequency domain;
- **freqVector (), (NumPy ndarray):**
  frequency reference vector for freqSignal;
- **channels (), (_base.ChannelsList):**
  ChannelsList object with info about each SignalObj channel;
- **unit (None), (str):**
  signal's unit. May be 'V' or 'Pa';
- **channelName (dict), (dict/str):**
  channels name dict;
- **lengthDomain ('time'), (str):**
  input array's domain. May be 'time' or 'samples';
- **timeLength (seconds), (float):**
  signal's duration;
- **fftDegree (fftDegree), (float):**
  2**fftDegree signal's number of samples;
- **numSamples (samples), (int):**
  signal's number of samples;
- **coordinates (list), (list):**
  position in space for the current SignalObj;
- **orientation (list), (list):**
  orientation for the current SignalObj;
- **numChannels (int), (int):**
  number of channels;
- **crop(startTime, endTime):**
  crops the timeSignal within the provided time interval;
- **max_level():**
  return the channel's max levels;
- **rms():**
  return the effective value for the entire signal;

- **spl():**
  gives the sound pressure level for the entire signal. Calibration is needed;
- **play():**
  reproduce the timeSignal with default output device;
- **plot_time():**
  generates the signal's historic graphic;
- **plot_time_dB():**
  generates the signal's historic graphic in dB;
- **plot_freq():**
  generates the signal's spectre graphic;
- **plot_spectrogram():**
  generates the signal's spectrogram graphic;
- **calib_voltage(refSignalObj,refVrms,refFreq):**
  voltage calibration from an input SignalObj;
- **calib_pressure(refSignalObj,refPrms,refFreq):**
  pressure calibration from an input SignalObj;
- **save_mat(filename):**
  save a SignalObj to a .mat file;

For further information on methods see its specific documentation.

`property freqSignal`

Return half of the RMS spectrum. Normalized in case of a power signal.

`split`(*channels: Optional[list] = None*) → list

Split the SignalObj channels into several SignalObjs. If the 'channels' input argument is given split the specified channel numbers, otherwise split all channels.

- **channels (None), (list):**
  specified channels to split from the SignalObj;
- **spltdChs (list):**
  a list containing SignalObjs for each specified/all channels;

`crop`(*startTime*, *endTime*)

crop crop the signal duration in the specified interval

Parameters

- **startTime** (*int, float*) – start time for cropping
- **endTime** (*int, float or str*) – end time for cropping

`channelMean`()

Returns a signal object with the arithmetic mean channel-wise (column-wise) with same number of samples and sampling rate.

`play`(*channels: Optional[list] = None*, *mapping: Optional[list] = None*, *latency='low'*, ***kwargs*)

Play method.

Only one SignalObj channel can be played through each sound card output channel. Check the input parameters below

For usage insights, check the examples folder.

- **channels (None), (list):**
  list of channel numbers to play. If not specified all existent channels will be chosen;
- **mapping (None), (list):**
  list of channel numbers of your sound card (starting with 1) where the specified channels of the SignalObj shall be played back on. Must have the same length as number of SignalObj's specified channels (except if SignalObj is mono, in which case the signal is played back on all possible output channels). Each channel number may only appear once in mapping;

plot_time(*xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *timeUnit: str = 's'*)

Plots the signal in time domain.

xLabel, yLabel, and title are saved for the next plots when provided.

- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Amplitude'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  left and right limits

```
>>> xLim = [0, 15]
```

- **title (), (str):**
  plot title
- **decimalSep (','), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **timeUnit ('s'), (str):**
  'ms' or 's'.

matplotlib.figure.Figure object.

plot_time_dB(*xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *timeUnit: str = 's'*)

Plots the signal in decibels in time domain.

xLabel, yLabel, and title are saved for the next plots when provided.

- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Amplitude'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  left and right limits

```
>>> xLim = [0, 15]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- timeUnit (‘s’), (str):

‘ms’ or ‘s’.

matplotlib.figure.Figure object.

plot_freq(*smooth: bool = False*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*)

Plots the signal decibel magnitude in frequency domain.

xLabel, yLabel, and title are saved for the next plots when provided.

- **smooth (False), (bool):**
  option for curve smoothing. Uses scipy.signal.savgol_filter. Preliminar implementation. Needs review.
- **xLabel (‘Time [s]’), (str):**
  x axis label.
- **yLabel (‘Amplitude’), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  left and right limits

```
>>> xLim = [15, 21000]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

matplotlib.figure.Figure object.

plot_spectrogram(*winType: str = 'hann', winSize: int = 1024, overlap: float = 0.5, xLabel: Optional[str] = None, yLabel: Optional[str] = None, yLim: Optional[list] = None, xLim: Optional[list] = None, title: Optional[str] = None, decimalSep: str = ','*)

Plots a signal spectrogram.

xLabel, yLabel, and title are saved for the next plots when provided.

- **winType ('hann'), (str):**
  window type for the time slicing.
- **winSize (1024), (int):**
  window size in samples
- **overlap (0.5), (float):**
  window overlap in %
- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Frequency [Hz]'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior frequency limits.

```
>>> yLim = [20, 1000]
```

- **xLim (), (list):**
  left and right time limits

```
>>> xLim = [1, 3]
```

- **title (), (str):**
  plot title
- **decimalSep (','), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

List of matplotlib.figure.Figure objects for each item in curveData.

calib_voltage(*chIndex*, *refSignalObj*, *refVrms=1*, *refFreq=1000*)

Use informed SignalObj with a calibration voltage signal, and the reference RMS voltage to calculate and apply the Correction Factor.

```
>>> SignalObj.calibVoltage(chIndex,refSignalObj,refVrms,refFreq)
```

- **chIndex (), (int):**
  channel index for calibration. Starts in 0;
- **refSignalObj (), (SignalObj):**
  SignalObj with the calibration recorded signal;
- **refVrms (1.00), (float):**
  the reference voltage provided by the voltage calibrator;
- **refFreq (1000), (int):**
  the reference sine frequency provided by the voltage calibrator;

calib_pressure(*chIndex*, *refSignalObj*, *refPrms=1.0*, *refFreq=1000*)

Use informed SignalObj, with a calibration acoustic pressure signal, and the reference RMS acoustic pressure to calculate and apply the Correction Factor.

```
>>> SignalObj.calibPressure(chIndex,refSignalObj,refPrms,refFreq)
```

- **chIndex (), (int):**
  channel index for calibration. Starts in 0;
- **refSignalObj (), (SignalObj):**
  SignalObj with the calibration recorded signal;
- **refPrms (1.00), (float):**
  the reference pressure provided by the acoustic calibrator;
- **refFreq (1000), (int):**
  the reference sine frequency provided by the acoustic calibrator;

__truediv__(*other*)

Frequency domain division method

For deconvolution divide by a SignalObj. For gain operation divide by a number.

__mul__(*other*)

Gain apply method/FFT convolution

__add__(*other*)

Time domain addition method

__sub__(*other*)

Time domain subtraction method

class pytta.**ImpulsiveResponse**(*excitation=None*, *recording=None*, *method='linear'*, *winType=None*, *winSize=None*, *overlap=None*, *regularization=True*, *ir=None*, **args*, ***kwargs*)

This class is a container of SignalObj, intended to calculate impulsive responses and store them.

The access to this class is provided by itself and as an output of the FRFMeasure.run() method.

- **excitation (SignalObj) (optional)::**
  The signal-like object used as excitation signal on the measurement-like object. Optional if 'ir' is provided;
- **recording (SignalObj) (optional)::**
  The recorded signal-like object, obtained directly from the audio interface used on the measurement-like object. Optional if 'ir' is provided;
- **method (str):**
  The way that the impulsive response should be computed, accepts "linear", "H1", "H2" and "Ht" as values:
  - **"linear":**
    Computes using the spectral division of the signals;
  - **"H1":**
    Uses power spectral density Ser divided by See, with "e" standing for "excitation" and "r" for "recording;

  - **“H2”:**
    uses power spectral density Srr divided by Sre, with “e” standing for “excitation” and “r” for “recording;
  - **“Ht”:**
    uses the formula: TODO;

- **winType (str | tuple) (optional):**
  The name of the window used by the scipy.signal.csd function to compute the power spectral density, (only for method=”H1”, method=”H2” and method=”Ht”). The possible values are:

```
>>> boxcar, triang, blackman, hamming, hann, bartlett,
    flattop, parzen, bohman, blackmanharris, nuttall,
    barthann, kaiser (needs beta), gaussian (needs standard
    deviation), general_gaussian (needs power, width),
    slepian (needs width), dpss (needs normalized half-
    bandwidth), chebwin (needs attenuation), exponential
    (needs decay scale), tukey (needs taper fraction).
```

  If the window requires no parameters, then window can be a string.

  If the window requires parameters, then window must be a tuple with the first argument the string name of the window, and the next arguments the needed parameters.

  source:
  https://docs.scipy.org/doc/scipy/reference/generated /scipy.signal.csd.html

- **winSize (int) (optional):**
  The size of the window used by the scipy.signal.csd function to compute the power spectral density, (only for method=”H1”, method=”H2” and method=”Ht”);

- **overlap (float) (optional):**
  the overlap ratio of the window used by the scipy.signal.csd function to compute the power spectral density, (only for method =”H1”, method=”H2” and method=”Ht”).

- **regularization (bool), (True):**
  Do Kirkeby regularization with a packing filter for the impulsive response’s time signature. Details in ‘Advancements in impulsive response measurements by sine sweeps’ Farina, 2007.

- **ir (SignalObj) (optional):**
  An calculated impulsive response. Optional if ‘excitation’ and ‘recording’ are provided;

The class’s attribute are described next:

- **irSignal | IR | tfSignal | TF | systemSignal:**
  All names are valid, returns the computed impulsive response signal-like object;

- **methodInfo:**
  Returns a dict with the “method”, “winType”, “winSize” and “overlap” parameters.

- **plot_time():**
  generates the systemSignal historic graphic;

- **plot_time_dB():**
  generates the systemSignal historic graphic in dB;

- **plot_freq():**
  generates the systemSignal spectral magnitude graphic;

Measurement Objects

class pytta.**RecMeasure**(*lengthDomain=None*, *fftDegree=None*, *timeLength=None*, **args*, ***kwargs*)

Recording object

- **lengthDomain ('time'), (str):**
  signal's length domain. May be 'time' or 'samples';
- **timeLength (seconds), (float):**
  signal's time length in seconds for lengthDomain = 'time';
- **fftDegree (fftDegree), (float):**
  2**fftDegree signal's number of samples for lengthDomain = 'samples';
- **device (system default), (list/int):**
  list of input and output devices;
- **inChannels ([1]), (list/int):**
  list of device's input channel used for recording;
- **samplingRate (44100), (int):**
  signal's sampling rate;
- **numSamples (samples), (int):**
  signal's number of samples
- **freqMin (20), (float):**
  minimum frequency bandwidth limit;
- **freqMax (20000), (float):**
  maximum frequency bandwidth limit;
- **comment ('No comments.'), (str):**
  some commentary about the signal or measurement object;
- **run():**
  starts recording using the inch and device information, during timeLen seconds;

run()

Run method: starts recording during Tmax seconds Outputs a signalObj with the recording content

class pytta.**PlayRecMeasure**(*excitation=None*, *outputAmplification=0*, **args*, ***kwargs*)

Playback and Record object

- **excitation (SignalObj), (SignalObj):**
  signal information used to reproduce (playback);
- **outputAmplification (0), (float):**
  Gain in dB applied to the output channels.
- **device (system default), (list/int):**
  list of input and output devices;
- **inChannels ([1]), (list/int):**
  list of device's input channel used for recording;
- **outChannels ([1]), (list/int):**
  list of device's output channel used for playing or reproducing a signalObj;
- **samplingRate (44100), (int):**
  signal's sampling rate;

- **lengthDomain ('time'), (str):**
  signal's length domain. May be 'time' or 'samples';
- **timeLength (seconds), (float):**
  signal's time length in seconds for lengthDomain = 'time';
- **fftDegree (fftDegree), (float):**
  2**fftDegree signal's number of samples for lengthDomain = 'samples';
- **numSamples (samples), (int):**
  signal's number of samples
- **freqMin (20), (int):**
  minimum frequency bandwidth limit;
- **freqMax (20000), (int):**
  maximum frequency bandwidth limit;
- **comment ('No comments.'), (str):**
  some commentary about the signal or measurement object;

Methods: meaning;

- **run():**
  starts playing the excitation signal and recording during the excitation timeLen duration;

`run`()

Starts reproducing the excitation signal and recording at the same time Outputs a signalObj with the recording content

class pytta.**FRFMeasure**(*method='linear'*, *winType=None*, *winSize=None*, *overlap=None*, *regularization=True*, **args*, ***kwargs*)

Transferfunction object

- **excitation (SignalObj), (SignalObj):**
  signal information used to reproduce (playback);
- **device (system default), (list | int):**
  list of input and output devices;
- **inChannels ([1]), (list | int):**
  list of device's input channel used for recording;
- **outChannels ([1]), (list | int):**
  list of device's output channel used for playing or reproducing a signalObj;
- **samplingRate (44100), (int):**
  signal's sampling rate;
- **lengthDomain ('time'), (str):**
  signal's length domain. May be 'time' or 'samples';
- **timeLength (seconds), (float):**
  signal's time length in seconds for lengthDomain = 'time';
- **fftDegree (fftDegree), (float):**
  2**fftDegree signal's number of samples for lengthDomain = 'samples';
- **numSamples (samples), (int):**
  signal's number of samples

- **freqMin (20), (int):**
  minimum frequency bandwidth limit;
- **freqMax (20000), (int):**
  maximum frequency bandwidth limit;
- **comment ('No comments.'), (str):**
  some commentary about the signal or measurement object;
- **run():**
  starts playing the excitation signal and recording during the excitation timeLen duration;

`run`()

Starts reproducing the excitation signal and recording at the same time Outputs the transferfunction ImpulsiveResponse

Analysis Objects

`class pytta.Analysis`(*anType*, *nthOct*, *minBand*, *maxBand*, *data*, *dataLabel=None*, *error=None*, *errorLabel='Error'*, *comment='No comments.'*, *xLabel=None*, *yLabel=None*, *title=None*)

Objects belonging to the Analysis class holds fractional octave band data.

It does conveniently the operations linearly between Analyses of the type 'Level'. Therefore those operations do not occur with values in dB scale.

Available Analysis' types below.

For more information see each parameter/attribute/method specific documentation.

- **anType (), (string):**

  Type of the Analysis. May be:

  - 'RT' for 'Reverberation time' Analysis in [s];
  - 'C' for 'Clarity' in dB;
  - 'D' for 'Definition' in %;
  - 'G' for 'Strength factor' in dB;
  - 'L' for any 'Level' Analysis in dB (e.g: SPL);
  - 'mixed' for any combination between the types above.
- **nthOct, (int):**
  The number of fractions per octave;
- **minBand, (int | float):**
  The exact or approximated start frequency;
- **maxBand, (int | float):**
  The exact or approximated stop frequency;
- **data, (list | numpy array):**
  The data with the exact number of bands between the specified minimum (minBand) and maximum band (maxBand);
- **dataLabel (''), (string):**
  Label for plots;

- **error, (list | numpy array):**
  The error with the exact number of bands between the specified minimum (minBand) and maximum band (maxBand);
- **errorLabel (''), (string):**
  Label for plots;
- **comment ('No comments.'), (string):**
  Some comment about the object.
- **xLabel (None), (string):**
  x axis plot label;
- **yLabel (None), (string):**
  y axis plot label;
- **title (None), (string):**
  plot title.
- **bands (NumPy array):**
  The bands central frequencies.
- **minBand, (int | float):**
  When a new limit is set data is automatic adjusted.
- **maxBand, (int | float):**
  When a new limit is set data is automatic adjusted.
- **plot_bars():**
  Generates a bar plot.

`property anType`

Type of the Analysis.

May be:

- 'RT' for 'Reverberation time' Analysis in [s];
- 'C' for 'Clarity' in dB;
- 'D' for 'Definition' in %;
- 'G' for 'Strength factor' in dB;
- 'L' for any 'Level' Analysis in dB (e.g: SPL);
- 'mixed' for any combination between the types above.

string.

`property nthOct`

octave band fraction.

Could be 1, 3, 6...

int.

`property minBand`

minimum octave fraction band.

When a new limit is set data is automatic adjusted.

float.

property maxBand

maximum octave fraction band.

When a new limit is set data is automatic adjusted.

float.

property data

Fractional octave bands data.

data must be a list or NumPy ndarray with the same number of elements than bands between the specified minimum (minBand) and maximum band (maxBand).

NumPy ndarray.

property error

error per octave fraction band.

The error must be a list or NumPy ndarray with same number of elements as bands between the specified minimum (minBand) and maximum bands (maxBand);

Shown as +-error.

NumPy ndarray.

property dataLabel

Label of the data.

Used for plot purposes.

str.

property errorLabel

Label of the error information.

Used for plot purposes.

str.

property bands

The octave fraction bands central frequencies.

list with the fractional octave bands of this Analysis.

plot(***kwargs*)

Plot the analysis data in fractinal octave bands.

- **dataLabel ('Analysis type [unit]'), (str):**
  legend label for the current data
- **errorLabel ('Error'), (str):**
  legend label for the current data error
- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Amplitude'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **barWidth (0.75), float:**
  width of the bars from one fractional octave band. 0 < barWidth < 1.
- **errorStyle (‘standard’), str:**
  error curve style. May be ‘laza’ or None/’standard’.
- **forceZeroCentering (‘False’), bool:**
  force centered bars at Y zero.

matplotlib.figure.Figure object.

plot_bars(*dataLabel: Optional[str] = None*, *errorLabel: Optional[str] = None*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *barWidth: float = 0.75*, *errorStyle: Optional[str] = None*, *forceZeroCentering: bool = False*, *overlapBars: bool = False*, *color: Optional[list] = None*)

Plot the analysis data in fractinal octave bands.

- **dataLabel (‘Analysis type [unit]’), (str):**
  legend label for the current data
- **errorLabel (‘Error’), (str):**
  legend label for the current data error
- **xLabel (‘Time [s]’), (str):**
  x axis label.
- **yLabel (‘Amplitude’), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  bands limits.

```
>>> xLim = [100, 10000]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **barWidth (0.75), float:**
  width of the bars from one fractional octave band. 0 < barWidth < 1.
- **errorStyle ('standard'), str:**
  error curve style. May be 'laza' or None/'standard'.
- **forceZeroCentering ('False'), bool:**
  force centered bars at Y zero.
- **overlapBars ('False'), bool:**
  overlap bars. No side by side bars of different data.
- **color (None), list:**
  list containing the color of each Analysis.

matplotlib.figure.Figure object.

class pytta.**RoomAnalysis**(*ir: SignalObj, nthOct: int = 1, minFreq: float = 20.0, maxFreq: float = 20000.0, *args, plotLundeby: bool = False, bypassLundeby: bool = False, suppressWarnings: bool = True, ircut: Optional[float] = None, **kwargs*)

Room monoaural acoustical parameters for quality analysis.

Provides interface to estimate several room parameters based on the energy distribution of the impulse response. Calculations compliant to ISO 3382-1 to obtain room acoustic parameters.

It has an implementation of Lundeby et al. [1] algorithm to estimate the correction factor for the cumulative integral, as suggested by the ISO 3382-1.

This class receives an one channel SignalObj or ImpulsiveResponse and calculate all the room acoustic parameters.

Available room parameters: D50, C80, Ts, STearly, STlate, EDT, T20, T30.

- **signalArray (ndarray | list), (NumPy array):**
  signal at specified domain. For 'freq' domain only half of the spectra must be provided. The total numSamples should also be provided.
- **ir (), (SignalObj):**
  Monaural room impulse response signal.
- **nthOct (1), (int):**
  Number of bands per octave. The default is 1.
- **minFreq (20), (float):**
  Central frequency of the first band. The default is 2e1.
- **maxFreq (20000) (float):**
  Central frequency of the last band. The default is 2e4.
- ***args**
  [() (Tuple):] See Analysis class.
- **bypassLundeby (false), (bool):**
  Bypass Lundeby calculation, or not. The default is False.
- **suppressWarnings (false), (bool):**
  Supress Lundeby warnings. The default is True.
- **ircut (None), (float):**
  Cut the IR and throw away the silence tail. The default is None.
- ****kwargs (), (Dict):**
  See Analysis.

- parameters (), (Tuple):

List of parameters names. return tuple(self._params.keys())

- **rms (), (np.ndarray):**
  Effective IR amplitude by frequency *band*.
- **SPL (), (np.ndarray):**
  Equivalent IR level by frequency *band*.
- **D50 (), (np.ndarray):**
  Room Definition by frequency *band*.
- **C80 (), (np.ndarray):**
  Room Clarity by frequency *band*.
- **Ts (), (np.ndarray):**
  Central Time by frequency *band*.
- **STearly (), (np.ndarray):**
  Early energy distribution by frequency *band*.
- **STlate (), (np.ndarray):**
  Late energy distribution by frequency *band*.
- **EDT (), (np.ndarray):**
  Early Decay Time by frequency *band*.
- **T20 (), (np.ndarray):**
  Reverberation time with 20 dB decay, by frequency *band*.
- **T30 (), (np.ndarray):**
  Reverberation time with 30 dB decay, by frequency *band*.

- **plot_param(name [str], **kwargs):**
  Plot a chart with the parameter passed in as *name*.
- **plot_rms(label [str], **kwargs):**
  Plot a chart for the impulse response's *rms* by frequency *bands*.
- **plot_SPL(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *SPL* by frequency *bands*.
- **plot_C80(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *C80* by frequency *bands*.
- **plot_D50(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *D50* by frequency *bands*.
- **plot_T20(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *T20* by frequency *bands*.
- **plot_T30(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *T30* by frequency *bands*.
- **plot_Ts(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *Ts* by frequency *bands*.
- **plot_EDT(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *EDT* by frequency *bands*.
- **plot_STearly(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *STearly* by frequency *bands*.

- **plot_STlate(label [str], yaxis [str], **kwargs):**
  Plot a chart for the impulse response's *STlate* by frequency *bands*.

For further information on methods see its specific documentation.

Authors:
João Vitor Gutkoski Paes, joao.paes@eac.ufsm.br Matheus Lazarin, matheus.lazarin@eac.ufsm.br Rinaldi Petrolli, rinaldi.petrolli@eac.ufsm.br

`static estimate_energy_parameters`(*ir: SignalObj*, *bands: ndarray*, *plotLundeby: bool = False*, *bypassLundeby: bool = False*, *suppressWarnings: bool = False*, ***kwargs*)

Estimate the Impulse Response energy parameters.

Parameters

- **bypassLundeby** (*bool*) – Whether to bypass calculation of Lundeby IR improvements or not. The default is False.
- **suppressWarnings** (*bool*) – If supress warnings about IR quality and the bypassing of Lundeby calculations. The default is False.

Returns
params – A dict with parameters by name.

Return type
Dict[str, np.ndarray]

`property parameters`
List of parameters names.

`property rms`
Effective IR amplitude by frequency *band*.

`property SPL`
Equivalent IR level by frequency *band*.

`property D50`
Room Definition by frequency *band*.

`property C80`
Effective IR amplitude, by frequency *band*.

`property Ts`
Central Time by frequency *band*.

`property STearly`
Early energy distribution by frequency *band*.

`property STlate`
Late energy distribution by frequency *band*.

`property EDT`
Early Decay Time by frequency *band*.

`property T20`
Reverberation time with 20 dB decay, by frequency *band*.

`property T30`
Reverberation time with 30 dB decay, by frequency *band*.

plot_param(*name: str*, ***kwargs*)

Plot a chart with the parameter passed in as *name*.

Parameters

- **name** (*str*) – Room parameter name, e.g. *'T20'* | *'C80'* | *'SPL'*, etc.
- **kwargs** (*Dict*) – All kwargs accepted by *Analysis.plot_bar*.

Returns
f – The figure of the plot chart.

Return type
matplotlib.Figure

plot_rms(*label='RMS'*, ***kwargs*)

Plot a chart for the impulse response's *rms* by frequency *bands*.

plot_SPL(*label='SPL'*, *yaxis='Level [dB]'*, ***kwargs*)

Plot a chart for the impulse response's *SPL* by frequency *bands*.

plot_C80(*label='C80'*, *yaxis='Clarity [dB]'*, ***kwargs*)

Plot a chart for the impulse response's *C80* by frequency *bands*.

plot_D50(*label='D50'*, *yaxis='Definition [%]'*, ***kwargs*)

Plot a chart for the impulse response's *D50* by frequency *bands*.

plot_T20(*label='T20'*, *yaxis='Reverberation time [s]'*, ***kwargs*)

Plot a chart for the impulse response's *T20* by frequency *bands*.

plot_T30(*label='T30'*, *yaxis='Reverberation time [s]'*, ***kwargs*)

Plot a chart for the impulse response's *T30* by frequency *bands*.

plot_Ts(*label='Ts'*, *yaxis='Central time [s]'*, ***kwargs*)

Plot a chart for the impulse response's *Ts* by frequency *bands*.

plot_EDT(*label='EDT'*, *yaxis='Early Decay Time [s]'*, ***kwargs*)

Plot a chart for the impulse response's *EDT* by frequency *bands*.

plot_STearly(*label='STearly'*, *yaxis='Early reflection level [dB]'*, ***kwargs*)

Plot a chart for the impulse response's *STearly* by frequency *bands*.

plot_STlate(*label='STlate'*, *yaxis='Late reflection level [dB]'*, ***kwargs*)

Plot a chart for the impulse response's *STlate* by frequency *bands*.

AudioIO Objects

Provide real time audio playback and recording, with special classes to concurrently read input audio.

class pytta.**Streaming**(*IO: str*, *samplingRate: int*, *device: int*, *datatype: str = 'float32'*, *blocksize: int = 0*, *inChannels: Optional[ChannelsList] = None*, *outChannels: Optional[ChannelsList] = None*, *excitation: Optional[SignalObj] = None*, *duration: Optional[float] = None*, *numSamples: Optional[int] = None*, *monitor: Optional[Monitor] = None*, **args*, ***kwargs*)

Stream control.

__init__(*IO: str*, *samplingRate: int*, *device: int*, *datatype: str = 'float32'*, *blocksize: int = 0*, *inChannels: Optional[ChannelsList] = None*, *outChannels: Optional[ChannelsList] = None*, *excitation: Optional[SignalObj] = None*, *duration: Optional[float] = None*, *numSamples: Optional[int] = None*, *monitor: Optional[Monitor] = None*, **args*, ***kwargs*)

Manage input and output of audio.

Parameters

- **IO** (*str*) – DESCRIPTION.
- **msmnt** (*Measurement*) – DESCRIPTION.
- **datatype** (*str, optional*) – DESCRIPTION. Defaults to 'float32'.
- **blocksize** (*int, optional*) – DESCRIPTION. Defaults to 0.
- **duration** (*Optional[float], optional*) – DESCRIPTION. Defaults to 5.
- **monitor** (*Optional[Monitor], optional*) – DESCRIPTION. Defaults to None.
- ***args** (*TYPE*) – DESCRIPTION.
- ****kwargs** (*TYPE*) – DESCRIPTION.

Returns

None.

__enter__()

Provide context functionality, the *with* keyword, e.g.

```
>>> with Streaming(*args, **kwargs) as strm:  # <-- called here
...     strm.playrec()
...
>>>
```

__exit__(*exc_type: Type*, *exc_val: Exception*, *exc_tb: Type*)

Provide context functionality, the *with* keyword, e.g.

```
>>> with Streaming(*args, **kwargs) as strm:
...     strm.playrec()
...                             # <-- called here
>>>
```

set_io_properties(*io: str*, *channels: ChannelsList*)

Allocate memory for input and output of data, set counter.

Parameters

msmnt (*TYPE*) – DESCRIPTION.

Returns

None.

set_monitoring(*monitor: Optional[Monitor] = None*)

Set up the class used as monitor. It must have the following methods with these names.

def setup(None) -> None:

Call any function and other object configuration needed for the **monitoring** return

def callback(indata: np.ndarray, outdata: np.ndarray,

frames: int, status: sd.CallbackFlags) -> None:

Process the data gathered from the **stream** return

It will be called from within a parallel process that the Recorder starts and terminates during it's .run() call.

Parameters
monitor (*object*) – Object or class that will be used to monitor the stream data flow.

runner(*StreamType: Type*, *stream_callback: Callable*, *numchannels: Union[List[int], int]*)

Do the work.

Instantiates a sounddevice.*Stream and calls for a threading.Thread if any Monitor is set up. Then turn on the monitorCheck Event, and starts the stream. Waits for it to finish, unset the event And terminates the process

Returns

Return type

input_callback(*indata: ndarray*, *frames: int*, *times: type*, *status: CallbackFlags*)

This method will be called from the stream, as stated on sounddevice's documentation.

output_callback(*outdata: ndarray*, *frames: int*, *times: type*, *status: CallbackFlags*)

This method will be called from the stream, as stated on sounddevice's documentation.

stream_callback(*indata: ndarray*, *outdata: ndarray*, *frames: int*, *time: type*, *status: CallbackFlags*)

This method will be called from the stream, as stated on sounddevice's documentation.

class pytta.**Monitor**(*numsamples: int*, *samplingrate: int = 44100*, *numchannels: List[int] = [1, 1]*, *datatype: str = 'float32'*)

PyTTa default Monitor base class.

__init__(*numsamples: int*, *samplingrate: int = 44100*, *numchannels: List[int] = [1, 1]*, *datatype: str = 'float32'*)

Default Monitor class.

Subclasses must override *setup*, *callback* and *tear_down* methods.

Parameters

- **numsamples** (*int*) – DESCRIPTION.
- **samplingrate** (*int, optional*) – DESCRIPTION. The default is default.samplingRate.
- **numchannels** (*List[int], optional*) – DESCRIPTION. The default is [len(default.inChannel), len(default.outChannel)].
- **datatype** (*str, optional*) – DESCRIPTION. The default is 'float32'.

Return type
None.

setup()

Start up widgets, threads, anything that will be used during audio processing.

reset()

Reset write counter.

callback(*frames: int*, *indata: ndarray*, *outdata: Optional[ndarray] = None*)

The audio processing itself, will be called for every chunk of data taken from the queue.

tear_down()

Finish any started object here, like GUI members, to allow the Monitor parallel process be joined.

Filter Objects

class pytta.OctFilter(*order: Optional[int] = None*, *nthOct: Optional[int] = None*, *samplingRate: Optional[int] = None*, *minFreq: Optional[float] = None*, *maxFreq: Optional[float] = None*, *refFreq: Optional[float] = None*, *base: Optional[int] = None*)

Octave filter.

__init__(*order: Optional[int] = None*, *nthOct: Optional[int] = None*, *samplingRate: Optional[int] = None*, *minFreq: Optional[float] = None*, *maxFreq: Optional[float] = None*, *refFreq: Optional[float] = None*, *base: Optional[int] = None*) → None

Parameters

- **order** (*int, optional*) – DESCRIPTION. The default is None.
- **nthOct** (*int, optional*) – DESCRIPTION. The default is None.
- **samplingRate** (*int, optional*) – DESCRIPTION. The default is None.
- **minFreq** (*float, optional*) – DESCRIPTION. The default is None.
- **maxFreq** (*float, optional*) – DESCRIPTION. The default is None.
- **refFreq** (*float, optional*) – DESCRIPTION. The default is None.
- **base** (*int, optional*) – DESCRIPTION. The default is None.

Returns
DESCRIPTION.

Return type
None

filter(*sigobj*)

Filter the signal object.

For each channel inside the input signalObj, will be generated a new SignalObj with the channel filtered signal.

Parameters
sigobj – SignalObj

Returns

List
A list containing one SignalObj with the filtered data for each channel in the original signalObj.

Return type
output

3.2.2 Utilities

Utilities sub-package.

Contains simple tools which help to keep things modularized and make accessible the reuse of some operations. Intended to hold tools (classes and functions) whose operate built-in python classes, NumPy arrays, and other stuff not contained by the pytta.classes sub-package.

Available modules:

- colore
- freq
- maths

Created on Tue May 5 00:31:42 2020

@author: João Vitor G. Paes

Mathematical utilities

Created on Tue May 5 00:34:36 2020

@author: joaovitor

pytta.utils.**maxabs**(*arr: array*) → int

Maximum of the absolute of array values.

Parameters
(np.array) (*arr*) –

Returns
int or float

Return type
Maximum of the absolute values.

pytta.utils.**arr2rms**(*arr: array*) → float

Root of the mean of a squared array.

Parameters
(np.array) (*arr*) –

Returns
float

Return type
RMS of data.

pytta.utils.**rms2dB**(*rms: float*, *power: bool = False*, *ref: float = 1.0*) → float

RMS to decibel.

Parameters

- **(float)** (*rms*) –
- **(bool** (*power*) –
- **optional)** (*Reference value for decibel scale. Defaults to 1.0.*) –
- **(float** (*ref*) –

- **optional)** –

Returns
float

Return type
Decibel scaled value.

pytta.utils.**arr2dB**(*arr: array*, *power: bool = False*, *ref: float = 1.0*) → float

Calculate the decibel level of an array of data.

Parameters

- **(np.array)** (*arr*) –
- **(bool** (*power*) –
- **optional)** (*Decibel reference. Defaults to 1.*) –
- **(float** (*ref*) –
- **optional)** –

Returns
float

Return type
Decibel scaled value.

Colore Textos

Módulo simples para colorir textos

@author: João Vitor G. Paes

class pytta.utils.**ColorStr**(*font: str = 'clear'*, *back: str = 'clear'*)

Color string

__init__(*font: str = 'clear'*, *back: str = 'clear'*) → None

Pintor de linhas.

Example use:

```
>>> black_on_white_str = ColorStr("black", "white")
>>> red_on_green_str = ColorStr("red", "green")
>>> print(black_on_white_str("Estou usando colore.ColorStr"))
>>> print(red_on_green_str("I'm using colore.ColorStr"))
```

Parameters

- **font** (*str, optional*) – Cor da fonte | Font color. Defaults to “clear”.
- **back** (*str, optional*) – Cor do fundo | Background color. Defaults to “clear”.

Returns
None.

__call__(*text: Optional[str] = None*) → str

Paint the text with its font and background colors.

Parameters
text (*str, optional*) – The text to be painted. Defaults to None.

Returns
Colored text and background.

Return type
str

property fntclr: str
Font color.

property font: str
Alias for fntclr.

property bgrclr: str
Background color.

property background: str
Alias for bgrclr.

property back: str
Alias for bgrclr.

pytta.utils.**colorir**(*texto: str*, *fntclr: Optional[str] = None*, *bgrclr: Optional[str] = None*) → str
Retorna o texto colorido nas cores escolhidas.

Caso nenhuma cor seja informada, retorna o texto sem alterações.

Parameters
- **texto** (*str*) – DESCRIPTION.
- **fntclr** (*str, optional*) – Font color. Defaults to None.
- **bgrclr** (*str, optional*) – Background color. Defaults to None.

Returns
Colored font and background.

Return type
texto (str)

pytta.utils.**pinta_fundo**(*text: str*, *color: str*) → str
Pinta o fundo do texto com a cor escolhida.

Parameters
- **text** (*str*) – O texto em si.
- **color** (*str*) – A cor escolhida.

Returns
Texto com fundo colorido.

Return type
str

pytta.utils.**pinta_texto**(*text: str*, *color: str*) → str
Pinta o texto com a cor escolhida.

Parameters
- **text** (*str*) – O texto em si.
- **color** (*str*) – A cor escolhida.

Returns
Texto colorido.

Return type
str

Frequency utilities

This utility provides frequency and fractional octave frequency bands functionalities.

For syntax purposes you should start with:

```
>>> from pytta import utils as utils
```

Available functions:

```
>>> utils.freq_to_band(freq, nthOct, ref, base)
>>> utils.fractional_octave_frequencies(nthOct = 3, freqRange = (20., 20000.), refFreq =␣
↪1000., base = 10)
>>> utils.normalize_frequencies(freqs, samplingRate = 44100)
>>> utils.freqs_to_center_and_edges(freqs)
>>> utils.filter_values(freq, values, nthOct = 3)
```

For further information, check the docstrings for each function mentioned above.

Authors:
João Vitor G. Paes joao.paes@eac.ufsm.br and Caroline Gaudeoso caroline.gaudeoso@eac.ufsm.br Rinaldi Petrolli rinaldi.petrolli@eac.ufsm.br

pytta.utils.**freq_to_band**(*freq: float*, *nthOct: int*, *ref: float*, *base: int*) → int

Band number from frequency value.

Parameters

- **freq** (*float*) – The frequency value.
- **nthOct** (*int*) – How many bands per octave.
- **ref** (*float*) – Frequency of reference, or band number 0.
- **base** (*int*) – Either 10 or 2.

Raises
ValueError – If base is not 10 nor 2 raises value error.

Returns
The band number from center.

Return type
int

pytta.utils.**fractional_octave_frequencies**(*nthOct: int = 3*, *freqRange: Tuple[float] = (20.0, 20000.0)*, *refFreq: float = 1000.0*, *base: int = 10*) → ndarray

Lower, center and upper frequency values of all bands within range.

Parameters

- **nthOct** (*int, optional*) – bands of octave/nthOct. The default is 3.
- **freqRange** (*Tuple[float], optional*) – frequency range. These frequencies are inside the lower and higher band, respectively. The default is (20., 20000.).

- **refFreq** (*float, optional*) – Center frequency of center band. The default is 1000..
- **base** (*int, optional*) – Either 10 or 2. The default is 10.

Returns
freqs – Array with shape (N, 3).

Return type
numpy.ndarray

pytta.utils.**normalize_frequencies**(*freqs: ndarray*, *samplingRate: int = 44100*) → ndarray

Normalize frequencies for any sampling rate.

Parameters

- **freqs** (*np.ndarray*) – DESCRIPTION.
- **samplingRate** (*int, optional*) – DESCRIPTION. The default is 44100.

Returns
DESCRIPTION.

Return type
TYPE

pytta.utils.**freqs_to_center_and_edges**(*freqs: ndarray*) → Tuple[ndarray]

Separate the array returned from *fractional_octave_frequencies*.

The returned arrays corresponde to the center and edge frequencies of the fractional octave bands

Parameters
freqs (*np.ndarray*) – Array returned from *fractional_octave_frequencies*.

Returns

- **center** (*np.ndarray*) – Center frequencies of the bands.
- **edges** (*np.ndarray*) – Edge frequencies (lower and upper) of the bands.

3.2.3 Apps

Applications sub-package.

This sub-package contains applications built from the toolbox's functionalities. The apps are called from the top-level (e.g. pytta.roomir).

Available apps:

- **pytta.roomir:**
  Room impulsive response acquisition and post-processing;

Created on Sun Jun 23 15:02:03 2019

@author: Matheus Lazarin - matheus.lazarin@eac.ufsm.br

Room Impulsive Response

RooomIR application.

This application was developed using the toolbox's existent functionalities, and is aimed to give support to acquire and post-process impulsive responses, and calculate room acoustics parameters.

For syntax purposes you should start with:

```
>>> from pytta import roomir as roomir
```

The functionalities comprise tools for data acquisition (MeasurementSetup, MeasurementData, and TakeMeasure classes) and post-processing (MeasurementPostProcess class), including some really basic statistical treatment.

The workflow consists of creating a roomir.MeasurementSetup and a roomir.MeasurementData object, then taking a measurement with a roomir.TakeMeasure object (see its documentation for possible take kinds); there will be as many TakeMeasure instantiations as measurement takes. After a measurement is taken it's saved through the roomir.MeasurementData.save_take() method.

All measured responses and the post-processed impulsive responses are stored as MeasuredThing class objects, while the calculated results as Analysis class objects. All data is stored in the HDF5 file scheme designed for the toolbox. For more information about the specific file scheme for the MeasurementData.hdf5 file check the MeasurementData class documentation.

It is also possible to use the LabJack U3 hardware with the EI 1050 probe to acquire humidity and temperature values during the measurement. Check the TakeMeasure class documentation for more information.

The usage of this app is shown in the files present in the examples folder.

Available classes:

```
>>> roomir.MeasurementSetup
>>> roomir.MeasurementData
>>> roomir.TakeMeasure
>>> roomir.MeasuredThing
>>> roomir.MeasurementPostProcess
```

Available functions:

>> MS, D = roomir.med_load('med-name')

For further information, check the docstrings for each class and function mentioned above. This order is also recommended.

Authors:
Matheus Lazarin, matheus.lazarin@eac.ufsm.br

class pytta.roomir.**MeasurementSetup**(*name*, *samplingRate*, *device*, *excitationSignals*, *freqMin*, *freqMax*, *inChannels*, *inCompensations*, *outChannels*, *outCompensations*, *averages*, *pause4Avg*, *noiseFloorTp*, *calibrationTp*)

Holds the measurement setup information. Managed in disc by the roomir.MeasurementData class and loaded back to memory through the roomir.med_load function.

- **name (), (str):**
  Measurement identification name. Used on roomir.med_load('name');
- **samplingRate (), (int):**
  Device sampling rate;

- **device (), (list | int):**
  Audio I/O device identification number from pytta.list_devices(). Can be an integer for input and output with the same device, or a list for different devices. E.g.:

```
>>> device = [1,2]  % [input,output]
```

- **excitationSignals (), (dict):**
  Dictionary containing SignalObjs with excitation signals. E.g.:

```
>>> excitSigs = {'sweep18': pytta.generate.sweep(fftDegree=18),
                 'speech': pytta.read_wave('sabine.wav')}
```

- **freqMin (), (float):**
  Analysis band's lower limit;

- **freqMax (), (float):**
  Analysis band's upper limit

- **inChannels (), (dict):**
  Dict containing input channel codes, hardware channel and name. Aditionally is possible to group channels with an extra 'groups' key. E.g.:

```
>>> inChannels={'LE': (4, 'Left ear'),
                'RE': (3, 'Right ear'),
                'AR1': (5, 'Array mic 1'),
                'AR2': (6, 'Array mic 2'),
                'AR3': (7, 'Array mic 3'),
                'Mic1': (1, 'Mic 1'),
                'Mic2': (2, 'Mic 2'),
                'groups': {'HATS': (4, 3),
                           'Array': (5, 6, 7)} }
```

- **inCompensations (), (dict):**
  Magnitude compensation for each input transducers, but not mandatory for all. E.g.:

```
>>> inCompensations={'AR1': (AR1SensFreq, AR1SensdBMag),
                     'AR2': (AR2SensFreq, AR2SensdBMag),
                     'AR3': (AR3SensFreq, AR3SensdBMag),
                     'Mic2': (M2SensFreq, M2SensdBMag) }
```

- **outChannels (default), (type):**
  Dict containing output channel codes, hardware channel and name. E.g.:

```
>>> outChannels={'O1': (1, 'Dodecahedron 1'),
                 'O2': (2, 'Dodecahedron 2'),
                 'O3': (4, 'Room sound system') }
```

- **outCompensations (default), (type):**
  Magnitude compensation for each output transducers, but not mandatory for all. E.g.:

```
>>> outCompensations={'O1': (Dodec1SensFreq, Dodec1SensdBMag),
                      'O2': (Dodec2SensFreq, Dodec2SensdBMag) }
```

- **averages (), (int):**
  Number of averages per take. This option is directly connected to the confidence interval calculated by the MeasurementPostProcess class methods. Important in case you need some statistical treatment;

- **pause4Avg (), (bool):**
  Option for pause between averages;
- **noiseFloorTp (), (float):**
  Recording time length in seconds for noisefloor measurement take type;
- **calibrationTp (default), (type):**
  Recording time length in seconds for microphone indirect calibration take type;

class pytta.roomir.**MeasurementData**(*MS*, *skipFileInit=False*)

Class intended to store and retrieve from disc the acquired data as MeasuredThing objects plus the MeasurementSetup. Used to calculate the impulsive responses and calibrated responses as well.

Instantiation:

```
>>> MS = pytta.roomir.MeasurementSetup(...)
>>> D = pytta.roomir.MeasurementData(MS)
```

- **MS (), (roomir.MeasurementSetup):**
  MeasurementSetup object;
- **save_take(...):**
  Save an acquired roomir.TakeMeasure in disc;
- **get(...):**
  Retrieves from disc MeasuredThings that matches the provided tags and returns a dict;
- **calculate_ir(...):**
  Calculate the Impulsive Responses from the provided dict, which is obtained trough the get(...) method. Saves the result as new MeasuredThing objects;
- **calibrate_res(...):**
  Apply indirect calibration to the measured signals from the provided dict, which is obtained trough the get(...) method. Saves the result as new MeasuredThing objects;

For further information on methods see its specific documentation.

save_take(*TakeMeasureObj*)

Saves in disc the resultant roomir.MeasuredThings from a roomir.TakeMeasure's run.

- TakeMeasureObj (), (roomir.TakeMeasure)

Usage:

```
>>> myTake = roomir.TakeMeasure(kind="roomres", ...)
>>> myTake.run()
>>> D.save_take(myTake)
```

get(**args*, *skipMsgs=False*)

Gets from disc the MeasuredThings that matches with the provided tags as non-keyword arguments.

- non-keyword arguments (), (string):

  Those are tags. Tags are the main information about the roomir.MeasuredThings found on its filename stored in disc. You can mix as many tags as necessary to define well your search. The possible tags are:

  – **kind (e.g. roomres, roomir... See other available kinds**

  in roomir.MeasuredThing's docstrings);

```
>>> D.get('roomres')  % Gets all measured roomres;
```

– **source-receiver cfg. (single info for 'noisefloor'**

MeasuredThing kind, e.g. 'R3'; or a pair for other kinds, e.g. 'S1-R1');

```
>>> D.get('S1')  % Gets everything measured in source
                 % position 1
```

– **output-input cfg. (single channel for 'miccalibration',**

e.g. 'Mic1'; output-input pair for other kinds, e.g. 'O1-Mic1'):

```
>>> D.get('O1')  % Gets everything measured through
                 % ouput 1
```

```
>>> D.get('O1','Mic1')  % or
>>> D.get('O1-Mic1')
```

– excitation signal (e.g. 'SWP19'):

```
>>> D.get('SWP20')  % Gets everything measured with
                    % excitation signal 'SWP20'
```

– take counter (e.g. '_1', '_2', ...):

```
>>> D.get('_1')  % Gets all first takes
```

Only MeasuredThings that matches all provided tags will be returned.

- **skipMsgs (false), (bool):**
  Don't show the search's resultant messages.

- **getDict (dict):**
  Dict with the MeasuredThing's filenames as keys and the MeasuredThing itself as values. e.g.:

```
>>> getDict = {'roomres_S1-R1_O1-Mic1_SWP19_1':
                   roomir.MeasuredThing}
```

Specific usage example:

Get the 'roomir' MeasuredThing's first take at S1-R1 with Mic1, Output1, and sweep:

```
>>> getDict = MeasurementData.get('roomrir', 'Mic1-O1', '_1', ...
                                  'SWP19', 'S1-R1')
```

As you see, the order doesn't matter because the algorithm just look for matching tags in the filenames.

calculate_ir(*getDict*, *calibrationTake=1*, *skipInCompensation=False*, *skipOutCompensation=False*, *skipBypCalibration=False*, *skipRegularization=False*, *skipIndCalibration=False*, *IREndManualCut=None*, *IRStartManualCut=None*, *skipSave=False*, *whereToOutComp='excitation'*)

Gets the dict returned from the roomir.MeasuremenData.get() method, calculate the impulsive responses, store to disc, and return the correspondent getDict. Check the input arguments below for options.

This method generates new MeasuredThings with a kind derived from the measured kind. The possible conversions are:

- 'roomres' MeasuredThing kind to a 'roomir' MeasuredThing kind;
- 'channelcalibration' to 'channelcalibir';
- **'sourcerecalibration' to 'sourcerecalibir' (for the Strengh**
  Factor recalibration method. See pytta.rooms.G for more information);
- **getDict (), (dict):**
  Dict from the roomir.MeasurementData.get(...) method;
- **calibrationTake (1), (int):**
  Choose the take from the 'miccalibration' MeasuredThing for the indirect calibration of the correspondent input channels;
- **skipInCompensation (False), (bool):**
  Option for skipping compensation on the input chain with the provided response to the MeasurementSetup;
- **skipOutCompensation (False), (bool):**
  Option for skipping compensation on the output chain with the provided response to the MeasurementSetup;
- **skipBypCalibration (False), (bool):**
  Option for skipping bypass calibration. Bypass calibration means deconvolving with the impulsive response measured between the output and input of the soundcard.
- **skipRegularization (False), (bool):**
  Option for skipping Kirkeby's regularization. For more information see pytta.ImpulsiveResponse's docstrings.
- **skipIndCalibration (False), (bool):**
  Option for skipping the indirect calibration;
- **IREndManualCut (None), (float):**
  Manual cut of the calculated impulsive response;
- **IRStartManualCut (None), (float):**
  Manual cut of the calculated impulsive response;
- **skipSave (False), (bool):**
  Option to skip saving the new MeasuredThings to disc. Usefull when you need to calculated the same impulsive response with different options and don't want to override the one saved previously.
- **getDict (dict):**
  Dict with the calculated MeasuredThings, with filenames as keys and the MeasuredThing itself as values. e.g.:

```
>>> getDict = {'roomir_S1-R1_O1-Mic1_SWP19_1':
                    roomir.MeasuredThing}
```

calibrate_res(*getDict*, *calibrationTake=1*, *skipInCompensation=False*, *skipSave=False*)

Gets the dict returned from the roomir.MeasuremenData.get() method, apply the indirect calibration, store to disc, and return the correspondent getDict. Check the input arguments below for options.

This method generates new MeasuredThings with a kind derived from the measured kind. The possible conversions are:

- 'roomres' MeasuredThing kind to a 'calibrated-roomres';
- 'noisefloor' to 'calibrated-noisefloor';
- 'sourcerecalibration' to 'calibrated-sourcerecalibration';

- **getDict (), (dict):**
  Dict from the roomir.MeasurementData.get(...) method;
- **calibrationTake (1), (int):**
  Choose the take from the 'miccalibration' MeasuredThing for the indirect calibration of the correspondent input channels;
- **skipInCompensation (False), (bool):**
  Option for skipping compensation on the input chain with the provided response to the MeasurementSetup;
- **skipSave (False), (bool):**
  Option to skip saving the new MeasuredThings to disc.

- **getDict (dict):**
  Dict with the calculated MeasuredThings, with filenames as keys and the MeasuredThing itself as values. e.g.:

```
>>> getDict = {'calibrated-roomres_S1-R1_O1-Mic1_SWP19_1':
                    roomir.MeasuredThing}
```

class pytta.roomir.**TakeMeasure**(*MS*, *kind*, *inChSel*, *receiversPos=None*, *excitation=None*, *outChSel=None*, *outputAmplification=0*, *sourcePos=None*, *tempHumid=None*)

Class intended to hold a measurement take configuration, take the measurement, and obtain the MeasuredThing for each channel/group in inChSel property.

- **MS (), (roomir.MeasurementSetup):**
  The setup of the current measurement;
- **tempHumid (), (pytta.classes.lju3ei1050):**
  Object for communication with the LabJack U3 with the probe EI1050 for acquiring temperature and humidity. For more information check pytta.classes.lju31050 docstrings;
- **kind (), (string):**
  Possible measurement kinds:
  - 'roomres': room response to the excitation signal;
  - **'noisefloor': acquire noise floor for measurement quality** analysis;
  - **'miccalibration': acquire calibrator signal (94dB SPL @ 1kHz)** for indirect calibration;
  - **'channelcalibration': acquire response of the ouput connected** to the input channel;
  - **'sourcerecalibration': acquire recalibration response for** Strength Factor measurement. For more information check pytta.rooms.strength_factor doctrings.
- **inChSel (), (list):**
  Active input channels (or groups) for the current take. E.g.:

```
>>> inChSel = ['Mic2', 'AR2']
```

- **receiversPos (), (list):**
  List with the positions of each input channel/group. E.g.:

```
>>> receiversPos = ['R2', 'R4']
```

- **excitation (), (string):**
  Code of the excitation signal provided to the MeasurementSetup. E.g.:

```
>>> excitation = 'SWP19'
```

- **outChSel (), (string):**
  Code of the output channel provided to the MeasurementSetup. E.g.:

```
>>> outChSel = 'O1'
```

- **outputAmplification (0) (float):**
  Output amplification in dB;
- **sourcePos (), (string):**
  Source's position. E.g.:

```
>>> sourcePos = 'R1'
```

- **run():**
  Acquire data;
- **measuredThings (list):**
  Contains the MeasuredThing objects resultant from the measurement take.

run()

Take measurement (initializate acquisition).

Has no input arguments.

Usage:

```
>>> myTake.run()
>>> D.save_take(myTake)
```

class pytta.roomir.**MeasuredThing**(*kind=''*, *arrayName=''*, *measuredSignals=[]*, *timeStamps=[]*, *tempHumids=[]*, *inChannels=None*, *sourcePos=None*, *receiverPos=None*, *excitation=None*, *outChannel=None*, *outputAmplification=0*)

Obtained through a roomir.TakeMeasure object. Contains information of a measurement take for one source-receiver configuration. Shouldn't be instantiated by user.

- **kind (str):**
  Possible kinds for a MeasuredThing:
  - 'roomres';
  - 'roomir' ('roomres' after IR calculation);
  - 'noisefloor';

  - 'miccalibration'
  - 'sourcerecalibration';
  - 'recalibir' ('sourcerecalibration' after IR calculation);
  - 'channelcalibration';
  - 'channelcalibir' ('channelcalibration' after IR calculation);

- **arrayName (str):**
  Code of the input channel or group (array);
- **measuredSignals (list):**
  Contains the resultant SignalObjs or ImpulsiveResponses;
- **timeStamps (list):**
  Contains the timestamps for each measurement take;
- **tempHumids (list):**
  Contains the temperature and humidity readings for each measurement take;
- **inChannels (roomir._MeasurementChList):**
  Measurement channel list object. Identifies the used soundcard's input channels;
- **sourcePos (str):**
  Source position;
- **receiverPos (str):**
  Receiver's (microphone or array) position;
- **excitation (str):**
  Excitation signal code in MeasurementSetup;
- **outChannel (roomir._MeasurementChList):**
  Measurement channel list object. Identifies the used soundcard's output channel;
- **outputAmplification (float):**
  Output amplification in dB set for the take;
- **outputLinearGain (float):**
  Output amplification in linear scale;
- **numChannels (float):**
  The number of channels;
- **averages (int):**
  The number of averages;

`class` pytta.roomir.**MeasurementPostProcess**(*nthOct=3*, *minFreq=100*, *maxFreq=10000*)

Holds a measurement post processing session.

- **nthOct (3), (int):**
  Number of bands per octave;
- **minFreq (100), (float):**
  Minimum analysis' frequency;
- **maxFreq (1000), (float):**
  Minimum analysis' frequency;
- **RT (getDict, decay, IREndManualCut)**
  Calculates the average reverberation time for each source-receiver pair from the provided getDict. Also calculates a 95% confidence interval from a T-Student distribution, which is depen-

dent on the number of averages. For more information on reverberation time calculation go to pytta.rooms.reverberation_time

- **G (Lpe_avgs, Lpe_revCh, V_revCh, T_revCh, Lps_revCh, Lps_inSitu):**
  Calculates the strength factor from the G terms provided by several getDict-like dicts from other G methods of this class.
- **G_Lpe_inSitu (roomirsGetDict, IREndManualCut):**
  Calculates the sound exposure level of the room impulsive response. For more information about this G term go to pytta.rooms.G_Lpe;
- **G_Lpe_revCh(roomirsGetDict, IREndManualCut):**
  Calculates the sound exposure level of the reverberation chamber's impulsive response. For more information about this G term go to pytta.rooms.G_Lpe;
- **G_Lps (sourcerecalibirGetDict):**
  Calculates the sound exposure level of the recalibration impulsive response. For more information about this G term go to pytta.rooms.G_Lps;
- **G_T_revCh (roomirsGetDict, IREndManualCut, T):**
  Calculates the mean reverberation time of the reverberation chamber;

For further information check the specific method's docstrings.

`RT`(*roomirsGetDict*, *decay=20*, *IREndManualCut=None*)

Calculates the average reverberation time for each source-receiver pair from the dict provided by the roomir.MeasurementData.get method.

Also calculates a 95% confidence interval from a T-Student distribution, which is dependent on the number of averages. For more information on reverberation time calculation go to pytta.rooms.reverberation_time.

- **getDict ('Time [s]'), (str):**
  a dict from the roomir.MeasurementData.get method containing MeasuredThings of the type 'roomir' (room impulsive response);
- **decay (20), (int):**
  dynamic range of the line fit;
- **IREndManualCut (None), (float):**
  remove the end of the impulsive response from IREndManualCut, given in seconds;
- **TR (dict):**
  a dict containing the mean reverberation time (Analysis) for each source-receiver configuration, which is a key;

`G_Lps`(*recalibirsGetDict*)

Calculates the mean sound exposure level from the recalibration impulsive responses. For more information about this G term go to pytta.rooms.G_Lps;

- **G_Lps (), (dict from rmr.D.get(...)):**
  a dict from the roomir.MeasurementData.get method containing MeasuredThings of the type 'recalibir';
- **finalLps (Analysis):**
  an Analysis object with the averaged recalibration exposure level;

`G_Lpe_inSitu`(*roomirsGetDict*, *IREndManualCut=None*)

Calculates the room impulsive response' sound exposure level for each source-receiver cfg. For more information about this G term go to pytta.rooms.G_Lpe;

Receives

- **roomirsGetDict (), ():**
  a dict from the roomir.MeasurementData.get method containing MeasuredThings of the type 'roomir' (room impulsive response);
- **IREndManualCut (None), (float):**
  remove the end of the impulsive response from IREndManualCut, given in seconds;
- **Lpe_avgs (dict):**
  a dict containing a list with the sound exposure level averages (Analyses) for each source-receiver configuration, which is a key;

`G_Lpe_revCh`(*roomirsGetDict*, *IREndManualCut=None*)

Calculates the mean sound exposure level of the reverberation chamber's impulsive response. For more information about this G term go to pytta.rooms.G_Lpe;

- **roomirsGetDict (), ():**
  a dict from the roomir.MeasurementData.get method containing MeasuredThings of the type 'roomir' (room impulsive response);
- **IREndManualCut (None), (float):**
  remove the end of the impulsive response from IREndManualCut, given in seconds;
- **Lpe (Analysis):**
  an Analysis with the mean sound exposure level calculated from all the reverberation chamber's impulsive responses;

`G`(*Lpe_avgs*, *Lpe_revCh*, *V_revCh*, *T_revCh*, *Lps_revCh*, *Lps_inSitu*)

Calculates the mean strength factor for each source-receiver configuration with the G terms provided by other methods of this class. Also provides some basic statistical treatment.

For further information on the recalibration method (to correct changes on the source's sound power) check:

Christensen, C. L.; Rindel, J. H. APPLYING IN-SITU RECALIBRATION FOR SOUND STRENGTH MEASUREMENTS IN AUDITORIA.

- **Lpe_avgs (), (dict from rmr.get(...)):**
  a dict provided by the rmr.get(...) method. Calculates a mean G for all source-receiver configurations provided with the dict. Also calculates the 95% confidence interval for a T-Student distribution;
- **Lpe_revCh (), (Analysis):**
  a pytta.Analysis object with the mean exposure level inside the reverberation chamber during the source calibration (sound power measurement);
- **V_revCh (), (float):**
  the volume of the reverberation chamber;
- **T_revCh (), (Analysis):**
  a pytta.Analysis object for the reverberation chamber's reverberation time;
- **Lps_revCh (), (Analysis)**
  the exposure level of the recalibration procedure in the reverberation chamber;

- **Lps_inSitu (), (Analysis):**
  the exposure level of the recalibration procedure in situ;

Return (type):

- **G (dict):**
  a dict containing the mean G (Analysis) for each source-receiver configuration, which is a key;

G_T_revCh(*roomirsGetDict*, *IREndManualCut=None*, *T=20*)

Calculates the mean reverberation time of the reverberation chamber;

- **roomirsGetDict (), ():**
  a dict from the roomir.MeasurementData.get method containing MeasuredThings of the type 'roomir' (room impulsive response);
- **IREndManualCut (None), (float):**
  remove the end of the impulsive response from IREndManualCut, given in seconds;
- **T_revCh (Analysis):**
  an Analysis with the mean reverberation time calculated from all the reverberation chamber's impulsive responses;

pytta.roomir.**med_load**(*medname*)

Loads a measurement to continue measuring or either post processing.

Usage:

```
>>> MS, D = roomir.med_load('measurement name')
```

- **medname (), (str):**
  the measurement name given in the MeasurementSetup object instantiation;
- (roomir.MeasurementSetup, roomir.MeasurementData) (tuple)

3.2.4 Functions

Set of useful functions of general purpouses when using PyTTa.

Includes reading and writing wave files, seeing the audio IO devices available and few signal processing tools.

Available functions:

```
>>> pytta.list_devices()
>>> pytta.read_wav(fileName)
>>> pytta.write_wav(fileName, signalObject)
>>> pytta.merge(signalObj1, signalObj2, ..., signalObjN)
>>> pytta.slipt(signalObj)
>>> pytta.fft_convolve(signalObj1, signalObj2)
>>> pytta.find_delay(signalObj1, signalObj2)
>>> pytta.corr_coef(signalObj1, signalObj2)
>>> pytta.resample(signalObj, newSamplingRate)
>>> pytta.peak_time(signalObj1, signalObj2, ..., signalObjN)
>>> pytta.plot_time(signalObj1, signalObj2, ..., signalObjN)
>>> pytta.plot_time_dB(signalObj1, signalObj2, ..., signalObjN)
>>> pytta.plot_freq(signalObj1, signalObj2, ..., signalObjN)
```

(continues on next page)

(continued from previous page)

```
>>> pytta.plot_bars(signalObj1, signalObj2, ..., signalObjN)
>>> pytta.save(fileName, obj1, ..., objN)
>>> pytta.load(fileName)
```

For further information, check the function specific documentation.

pytta.**merge**(*signal1*, **signalObjects*)

Gather all channels of the signalObjs given as input arguments into a single SignalObj.

pytta.**fft_convolve**(*signal1*, *signal2*)

Use scipy.signal.fftconvolve() to convolve two time domain signals.

```
>>> convolution = pytta.fft_convolve(signal1,signal2)
```

pytta.**read_wav**(*fileName*)

Read a wave file into a SignalObj.

pytta.**write_wav**(*fileName*, *signalIn*)

Write a SignalObj into a single wave file.

pytta.**list_devices**()

Shortcut to sounddevice.query_devices().

Made to exclude the need of importing Sounddevice directly just to find out which audio devices can be used.

```
>>> pytta.list_devices()
```

Return type
A tuple containing all available audio devices.

pytta.**get_device_from_user**() → Union[List[int], int]

Print the device list and query for a number input of the device, or devices.

Returns
Practical interface for querying devices to be used within scripts.

Return type
Union[List[int], int]

pytta.**SPL**(*signal*, *nthOct=3*, *minFreq=100*, *maxFreq=4000*)

Calculate the *signal*'s Sound Pressure Level

The calculations are made by frequency bands and ranges from *minFreq* to *maxFreq* with *nthOct* bands per octave.

Returns
Analysis

Return type
The sound pressure level packed into an Analysis object.

pytta.**find_delay**(*signal1*, *signal2*)

Cross Correlation alternative.

More efficient fft based method to calculate time shift between two signals.

```
>>> shift = pytta.find_delay(signal1,signal2)
```

pytta.**resample**(*signal*, *newSamplingRate*)

Resample the timeSignal of the input SignalObj to the given sample rate using the scipy.signal.resample() function

pytta.**corr_coef**(*signal1*, *signal2*)

Finds the correlation coefficient between two SignalObjs using the numpy.corrcoef() function.

pytta.**peak_time**(*signal*)

Return the time at signal's amplitude peak.

pytta.**save**(*fileName: str = 'Fri Oct 21 03:23:25 2022'*, **PyTTaObjs*)

Main save function for .hdf5 and .pytta files.

The file format is chose by the extension applied to the fileName. If no extension is provided choose the default file format (.hdf5).

For more information on saving PyTTa objects in .hdf5 format see pytta.functions._h5_save documentation.

For more information on saving PyTTa objects in .pytta format see pytta.functions.pytta_save' documentation. (DEPRECATED)

pytta.**load**(*fileName: str*)

Main save function for .pytta and .hdf5 files.

pytta.**weighting**(*kind='A'*, *nth=None*, *freqs=None*)

Level weighting curve.

Parameters

- **kind** (*TYPE, optional*) – DESCRIPTION. The default is 'A'.
- **nth** (*TYPE, optional*) – DESCRIPTION. The default is None.
- **freqs** (*TYPE, optional*) – DESCRIPTION. The default is None.

Returns

The weighting curve in dB.

Return type

np.ndarray

pytta.**fft_degree**(**args*, ***kwargs*)

Being replaced by pytta.utils.maths.fft_degree on version 0.1.0.

Power-of-two value that can be used to calculate the total number of samples of the signal.

```
>>> numSamples = 2**fftDegree
```

Parameters

- **0)** (** timeLength (float =*) – Value, in seconds, of the time duration of the signal or recording.
- **1)** (** samplingRate (int =*) – Value, in samples per second, that the data will be captured or emitted.

Returns

Power of 2 that can be used to calculate number of samples.

Return type
fftDegree (float = 0)

pytta.**plot_time**(**sigObjs*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *timeUnit: str = 's'*)

Plot provided SignalObjs togheter in time domain.

Saves xLabel, yLabel, and title when provided for the next plots.

- **sigObjs (), (SignalObj):**
  non-keyworded input arguments with N SignalObjs.
- **xLabel (None), (str):**
  x axis label.
- **yLabel (None), (str):**
  y axis label.
- **yLim (None), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (None), (str):**
  left and right limits.

```
>>> xLim = [0, 15]
```

- **title (None), (str):**
  plot title.
- **decimalSep (','), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **timeUnit ('s'), (str):**
  'ms' or 's'.

matplotlib.figure.Figure object.

pytta.**plot_time_dB**(**sigObjs*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *timeUnit: str = 's'*)

Plot provided SignalObjs togheter in decibels in time domain.

- **sigObjs (), (SignalObj):**
  non-keyworded input arguments with N SignalObjs.
- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Amplitude'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  left and right limits

```
>>> xLim = [0, 15]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **timeUnit (‘s’), (str):**
  ‘ms’ or ‘s’.

matplotlib.figure.Figure object.

pytta.**plot_freq**(**sigObjs*, *smooth: bool = False*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*)

Plot provided SignalObjs magnitudes togheter in frequency domain.

- **sigObjs (), (SignalObj):**
  non-keyworded input arguments with N SignalObjs.
- **xLabel (‘Time [s]’), (str):**
  x axis label.
- **yLabel (‘Amplitude’), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  left and right limits

```
>>> xLim = [15, 21000]
```

- **title (), (str):**
  plot title
- **decimalSep (‘,’), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

matplotlib.figure.Figure object.

pytta.**plot_bars**(**analyses*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*, *barWidth: float = 0.75*, *errorStyle: Optional[str] = None*, *forceZeroCentering: bool = False*, *overlapBars: bool = False*, *color: Optional[list] = None*)

Plot the analysis data in fractinal octave bands.

- **analyses (), (SignalObj):**
  non-keyworded input arguments with N SignalObjs.
- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Amplitude'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior limits.

```
>>> yLim = [-100, 100]
```

- **xLim (), (list):**
  bands limits.

```
>>> xLim = [100, 10000]
```

- **title (), (str):**
  plot title
- **decimalSep (','), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

- **barWidth (0.75), float:**
  width of the bars from one fractional octave band. 0 < barWidth < 1.
- **errorStyle ('standard'), str:**
  error curve style. May be 'laza' or None/'standard'.
- **forceZeroCentering ('False'), bool:**
  force centered bars at Y zero.
- **overlapBars ('False'), bool:**
  overlap bars. No side by side bars of different data.
- **color (None), list:**
  list containing the color of each Analysis.

matplotlib.figure.Figure object.

pytta.**plot_spectrogram**(**sigObjs*, *winType: str = 'hann'*, *winSize: int = 1024*, *overlap: float = 0.5*, *xLabel: Optional[str] = None*, *yLabel: Optional[str] = None*, *yLim: Optional[list] = None*, *xLim: Optional[list] = None*, *title: Optional[str] = None*, *decimalSep: str = ','*)

Plots provided SignalObjs spectrogram.

- **sigObjs (), (SignalObj):**
  non-keyworded input arguments with N SignalObjs.
- **winType ('hann'), (str):**
  window type for the time slicing.

- **winSize (1024), (int):**
  window size in samples
- **overlap (0.5), (float):**
  window overlap in %
- **xLabel ('Time [s]'), (str):**
  x axis label.
- **yLabel ('Frequency [Hz]'), (str):**
  y axis label.
- **yLim (), (list):**
  inferior and superior frequency limits.

```
>>> yLim = [20, 1000]
```

- **xLim (), (list):**
  left and right time limits

```
>>> xLim = [1, 3]
```

- **title (), (str):**
  plot title
- **decimalSep (','), (str):**
  may be dot or comma.

```
>>> decimalSep = ',' # in Brazil
```

List of matplotlib.figure.Figure objects for each item in curveData.

pytta.**plot_waterfall**(**sigObjs*, *step=512*, *n=8192*, *fmin=None*, *fmax=None*, *pmin=None*, *pmax=None*, *tmax=None*, *xaxis='linear'*, *time_tick=None*, *freq_tick=None*, *mag_tick=None*, *tick_fontsize=None*, *fpad=1*, *delta=60*, *dBref=2e-05*, *fill_value='pmin'*, *fill_below=True*, *overhead=3*, *winAlpha=0*, *plots=['waterfall']*, *show=True*, *cmap='jet'*, *alpha=[1, 1]*, *saveFig=False*, *figRatio=[1, 1, 1]*, *figsize=(950, 950)*, *camera=[2, 1, 2]*)

This function was gently sent by Rinaldi Polese Petrolli.

TO DO

Keyword Arguments

- **(default** (*filtered {bool} -- [description]*) – {10})
- **(default** – {None})
- **(default** – {20})
- **(default** – {None})
- **(default** – {0})
- **(default** – {None})
- **(default** – {-72})
- **(default** – {14})
- **(default** – {'jet'})
- **(default** – {False})

- **(default** – {True})
- **(default** – {True})
- **(default** – {False})
- **(default** – {1})
- **(default** – {(20, 8)})
- **(default** – {0})
- **(default** – {False})
- **(default** – {False})
- **(default** – {False})
- **(default** – {0.70})
- **(default** – {3})
- **(default** – {None})
- **(default** – {True})

Returns
[type] – [description]

3.2.5 Generate

This submodule provides the tools for instantiating the measurement and signal objects to be used. We strongly recommend the use of this submodule instead of directly instantiating classes, except when necessary.

The signal generating functions already have set up a few good practices on signal generation and reproduction through audio IO interfaces, like silences at beginning and ending of the signal, as well as fade ins and fade out to avoid abrupt audio currents from flowing and causing undesired peaks at start/ending of reproduction.

On the measurement side, it tries to set up the environment by already giving excitation signals, or by generating a SWEEP from default values

User intended functions:

```
>>> pytta.generate.sin()
>>> pytta.generate.sweep()
>>> pytta.generate.random_noise()
>>> pytta.generate.impulse()
>>> pytta.generate.measurement()
```

For further information see the specific function documentation

@authors: - João Vitor Gutkoski Paes, joao.paes@eac.ufsm.br - Matheus Lazarin Alberto, mtslazarin@gmail.com

`pytta.generate.`**`sin`**(*Arms=0.5*, *freq=1000*, *timeLength=1*, *phase=6.283185307179586*, *samplingRate=44100*, *fftDegree=None*)

Generates a sine signal with the traditional parameters plus some PyTTa options.

- **Arms (float) (optional):**
  The signal's RMS amplitude.

```
>>> Apeak = Arms*sqrt(2);
```

- **freq (float) (optional):**
  Nothing to say;
- **timeLength (float) (optional):**
  Sine timeLength in seconds;
- **fftDegree (int) (optional);**
  2**fftDegree signal's number of samples;
- **phase (float) (optional):**
  Sine phase in radians;
- **samplingRate (int) (optional):**
  Nothing to say;

pytta.generate.**sweep**(*freqMin=None*, *freqMax=None*, *samplingRate=None*, *fftDegree=None*, *startMargin=None*, *stopMargin=None*, *method='logarithmic'*, *windowing='hann'*)

Generates a chirp signal defined by the "method" input, windowed, with silence interval at the beggining and end of the signal, plus a hanning fade in and fade out.

```
>>> x = pytta.generate.sweep()
>>> x.plot_time()
```

Return a signalObj containing a logarithmic chirp signal from 17.8 Hz to 22050 Hz, with a fade in beginning at 17.8 Hz time instant and ending at the 20 Hz time instant; plus a fade out beginning at 20000 Hz time instant and ending at 22050 Hz time instant.

The fade in and the fade out are made with half hanning window. First half for the fade in and last half for the fade out. Different number of points are used for each fade, so the number of time samples during each frequency is respected.

- freqMin (20), (float)
- freqMax (20), (float)
- samplingRate (44100), (int)
- fftDegree (18), (float)
- startMargin (0.3), (float)
- stopMargin (0.7), (float)
- method (logarithmic'), (string)
- windowing ('hann'), (string)

pytta.generate.**random_noise**(*kind='white'*, *samplingRate=None*, *fftDegree=None*, *startMargin=None*, *stopMargin=None*, *windowing='hann'*)

See *colored_noise*.

pytta.generate.**colored_noise**(*color: str = 'white'*, *samplingRate: Optional[int] = None*, *fftDegree: Optional[int] = None*, *numChannels: Optional[int] = None*, *startMargin: Optional[float] = None*, *stopMargin: Optional[float] = None*, *windowing: str = 'hann'*)

Power law noise generator.

Based on the algorithm in: Timmer, J. and Koenig, M.: On generating power law noise. Astron. Astrophys. 300, 707-710 (1995)

Generate random noise with respect to the *(1/f)**B* rate. f stands for frequency and B is an integer power.

The colors and its spectrum characteristics:

- **Purple | Differentiated:**
  - +6.02 dB/octave | +20 dB/decade | B = -2;
  - color: 'purple', 'diff', 'differentiated';
- **Blue | Azure:**
  - +3.01 dB/octave | +10 dB/decade | B = -1;
  - color: 'blue', 'azure'
- **White | Flat:**
  - +0.00 dB/octave | +0 dB/decade | B = 0;
  - color: 'white', 'flat';
- **Pink | Flicker:**
  - -3.01 dB/octave | -10 dB/decade | B = 1;
  - color: 'pink', 'flicker', '1/f';
- **Red | Brownian:**
  - -6.02 dB/octave | -20 dB/decade | B = 2;
  - color: 'red', 'brown', 'brownian';

The output signal will have *startMargin* silence at the beginning of the waveform, and *stopMargin* silence at the end.

There is a fade-in between the starting silence and the noise itself that occurs during 5% of the total noise duration.

@author: Chum4k3r

pytta.generate.**impulse**(*samplingRate=None*, *fftDegree=None*)

Generates a normalized impulse signal at time zero, with zeros to fill the time length

pytta.generate.**measurement**(*kind='playrec'*, *samplingRate=None*, *freqMin=None*, *freqMax=None*, *device=None*, *inChannels=None*, *outChannels=None*, **args*, ***kwargs*)

Generates a measurement object of type Recording, Playback and Recording, Transferfunction, with the proper initiation arguments, a sampling rate, frequency limits, audio input and output devices and channels

```
>>> pytta.generate.measurement(kind,
                               [lengthDomain,
                               fftDegree,
                               timeLength,
                               excitation,
                               outputAmplification],
                               samplingRate,
                               freqMin,
                               freqMax,
                               device,
                               inChannels,
                               outChannels,
                               comment)
```

The parameters between brackets are different for each value of the (kind) parameter.

```
>>> msRec = pytta.generate.measurement(kind='rec')
>>> msPlayRec = pytta.generate.measurement(kind='playrec')
>>> msFRF = pytta.generate.measurement(kind='frf')
```

The input arguments may be different for each measurement kind.

Options for (kind='rec'):

- **lengthDomain: 'time' or 'samples', defines if the recording length**
  will be set by time length, or number of samples;
- **timeLength: [s] used only if (domain='time'), set the duration of the**
  recording, in seconds;
- **fftDegree: represents a power of two value that defines the number of**
  samples to be recorded:

  ```
  >>> numSamples = 2**fftDegree
  ```

- samplingRate: [Hz] sampling frequency of the recording;
- freqMin: [Hz] smallest frequency of interest;
- freqMax: [Hz] highest frequency of interest;
- device: audio I/O device to use for recording;
- inChannels: list of active channels to record;
- comment: any commentary about the recording.

Options for (kind='playrec'):

- excitation: object of SignalObj class, used for the playback;
- outputAmplification: output gain in dB;
- samplingRate: [Hz] sampling frequency of the recording;
- freqMin: [Hz] smallest frequency of interest;
- freqMax: [Hz] highest frequency of interest;
- device: audio I/O device to use for recording;
- inChannels: list of active channels to record;
- **outChannels: list of active channels to send the playback signal, for**
  M channels it is mandatory for the excitation signal to have M columns in the timeSignal parameter.
- comment: any commentary about the recording.

Options for (kind='frf'):

- same as for (kind='playrec');
- regularization: [boolean] option for Kirkeby regularization

pytta.generate.**stream**(*IO='IO'*, *device=None*, *integration=None*, *samplingRate=None*, *inChannels=None*, *outChannels=None*, *duration=None*, *excitation=None*, *callback=None*, **args*, ***kwargs*)

3.2.6 Rooms

DEPRECATED

Being replaced by class pytta.RoomAnalysis on version 0.1.1.

Calculations compliant to ISO 3382-1 to obtain room acoustic parameters.

It has an implementation of Lundeby et al. [1] algorithm to estimate the correction factor for the cumulative integral, as suggested by the ISO 3382-1.

Use this module through the function 'analyse', which receives an one channel SignalObj or ImpulsiveResponse and calculate the room acoustic parameters especified in the positional input arguments. For more information check pytta.rooms.analyse's documentation.

Available functions:

```
>>> pytta.rooms.Analyse(SignalObj, ...)
>>> pytta.rooms.strength_factor(...)
>>> pytta.rooms.G_Lpe
>>> pytta.rooms.G_Lps
```

Authors:
João Vitor Gutkoski Paes, joao.paes@eac.ufsm.br Matheus Lazarin, matheus.lazarin@eac.ufsm.br Rinaldi Petrolli, rinaldi.petrolli@eac.ufsm.br

pytta.rooms.**G_Lpe**(*IR*, *nthOct*, *minFreq*, *maxFreq*, *IREndManualCut=None*)

Calculate the energy level from the room impulsive response.

Reference:
Christensen, C. L.; Rindel, J. H. APPLYING IN-SITU RECALIBRATION FOR SOUND STRENGTH MEASUREMENTS IN AUDITORIA.

Parameters

- **IR** (ImpulsiveResponse) – one channel impulsive response
- **nthOct** (*int*) – number of fractions per octave
- **minFreq** (*float*) – analysis inferior frequency limit
- **maxFreq** (*float*) – analysis superior frequency limit

Returns
Analysis object with the calculated parameter

Return type
Analysis

pytta.rooms.**G_Lps**(*IR*, *nthOct*, *minFreq*, *maxFreq*)

Calculates the recalibration level, for both in-situ and reverberation chamber. Lps is applied for G calculation.

During the recalibration: source height and mic height must be >= 1 [m], while the distance between source and mic must be <= 1 [m]. The distances must be the same for in-situ and reverberation chamber measurements.

Reference:
Christensen, C. L.; Rindel, J. H. APPLYING IN-SITU RECALIBRATION FOR SOUND STRENGTH MEASUREMENTS IN AUDITORIA.

Parameters

- **IR** (ImpulsiveResponse) – one channel impulsive response
- **nthOct** (*int*) – number of fractions per octave
- **minFreq** (*float*) – analysis inferior frequency limit
- **maxFreq** (*float*) – analysis superior frequency limit

Returns
Analysis object with the calculated parameter

Return type
Analysis

pytta.rooms.**strength_factor**(*Lpe*, *Lpe_revCh*, *V_revCh*, *T_revCh*, *Lps_revCh*, *Lps_inSitu*)

Calculate strength factor (G) for theaters and big audience intended places.

Reference:
Christensen, C. L.; Rindel, J. H. APPLYING IN-SITU RECALIBRATION FOR SOUND STRENGTH MEASUREMENTS IN AUDITORIA.

pytta.rooms.**analyse**(*obj*, **params*, *bypassLundeby=False*, *plotLundebyResults=False*, *suppressWarnings=False*, *IREndManualCut=None*, ***kwargs*)

Being replaced by class pytta.RoomAnalysis on version 0.1.1.

Room analysis over a single SignalObj.

Receives an one channel SignalObj or ImpulsiveResponse and calculate the room acoustic parameters especified in the positional input arguments. Calculates reverberation time, definition and clarity.

The method for strength factor calculation implies in many input parameters and specific procedures, as the sound source's power estimation. The pytta.roomir app was designed aiming to support this room parameter measurement. For further information check pytta.roomir's and pytta.rooms.strength_factor's docstrings.

- **obj (), (SignalObj | ImpulsiveResponse):**
  one channel impulsive response
- **non-keyworded argument pairs:**
  Pair for 'RT' (reverberation time):
  - **RTdecay (20), (int):**
    Decay interval for RT calculation. e.g. 20

  Pair for 'C' (clarity): # TODO
  - **Cparam (50), (int):**
    ...

  Pair for 'D' (definition): # TODO
  - **Dparam (50), (int):**
    ...
- **nthOct (), (int):**
  Number of bands per octave;
- **minFreq (), (int | float):**
  Analysis' inferior frequency limit;
- **maxFreq (), (int | float):**
  Analysis' superior frequency limit;

- **bypassLundeby (false), (bool):**
  Bypass lundeby correction
- **plotLundebyResults (false), (bool):**
  Plot the Lundeby correction parameters;
- **suppressWarnings (false), (bool):**
  Suppress the warnings from the Lundeby correction;
- **Analyses (Analysis | list):**
  Analysis object with the calculated parameter or a list of Analyses for more than one parameter.

Usage example:

```
>>> myRT = pytta.rooms.analyse(IR,
                               'RT', 20',
                               'C', 50,
                               'D', 80,
                               nthOct=3,
                               minFreq=100,
                               maxFreq=10000)
```

For more tips check the examples folder.

3.2.7 Properties

As to provide an user friendly signal measurement package, a few default values where assigned to the main classes and functions.

These values where set using a dict called "default", and are passed to all PyTTa functions through the Default class object

```
>>> import pytta
>>> pytta.default()
```

The default values can be set differently using both declaring method, or the set_default() function

```
>>> pytta.default.propertyName = propertyValue
>>> pytta.default.set_defaults(propertyName1 = propertyValue1,
>>>                            ... ,
>>>                            propertyNameN = propertyValueN
>>>                            )
```

The main difference is that using the set_default() function, a list of properties can be set at the same time

The default device start as the one set default at the user's OS. We recommend changing it's value to the desired audio in/out device, as it can be identified using list_devices() method

```
>>> pytta.list_devices()
```

@author: - João Vitor Gutkoski Paes, joao.paes@eac.ufsm.br

`pytta.default`

alias of <pytta._properties.Default object>

CHAPTER

FOUR

EXAMPLES

Some examples

4.1 Sweep and play

Create an exponential sine sweep from 50 Hz to 16 kHz

```
>>> swp = pytta.generate.sweep(freqMin=50, freqMax=16e3)
>>> swp.play()
```

4.2 Recording

Create a measurement object that records sound and is capable of calibration of levels.

```
>>> recms = pytta.generate.measurement('rec')
>>> rec = recms.run()
>>> rec.play()
```

4.3 Playback and Record

Create a measurement object that plays a signal and records microphone input simultaneously.

```
>>> prms = pytta.generate.measurement('playrec', excitation=swp)
>>> rec = prms.run()
>>> rec.play()
```

4.4 More examples

Check the toolbox's 'examples' folder.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

W