Audio Engineering Society

Conference Paper 23

Presented at the International Conference on Spatial and Immersive Audio
2023 August 23–25, Huddersfield, UK

The *Hartufo* toolkit for machine learning with HRTF data

Johan Pauwels[1]

[1]*Queen Mary University of London*

Correspondence should be addressed to Johan Pauwels (`j.pauwels@qmul.ac.uk`)

ABSTRACT

In this paper, the *Hartufo* Python toolkit is presented. Its aim is to provide an easy way to manage and preprocess HRTF data into a form that is suitable for use with all major machine learning tools. It consolidates typical boilerplate code into a single reusable library, in the hope that setting up experiments spanning multiple HRTF collections becomes easier, leading to novel insights. Additional benefits include increasing reproducibility and lowering the barrier to entry for machine learning and/or HRTF novices. Available as an open-source public beta, the majority of public HRTF collections are already supported, including auxiliary data such as photos and anthropometric measurements in addition to the auditory data. An overview of the library's functionality is given in this text, ranging from practical examples for end-users to a discussion about the internal concepts of the library for those who want to extend it or interleave it with existing code.

1 Introduction

As in many other audio processing domains, data-driven machine learning techniques are increasingly used in the study of Head-Related Transfer Functions (HRTFs). Crucial for these techniques is the availability of data, for training or validation. While there are a fair number of HRTF collections publicly available, they are very heterogeneous in terms of size, measurement characteristics and auxiliary data such as corresponding images and anthropometric information (see table 1 and [1]).

This heterogeneity requires collection-specific data preprocessing in order to transform the available data into a format that is suitable to be used with common machine learning toolkits such as PyTorch [2], Tensorflow [3] and Scikit-Learn [4]. As a consequence, all too often only single collections are used in studies, whereas inter-collection comparisons become even more important when using data-driven techniques due to their additional challenges regarding interpretability [5]. Furthermore, the necessity for formatting and preprocessing pipelines leads to duplicated effort between researchers, complicates reproducibility [6] and raises additional barriers to entry, equally for machine learning practitioners who lack audio processing experience as for acousticians and DSP engineers who lack machine learning expertise.

To address the points raised above, the Python library *Hartufo* has been created. It provides a unified programming interface to many HRTF collections, such that it becomes trivial to switch between them and combine them. Its design goals are to be (1) intuitive and convenient to use, (2) flexible and extensible to integrate into existing workflows, (3) compatible with a wide range of machine learning libraries and (4) following best practices.

The library is built on the standard scientific Python software stack consisting of NumPy [19], SciPy [20] and Matplotlib [21] and makes good use of the SOFA standard [22] as unified storage format for HRTFs, but extends this with a flexible processing pipeline to transform and potentially unify multiple collections into datasets that can be directly fed into all major machine learning libraries.

The choice for Python as implementation language is motivated by its popularity for machine learning. While there is a small overlap when it comes to visualisation and basic processing steps, the specific focus on machine learning and HRTFs sets *Hartufo* apart from more general Python libraries for spatial audio such as *pyfar*[1], *spaudiopy*[2], *Sound Field Analysis toolbox for Python*[3] and *safpy*[4].

Before we start, let's go over some of the terminology used in the remainder of this paper. An HRTF *collection* includes all data collected and published as a whole from a specific measurement setup. Collections are unstructured and non-uniform. *Dataset* is used in the strict machine-learning sense of the word, meaning structured, selected and preprocessed data that is ready to be used as input for a machine learning algorithm. The goal of *Hartufo* therefore is to create *datasets* from *collections*. Each dataset consists of multiple *datapoints*. Collections, and consequently datasets too, can contain multiple *types* of data. Typically auditory data are included, but also auxiliary data such as images, tabular anthropometric measurements and 3D models can be present. *HRTF* is used to refer to the set of all *HRIRs* that are measured for a single individual. Therefore its meaning is not strictly the frequency transformation of a HRIR. Most sections of the following text concern auditory data regardless of its actual *representation domain*, so both HRTF and HRIR will be used interchangeably there.

The data collections that are currently supported are displayed in table 1. For now, the focus lies on collections that contain human subjects, though collections of manikins only could easily be added in the future.

[1]https://pyfar.org/
[2]https://spaudiopy.readthedocs.io/en/latest/
[3]https://appliedacousticschalmers.github.io/sound_field_analysis-py/
[4]https://github.com/leomccormack/Spatial_Audio_Framework

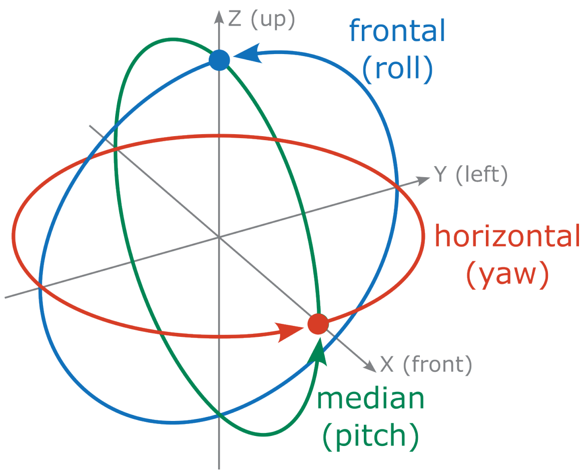

Fig. 1: A visualisation of the three principal planes used, along with their names and the origin, orientation and name of the angles in these planes.

2 Getting Started With Planar HRTF Data

2.1 Basic Data Access

At its most basic level, *Hartufo* can be used to access all HRIRs that are measured in the same plane. In order to unify access, independent of the 3D coordinate system that is used to store a collection, the convention displayed in fig. 1 is used. Three principal planes (*horizontal*, *median* and *frontal*) are defined, together with an origin, direction and name for the angle in each of the planes.

Next, the side of the head for which measurements will be retrieved needs to be decided. Any of the values *left*, *right*, *both*, *any*, *both-left*, *both-right*, *any-left*, *any-right* are possible. Selecting *both* returns left and right measurements only when both are available, whereas *any* returns all measurements regardless of the availability of the other side[5]. Values of the form **-left* return both left and right side measurements, but the right ones are mirrored to simulate additional left measurements, with the inverse applied for **-right*.

Finally, the representation domain of the HRTF needs to be selected out of *time*, *magnitude*, *magnitude_db*,

[5]In practice, all currently supported collections always provide both sides of the head for HRIRs, so there is no difference between *both* and *any*. The same selection mechanism is also used for the auxiliary data, however, where this is not the case. For instance, in some cases only a picture of one side of the head is provided.

Table 1: An overview of the data collections supported by *Hartufo* with the size and type of their constituting data. An “M” indicates the number of manikins that are measured. The “P”, “H” and “T” in the “3D models” column stand for “pinna”, “head” and “torso”, respectively, and indicate the type(s) of 3D model.

name	# HRTFs	3D models	2D images	anthropometry	acquisition method
CIPIC [7]	43 (+2M)		45 (+1M)	43 (+2M)	acoustic
Ircam Listen	51			49	acoustic
Ircam BiLi [8]	56				acoustic
Ircam CrossMod	24				acoustic
ARI [9]	224			60	acoustic
RIEC [10]	103 (+2M)	38 (+1M) PHT			acoustic
ITA[11]	48	48 P		47	acoustic
Princeton 3D3A [12]	36 (+2M)	30 (+2M) P, PH, PHT		30 (+2M)	acoustic & simulated
SADIE II [13]	18 (+2M)	18 PH	16 (+2M)		acoustic
SCUT [14]	10 (+1M)			10	acoustic
HUTUBS [15]	94 (+2M)	56 (+2M) PH		91 (+2M)	acoustic & simulated
CHEDAR [16]	1253	1253 PHT		1253	simulated
Widespread [17]	1005	1005 P			simulated
SONICOM [18]	200 (+1M)	200 PHT			acoustic

phase or *complex*. These choices can then be used together with classes from `hartufo.planar`. Each of the supported collections has a corresponding class `*Plane` , e.g. `CipicPlane` for the CIPIC collection. Passing a directory path together with the values discussed above and the option `download=True` ensures that the collection gets downloaded to that directory and loaded:

```
from hartufo.planar import CipicPlane
plane = 'median'
domain = 'magnitude_db'
side = 'both-left'
ds = CipicPlane('./cipic', plane,
    domain, side, download=True)
```

There’s also an option `verify=True`, which can be used to check the integrity of previously downloaded files on disk. By default, neither of the two options is run and an error is raised if the required files cannot be found in the given directory. Finally, the option `plane_offset` can be used to load HRIRs measured in planes parallel to any of the principal planes[6].

The type of the resulting variable `ds` is a subclass of `hartufo.HRTFDataset`. Its size can be obtained with `len(ds)`, where each datapoint corresponds to the combination of a subject and a side of the head. `HRTFDataset` classes support indexation to access individual datapoints, e.g. `ds[0]`. Each datapoint `p` is a Python dictionary, from which the HRTF data is available under the key *features*. It takes the form of a 2D array where each row corresponds to a single HRIR in the requested domain. The number of rows therefore equals the number of measurement positions in the plane.

[6]This is particularly relevant for the ITA collection, where no measurements in the exact horizontal plane are made, but `plane_offset=-0.72` loads the nearest plane.

Multiple datapoints can be accessed using the common Python slicing syntax, e.g. `ds[:5]`. The dictionary value of `features` is then a 3D array with the number of datapoints as the first dimension. As an alternative to accessing the data of the full slice, i.e. all datapoints `ds[:]['features']`, the *features* property of a dataset can be used: `ds.features`.

This minimal data access functionality is all you need to start using the *Hartufo* classes, but there’s a lot more functionality built-in.

2.2 Getting Dataset Info and Visualisation

Any planar dataset has a number of properties that provide information about the loaded HRTF data. The samplerate of the HRIRs

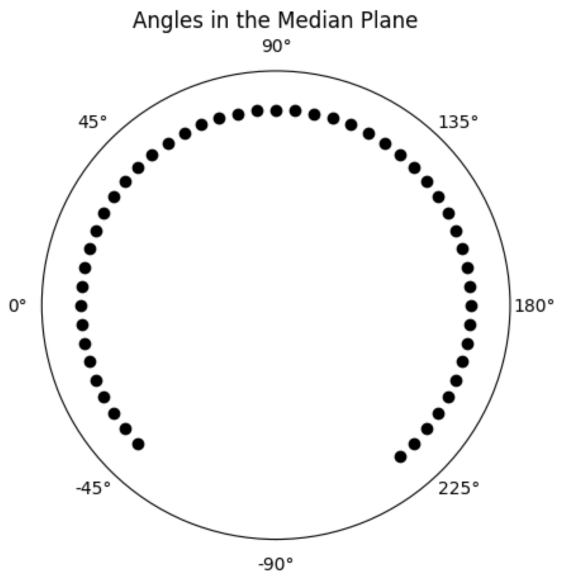

Fig. 2: Example of a plot visualising the HRTF measurement positions in a plane with the default options (concretely, the median plane of the CIPIC collection).

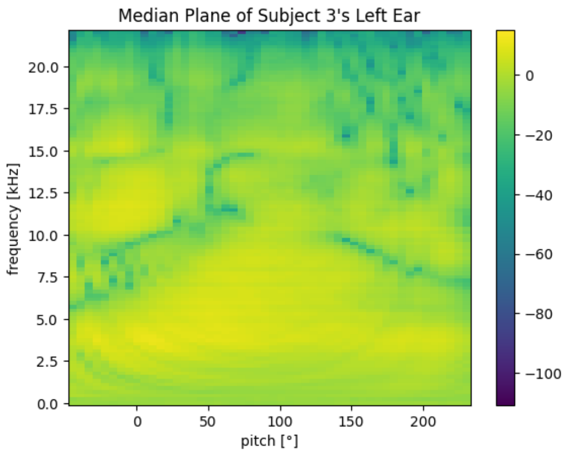

Fig. 3: Example of a plot showing HRTF data with default options (concretely the magnitude HRTF for the left ear of subject 3 in the CIPIC collection, measured in the median plane).

and corresponding frequency bins of the HRTF can be obtained as `ds.hrir_samplerate` and `ds.hrtf_frequencies`. A list of the subject IDs that correspond to each datapoint in a dataset can be obtained with `ds.subject_ids`. The angles of the measurement positions in the plane are retrieved with `ds.plane_angles`. By default, the interval for the angles is $[-180, 180)$ (horizontal and frontal plane) or $[-90, 27)$ (median plane). Positive angles in the range $[0, 360)$ can be requested by passing `positive_angles=True` to the dataset constructor or by setting the property of the same name after construction. In all cases, the angle extrema are available as `ds.min_angle` and `ds.max_angle`, and the name of the plane angle as `ds.plane_angle_name`.

Using this additional info, you could create your own visualisation, but plotting functionality is also included. You can plot the measurement positions in the plane with the `ds.plot_angles(ax=None, title=None)` method. By default, a new figure and a default title get created, but you can pass your own *Matplotlib* `Axes` instance or title string for more control. The created or extended `Axes` also gets returned by the method for further customisation. An example of a default measurement position plot is shown in fig. 2.

The HRIRs of a datapoint can be plotted by passing its index `i` to `ds.plot_plane(i)`. The default output can be seen in fig. 3, but many options for customisation are available. The `ax` and `title` options do the same as in the previous plot, and `Axes` are returned again. The visualisation automatically adapts to the selected representation domain. A colourmap and its extreme values can be specified with `cmap`, `vmin` and `vmax`, respectively. In absence of given values, the extreme values are set to the extrema in the dataset and the *viridis* colourmap is used. The colour bar that is shown by default can be disabled with `colorbar=False`.

The plane angles are plotted on a linear scale, so if the sampling of angles is non-uniform, certain angles will be drawn over larger areas in the plot than others. By default, the area up to halfway the next angle is filled with a uniform colour, resulting in a block-like appearance that can be used to visually inspect the distribution of angles in the plane. By passing `continuous=True`, intermediate angle values will be interpolated leading to a smooth picture. For frequency-domain HRTF representations, the option `log_freq=True` can be used to plot frequency on a logarithmic axis.

2.3 Customising Dataset Contents

To make a selection of the subjects, a `subject_ids` argument can be passed to the constructor, containing a sequence of subject IDs. In its absence, a default set is loaded, which generally corresponds to all human subjects. If no subject with the given ID exists, it gets silently skipped, but creating a completely empty dataset raises an error.

For any dataset, regardless of its contents, you can request the possible subject IDs you can pass to the constructor as `ds.available_subject_ids`. This makes the following workflow a convenient way to split a dataset into parts. Start by creating an empty dataset by passing an empty list or tuple to `subject_ids`, then read `ds.available_subject_ids` to find out what subjects are available and create another dataset with a subset of these IDs.

If just a single example of a data collection is needed, you can instead pass one of the strings *first*, *last* or *random* to `subject_ids`. The first two deterministically load the first, respectively last, ID in the collection, whereas *random* loads a random subject.

Similarly, a `plane_angles` argument can also be passed to the constructor to specify which measurement positions, specified in degrees, should be loaded. By default, the HRIR values get stored in `numpy.float32` format. It can be changed by passing a `dtype` argument to the constructor. When requesting an HRTF in the *complex* domain, specifying a complex `dtype` such as `numpy.complex64` is required, otherwise an error will be raised.

If only a few measurements per plane are requested (or present in the collection), the argument `lineplot=False` can be passed to `ds.plot_plane()` for a layered line-plot visualisation as in fig. 4.

2.4 Preprocessing Planar Data

Some common preprocessing steps for HRIRs are available as further constructor arguments, for instance to harmonise data from different collections as was done in [1]. Passing a number to `hrir_scaling` will multiply every HRIR with that number. HRIRs can be resampled by setting `hrir_samplerate` (using a

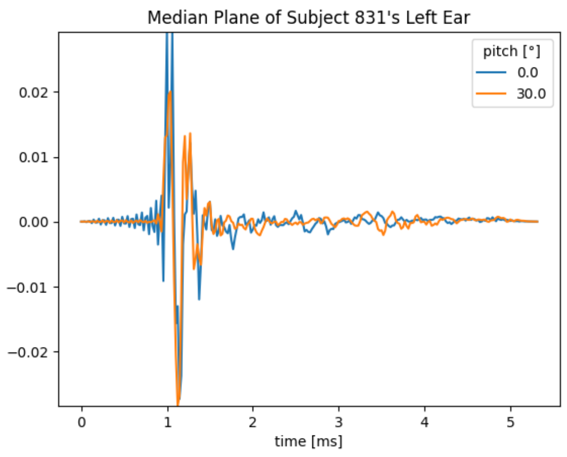

Fig. 4: Example of a layered line-plot showing HRIR data (concretely the HRIRs at angles $-30°, 0°$ and $30°$ in the median plane for the left ear of subject 3 in the CIPIC collection).

band-limited sinc interpolation implemented by *libsamplerate*[7]) and truncated to a maximum number of samples (after resampling) with `hrir_length`. By default, no resampling nor truncation takes place. Finally, `hrir_min_phase` accepts a boolean flag to indicate whether the minimum phase HRIR should be computed, which is not the case by default.

3 Creating Datasets for Supervised Learning

3.1 Different Roles of Data in a Dataset

Simply loading auditory data on its own already enables a multitude of data analysis and unsupervised learning workflows, but the true power of *Hartufo* lies in its capability to combine auxiliary data with auditory data to create datasets for supervised learning. As such, *Hartufo* is more of a dataset *builder* rather than just a dataset *loader*.

In order to explain this functionality, we first need to discuss the three different roles any type of data can have in a supervised learning dataset. First, we have the input data, known as the *features*, which we feed into an algorithm that aims to learn how to predict some output, known as the *target*. These two roles

[7] http://libsndfile.github.io/libsamplerate/

are necessarily present in a supervised learning dataset. A third, optional role is that of *grouping information*. During the training process, a dataset typically get split into parts, but this split is rarely completely random. For instance, when splitting HRTF data, it is good practice to ensure that measurements made on the same subject (such as left and right ear HRIRs) are kept in the same split, because these are not independent datapoints. Otherwise an algorithm might learn to rely on the presence of partial data of a subject in order to predict some property, which is undesirable because it does not scale to subjects that are not present in the dataset.

Unless requested otherwise, planar auditory data is loaded as *features*, through the default constructor argument `hrir_role='features'`. We've already seen that this makes the auditory data available under the *feature* key of the dictionary for a datapoint `p`. We can also pass the string value *target* or *group* to the `hrir_role` constructor argument. In that case, auditory data can be accessed through `p['target']` or `p['group']`, respectively. By default, the values associated to these dictionary keys are left empty. Similar to the `ds.features` property, properties `ds.target` and `ds.group` are available as alternatives to `ds[:]['target']` and `ds[:]['group']`.

3.2 Data Specifications

Having the flexibility to move auditory data to any of three roles, auxiliary data can fill in the two remaining roles. To describe what piece of info in a collection gets what role assigned, we need to define *specifications* or *specs* for each of the roles. These take the form of nested dictionaries in which the first level can be any combination of the keys below, listed together with a description of the type of data that will be made available for each datapoint.

collection a string identifier of the name of the data collection
subject an integer giving the identifier of the subject in the collection
side a string indicating the side of the head (*left*, *right*, *mirrored-left* or *mirrored-right*)
hrirs an array containing the HRIRs in a certain domain for the given positions and side of the head
image an array containing the pixel values of an image associated with a given side of the head
anthropometry an array containing anthropometric measurements
3d-model an array containing a 3D model

The first four pieces of information are available for every collection, whereas the availability of the latter three depends on the collection, as shown in table 1. The fourth key, *hrirs*, loads the same auditory data as specified in the `*Plane` constructor. Therefore it is superfluous in this context, but is mentioned for the use of specs in later contexts.

The values for each of these keys in the specs are Python dictionaries themselves, to allow passing optional parameters for each type of data. The first three keys *collection*, *subject* and *side* have no parameters, therefore always receive an empty dictionary `{}` as value.

3.3 Specifying Image and Anthropometric Data

Support for loading 3D models as auxiliary data is planned, but not currently implemented. Associated images and anthropometric data can already be loaded, however, and have a number of options that will be explained next. Both *image* and *anthropometry* accept a *side* key in their parameter dictionary (the second level of a nested spec dict), e.g. `{'image': {'side': 'any'}}`. It can take the same values as the `side` constructor argument introduced in section 2.1. In absence of an explicitly defined value, the same one as requested elsewhere will be used and if no *side* is defined anywhere, it will default to *any*. Note that these values are different from the ones that are returned as part of a datapoint when *side* is part of a spec: you can request *both* or *any* sides, but a datapoint can only correspond to the (mirrored) left or right side of the head.

Some collections (e.g. CIPIC) also accept a boolean parameter *rear* for *images*, to indicate whether a rear view of the requested side should be loaded. In any case, the associated image gets loaded as a variable of type `PIL.Image.Image` [23].

The *anthropometric* key accepts a *select* parameter, which should be one or more of *head-torso*, *pinna-size*, *pinna-angle*, *weight*, *age* or *sex* (with additional values for some collections like ARI). This can be used to select specific subset of the available anthropometric data. The entire set is used by default and the data for each

```
from hartufo.planar import AriPlane, CipicPlane
ds1 = AriPlane('./ari', plane='horizontal', domain='magnitude_db', side='both',
    subject_id='last', hrir_role='features', other_specs={
        'target_spec': {'collection': {}},
        'group_spec': {'subject': {}},
    }
)
ds2 = CipicPlane('./cipic', plane='median', domain='time', side='left',
    subject_id=(3,), hrir_role='target', plane_angles=(0, 30), hrir_min_phase=True,
    other_specs={
        'features_spec': {'image': {}, 'anthropometry': {'partial': True}},
        'group_spec': {'subject': {}},
    }
)
```

Listing 1: Two examples of datasets for supervised learning using planar HRTFs and auxiliary data.

datapoint is returned as a 1D array. The boolean *partial* parameter influences what happens with datapoints that only have partial anthropometric data. A value of `True` sees the missing data replaced with `numpy.nan`, `False` causes the datapoint to be dropped from the dataset.

3.4 Example Supervised Learning Datasets

To combine the dictionary-formatted specifications with the planar HRTF data, the specs for the remaining roles (i.e. those not equal to the value of `hrir_role`) need to be named either *feature_spec*, *target_spec*, *group_spec*. These names are then to be used as keys in a dictionary with the respective specs as values (thereby creating a three-level nested dictionary) and passed to the `other_specs` argument of an `*Plane` constructor.

An example should make this clearer. Variable `ds1` in listing 1 contains a dataset that has magnitude HRTFs in the horizontal plane for both sides as features, expressed in decibels, and the name of the collection (ARI) as target label. The subject index is returned as group info such that both ears of the same subject can end up in the same split. Variable `ds2` contains a dataset where the minimum-phase HRIRs of angles 0° and 30° in the median plane are used as target to be synthesised from left ear images and anthropometric data, similar to what was done in [24] for instance. In such cases where multiple keys are used in a spec, the corresponding components are grouped into a tuple for each datapoint.

4 Accessing the Full Sphere of HRTF Measurement Positions

4.1 Internal Data Layout

One of the benefits of the `*Plane` interface is that it does not require knowledge about the spatial distribution of measurements present in a collection (and its natural coordinate system). Unfortunately, accessing all measurements on a sphere, or an arbitrary selection thereof, requires some knowledge about the collection and the internal HRIR layout of *Hartufo*. Regardless of the precise sampling scheme, we can define the *fundamental plane* used in a collection and its *normal axis*, where the fundamental plane is the principal plane that is taken as the reference plane for the measurements, meaning that all spatial positions can be thought of as lying on planes parallel to this reference.

In reality, the CIPIC collection is the only one known that has the median plane as its fundamental plane, all other collections use the horizontal plane. It is most natural to express the latter as (azimuth, elevation) coordinates, whereas the former are most easily expressed in (lateral, vertical) coordinates (following the terminology in [26]), though conversions to other coordinate systems can obviously be made. The spatial distribution arises in practice from rotating subject or loudspeakers around the normal axis [25].

HRIRs for a single datapoint are stored internally as a 3-dimensional array, where the first two dimensions store the spatial position and the third the sample values. The rows are ordered by increasing angle in the fundamental plane (vertical angles for CIPIC, azimuths for all other), constrained to the interval $[-180, 180)$. The columns of the array are ordered by increasing angle in the *orthogonal* plane, which is the plane defined by the normal axis and the spatial position. The *orthogonal angles* are constrained to the interval $[-90, 90]$ (lateral angles for CIPIC, elevations for all the rest).

This internal layout means that the spherical positions are stored as a *plate carrée* projection. All HRIRs in the fundamental plane or in one of its parallel planes are readily available in this representation, whereas the other two principal planes and their parallel planes need to be stitched together from two half-plane representations (which is what the `*Plane` classes do). The first and last column are always the positions closest to the poles, regardless of the collection, although the measurements for the poles are not necessarily present (in the collection or just in a particular selection).

If we think of the unique *fundamental* (in the rows) and *orthogonal* angles (in the columns) as forming a matrix of available positions, then this matrix is not necessarily dense. Not every collection has chosen a spatial sampling where every combination is measured. Consequently, the 3D array of HRIRs is stored as a NumPy masked array, where combinations that have not been measured are masked.

4.2 Generalised Dataset API

Each of the supported collections also has a corresponding `HRTFDataset` class in `hartufo.full`, e.g. `Cipic` for the CIPIC collection. These are the generalised versions of the corresponding `*Plane` classes and provide more flexibility, but are slightly less convenient to use. Plane-specific properties and methods introduced in section 2, such as `ds.plane_angles`, `ds.plane_angle_name`, `ds.plot_plane()` and `ds.plot_angles()`, are unavailable.

All HRIR-specific arguments have been removed from the constructor too, which only takes `features_spec`, `target_spec`, `group_spec`, `subject_ids` and `dtype` as arguments. Instead of passing `domain` and `side` to the constructor, they should be used as options for the *hrirs* key of the relevant spec. The combination of `plane`, `plane_offset` and `plane_angles` constructor arguments is replaced by the more flexible `fundamental_angles` and `orthogonal_angles` parameters, again to be passed to the *hrirs* spec. Both parameters expect identically sized sequences of angles in degrees, where spatial positions to be loaded are defined by the coordinates obtained by pairing elements of the same index in both lists. Either list can be **`None`** too, in which case all available positions with the given fundamental/orthogonal angle(s) will be returned. If both values are **`None`** (or the parameters not defined in the spec), all available positions will be loaded.

Finally, HRIR-specific processing can be requested by passing more parameters to the *hrirs* spec. The parameter keys corresponding to the arguments in section 2.4 are *scale_factor*, *samplerate*, *length* and *min_phase*. The example in listing 2 illustrates this whole process. The variable `ds3` contains the same underlying HRIR data as `ds2` in listing 1.

5 Further Processing of Data

Any of the loaded data can be further processed without needing to modify any internal code. To that end, inspiration was taken from Scikit-Learn's *Pipeline API* [27]. Custom *callables* can be passed as value for the *preprocess* or *transform* key in any spec. A callable is either a simple function or a *functor*, a class that implements the `__call__()` method. In either case, the callable accepts a single argument, which is a single datapoint corresponding to a spec key. So for a *hrirs* spec, this is a (potentially masked) 3D array, for an *image* spec a Pillow `Image`, for *anthropometry* a 1D array and a string for other keys. The callable then returns the modified data, which allows them to be chained.

A sequence of callables can be passed to the *preprocess* and *transform* parameters of any spec. Both produce the same result, the only difference is when the callable gets executed. A callable passed to *preprocess* will be run during the construction of the dataset. This means it will be executed once and its output will be stored as the values of the dataset. For instance, if you save a dataset to disk, then load it again, it is the output of the callable that is loaded, not the original data. On the other hand, callables passed to *transform* are run every time a datapoint is accessed, e.g. through indexing or slicing. This means that the output of the callable is returned but never stored, the original data gets preserved. Therefore the latter is suitable for stochastic transformations, for instance random cropping of images, whereas the former is strictly deterministic.

As an example, suppose that the decibel conversion of the HRTF magnitudes was not built-in, then it could be added by passing the function below:

```
from hartufo.full import Cipic
ds3 = Cipic('./cipic', subject_id=(3,),
    features_spec = {'image': {}, 'anthropometry': {'partial': True}},
    target_spec = {'hrirs': {'domain': 'time', 'side': 'left', 'min_phase': True,
                   'fundamental_angles': (0, 30), 'orthogonal_angles': (0, 0)}},
    group_spec = {'subject': {}},
)
```

Listing 2: Example usage of the generalised dataset API, which loads the same underlying data as the `ds2` planar dataset in listing 1.

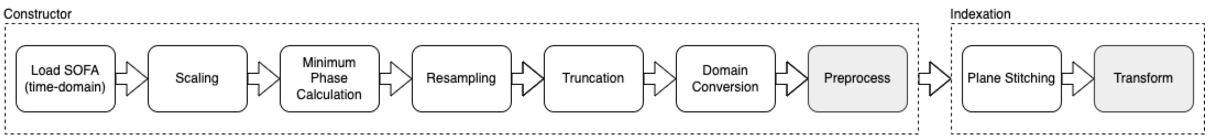

Fig. 5: Visualisation of the default processing pipeline for HRIRs. The plane stitching operation is only present for `*Plane` classes. The grey boxes indicate the location where custom callables can be inserted.

```
import numpy as np
def db_calc(hrtfs):
    return 20*np.log10(hrtfs)

ds4 = ARI('./ari', features_spec={
    'hrirs': {
        'domain': 'magnitude',
        'preprocess': db_calc,
    }
})
```

All preprocessing operations defined in section 2.4 are internally implemented as a sequence of callables, and are organised as shown in fig. 5. The individual callables can be loaded from `hartufo.transforms.hrirs`. Because the constructor arguments for planar datasets or corresponding spec params for generalised datasets by default switch all processing off, the external availability of processing callables allows for a full reassembly of the processing pipeline, rearranging their order and interleaving custom operations.

The returned data does not necessarily need to have the same shape as the input. For instance, this is what happens in the plane stitching of the `*Plane` classes, where 2D planes are returned. Internally, the `plane` argument of the constructor is converted into a set of fundamental and orthogonal angles, which get read into 3D arrays and subsequently stitched together into a single plane to form an output of two dimensions. With such operations, however, one needs to be careful to adjust the input of potential later callables in the pipeline to the new dimensions.

6 Example Integrations

6.1 PyTorch

All instances of `HRTFDataset` have a class interface that is directly compatible with PyTorch Datasets. They can therefore be directly used to create a `torch.util.data.DataLoader`. However, because `HRTFDataset` returns datapoints in a dict format (such that optional grouping info can be passed), the provided `hartufo.pytorch.collate_dict_dataset` collation function is required when creating a `DataLoader` to convert the dataset into expected `(feature, target)` pairs:

```
from hrtfdata.torch import \
    collate_dict_dataset
from torch.utils.data import DataLoader
loader = DataLoader(ds1,
    collate_fn=collate_dict_dataset)
features, target = next(iter(loader))
```

Other `torch.util.data` functionality can be used too, to chain or concatenate datasets, take subsets or use custom samplers, for instance.

6.2 Scikit-Learn

Scikit-Learn expects all datapoints to be passed together as inputs, for which the `ds.features`, `ds.target` and `ds.group` properties are suitable. It also requires 2D tabular data, therefore a `Flatten` functor is available in

```
from hartufo.full import Cipic
from hartufo.sklearn.estimators import Flatten, DomainConverter
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV
ds = Cipic('./cipic', feature_spec={'domain': 'time', 'side': 'both-left'},
    target_spec={'side': {}})
pipe = Pipeline([
    ('domain', None),
    ('flatten', Flatten()),
    ('reduce_dim', None),
    ('clf', LinearSVC()),
])
param_grid = {
    'domain': [DomainConverter('time'), DomainConverter('magnitude')],
    'reduce_dim': ['passthrough', PCA(n_components=None), PCA(n_components=3)],
}
exp = GridSearchCV(pipe, param_grid, scoring='accuracy').fit(ds.features, ds.target)
exp.cv_results_
```

Listing 3: Example Scikit-Learn cross-validation grid-search based on a pipeline mixing *Hartufo* and *sklearn* data processing.

`hartufo.sklearn.transforms`, which takes care of this and is compatible with Scikit-Learn's *transformer* API. All built-in data processing callables are also available in a version that is *transformer*-compatible, such that complicated Scikit-Learn native pipelines can be set up, mixing them with Scikit-Learn provided transformers such as `sklearn.decomposition.PCA`.

The example in listing 3 illustrates how such a hybrid processing pipeline can be combined with a cross-validated grid-search to easily examine which domain is more suitable for predicting the side of the head from HRIRs with a linear SVM, and whether PCA decomposition of the features influences classification accuracy. As can be expected for this contrived toy example, the time domain is a far more suitable representation when trying to predict at which side of the head the HRIRs where measured (with 97% accuracy compared to 89%) and PCA is detrimental for the accuracy when the number of retained components gets limited.

6.3 Additional Examples

More examples can be found as Jupyter Notebooks in the code repository, including an example integration with Tensorflow. The code for the study in [1] on the combination of multiple data collections, which was performed using *Hartufo*, is also fully available. Combining collections can be trivially done in practice using native operations for each machine learning library, e.g. NumPy concatenation for Scikit-Learn and `torch.utils.data.ConcatDataset` for PyTorch. However, just because it is technically possible does not mean that it is wise to do so without further harmonisation. A complete discussion of potential pitfalls can be found in [1].

7 Conclusion

The *Hartufo* toolkit has been presented in this paper, from its basic functionality of loading planes of HRTF measurements, over including auxiliary data to construct datasets for supervised learning, to advanced workflows mixing custom extensions with existing machine learning tools. Hopefully, the availability of this new toolkit will lead to an increase in quantity and quality of data-driven HRTF studies, since it removes a practical barrier to combining multiple collections in experiments. The library is available as a public beta from the Python Package Index, so can be installed with `pip install [--user] hartufo`. Its source code is published under the MIT open-source licence on GitHub[8]. Future efforts will be concentrated on increasing the number of supported datasets, implementing the proposed handling of 3D models and adding more options for data preprocessing.

[8]https://github.com/jpauwels/hartufo

Acknowledgements

This research has been partly funded by the SONICOM project (EU Horizon 2020 RIA grant agreement ID: 101017743).

References

[1] Pauwels, J. and Picinali, L., "On The Relevance Of The Differences Between HRTF Measurement Setups For Machine Learning," in *Proceedings of ICASSP*, 2023.

[2] Paszke, A. et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Proceedings of the 33rd NeurIPS*, Curran Associates Inc., 2019.

[3] Abadi, M. et al., "TensorFlow: Large-scale Machine Learning on Heterogeneous Systems," 2015.

[4] Pedregosa, F. et al., "Scikit-Learn: Machine Learning in Python," *Journal of Machine Learning Research*, 12(85), 2011.

[5] Murdoch, W. J., Singh, C., Kumbier, K., Abbasi-Asl, R., and Yu, B., "Definitions, Methods, and Applications in Interpretable Machine Learning," *Proceedings of the National Academy of Sciences*, 116(44), 2019.

[6] Vandewalle, P., Kovacevic, J., and Vetterli, M., "Reproducible Research in Signal Processing," *IEEE Signal Processing Magazine*, 26(3), 2009.

[7] Algazi, V., Duda, R., Thompson, D., and Avendano, C., "The CIPIC HRTF Database," in *Proceedings of WASPAA*, 2001.

[8] Carpentier, T., Bahu, H., Noisternig, M., and Warusfel, O., "Measurement of a Head-Related Transfer Function Database with High Spatial Resolution," in *7th Forum Acusticum*, 2014.

[9] Majdak, P., Goupell, M. J., and Laback, B., "3-D Localization of Virtual Sound Sources: Effects of Visual Environment, Pointing Method, and Training," *Attention, Perception, & Psychophysics*, 72(2), 2010.

[10] Watanabe, K., Iwaya, Y., Suzuki, Y., Takane, S., and Sato, S., "Dataset of Head-Related Transfer Functions Measured with a Circular Loudspeaker Array," *Acoustical Science and Technology*, 35(3), 2014.

[11] Bomhardt, R., de la Fuente Klein, M., and Fels, J., "A High-Resolution Head-Related Transfer Function and Three-Dimensional Ear Model Database," *Proceedings of Meetings on Acoustics*, 29(1), 2016.

[12] Sridhar, R., Tylka, J. G., and Choueiri, E., "A Database of Head-Related Transfer Functions and Morphological Measurements," in *Proceedings of the 143rd AES Convention*, 2017.

[13] Armstrong, C., Thresh, L., Murphy, D., and Kearney, G., "A Perceptual Evaluation of Individual and Non-Individual HRTFs: A Case Study of the SADIE II Database," *Applied Sciences*, 8(11), 2018.

[14] Yu, G., Wu, R., Liu, Y., and Xie, B., "Near-Field Head-Related Transfer-Function Measurement and Database of Human Subjects," *Journal of the ASA*, 143(3), 2018.

[15] Brinkmann, F., Dinakaran, M., Pelzer, R., Grosche, P., Voss, D., and Weinzierl, S., "A Cross-Evaluated Database of Measured and Simulated HRTFs Including 3D Head Meshes, Anthropometric Features, and Headphone Impulse Responses," *Journal of the AES*, 67(9), 2019.

[16] Ghorbal, S., Bonjour, X., and Séguier, R., "Computed HRIRs and Ears Database for Acoustic Research," in *Proceedings of the 148th AES Convention*, 2020.

[17] Guezenoc, C. and Séguier, R., "A Wide Dataset of Ear Shapes and Pinna-Related Transfer Functions Generated by Random Ear Drawings," *Journal of the ASA*, 147(6), 2020.

[18] Engel, I., Daugintis, R., Vicente, T., Hogg, A. O. T., Pauwels, J., Tournier, A. J., and Picinali, L., "The SONICOM HRTF Dataset," *Journal of the AES*, 71(5), 2023.

[19] Harris, C. R. et al., "Array Programming with NumPy," *Nature*, 585(7825), 2020.

[20] Virtanen, P. et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, 17(3), 2020.

[21] Hunter, J. D., "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, 9(3), 2007.

[22] Majdak, P. et al., M., "Spatially Oriented Format for Acoustics: A Data Exchange Format Representing Head-Related Transfer Functions," in *Proceedings of the 134th AES Convention*, 2013.

[23] Clark, J. A. and Contributors, "The Pillow Handbook," https://pillow.readthedocs.io/en/stable/handbook/index.html, 2010.

[24] Zhao, M., Sheng, Z., and Fang, Y., "Magnitude Modeling of Personalized HRTF Based on Ear Images and Anthropometric Measurements," *Applied Sciences*, 12(16), 2022.

[25] Li, S. and Peissig, J., "Measurement of Head-Related Transfer Functions: A Review," *Applied Sciences*, 10(14), 2020.

[26] Iida, K., *Head-Related Transfer Function and Acoustic Virtual Reality*, Springer Singapore, 2019.

[27] Buitinck, L. et al., "API Design for Machine Learning Software: Experiences from the Scikit-Learn Project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013.