Memo: CalH5 file format

Bryna Hazelton, and the pyuvdata team

October 5, 2023

1 Introduction

This memo introduces a new HDF5[1] based file format of a `UVCal` object in `pyuvdata`[2] a python package that provides package that provides an interface to interferometric data. UVCal is an object that supports calibration solutions and metadata for interferometric telescopes. Here, we describe the required and optional elements and the structure of this file format, called *CalH5*.

We assume that the user has a working knowledge of HDF5 and the associated python bindings in the package `h5py`[3], as well as `UVCal` objects in `pyuvdata`. For more information about HDF5, please visit `https://portal.hdfgroup.org/display/HDF5/HDF5`. For more information about the parameters present in a `UVCal` object, please visit `https://pyuvdata.readthedocs.io/en/latest/uvcal.html`. Examples of how to interact with `UVCal` objects in `pyuvdata` are available at `https://pyuvdata.readthedocs.io/en/latest/tutorial.html`.

Note that throughout the documentation, we assume a row-major convention (i.e., C-ordering) for the dimension specification of multi-dimensional arrays. For example, for a two-dimensional array with shape (N, M), the M-dimension is varying fastest, and is contiguous in memory. This convention is the same as Python and the underlying C-based HDF5 library. Users of languages with the opposite column-major convention (i.e., Fortran-ordering, seen also in MATLAB and Julia) must transpose these axes.

2 Overview

A CalH5 file contains calibration solutions for a radio telescope, as well as the associated metadata necessary to interpret it. A CalH5 file contains two primary HDF5 groups: the `Header` group, which contains the metadata, and the `Data` group, which contains the gains or delays as well as flags and measures of calibration quality (optional). Datasets in the

[1]`https://www.hdfgroup.org/`
[2]`https://github.com/RadioAstronomySoftwareGroup/pyuvdata`
[3]`https://www.h5py.org/`

`Data` group are also typically passed through HDF5's compression pipeline, to reduce the amount of on-disk space required to store the data. However, because HDF5 is aware of any compression applied to a dataset, there is little that the user has to explicitly do when reading data. For users interested in creating new files, the use of compression is not strictly required by the CalH5 format, again because the HDF5 file is self-documenting in this regard, but compression is quite common.

In the discussion below, we discuss required and optional datasets in the various groups. We note in parenthesis the corresponding attribute of a UVCal object. Note that in nearly all cases, the names are coincident, to make things as transparent as possible to the user.

3 Header

The `Header` group of the file contains the metadata necessary to interpret the data. We begin with the required parameters, then continue to optional ones. Unless otherwise noted, all datasets are scalars (i.e., not arrays). The precision of the data type is also not specified as part of the format, because in general the user is free to set it according to the desired use case (and HDF5 records the precision and endianness when generating datasets). When using the standard `h5py`-based implementation in pyuvdata, this typically results in 32-bit integers and double precision floating point numbers. Each entry in the list contains **(1)** the exact name of the dataset in the HDF5 file, in boldface, **(2)** the expected datatype of the dataset, in italics, **(3)** a brief description of the data, and **(4)** the name of the corresponding attribute on a UVCal object. Note that unlike in other formats, names of HDF5 datasets can be quite long, and so in most cases the name of the dataset corresponds to the name of the UVCal attribute.

Note that string datatypes should be handled with care. See Appendix A in the UVH5 memo[4] for appropriately defining them for interoperability between different HDF5 implementations.

3.1 Required Parameters

- **cal_type**: *string* The calibration type, supported options are "gain" or "delay". (*cal_type*)
- **cal_style**: *string* The calibration style, supported options are "sky" or "redundant". (*cal_style*)
- **gain_convention**: *string* The convention for applying the calibration solutions to data. Supported options are "divide" or "multiply", indicating that to calibrate one

[4]https://github.com/RadioAstronomySoftwareGroup/pyuvdata/blob/main/docs/references/uvh5_memo.pdf

should divide or multiply uncalibrated data by gains. Mathematically this indicates the α exponent in the equation:

$$v_{ij,\ calibrated} = g_i^\alpha g_j^\alpha * v_{ij,\ uncalibrated} \tag{1}$$

A value of “divide” represents $\alpha = -1$ and “multiply” represents $\alpha = 1$. (*gain_convention*)

- **wide_band**: *python bool*[5] Indicates whether this is a wide band calibration solutions with gains or delays that apply over a range of frequencies rather than having distinct values at each frequency. Delay type calibration solutions are always wide band. If it is True several other header items and data sets are affected: the data-like arrays have a spectral window axis that is Nspws long rather than a frequency axis that is Nfreqs long; the freq_range header item is required and the freq_array and channel_width header options should not be present. (*wide_band*)
- **latitude**: *float* The latitude of the telescope site, in degrees. (*latitude*)
- **longitude**: *float* The longitude of the telescope site, in degrees. (*longitude*)
- **altitude**: *float* The altitude of the telescope site, in meters. (*altitude*)
- **telescope_name**: *string* The name of the telescope used to take the data. The value is used to check that metadata is self-consistent for known telescopes in pyuvdata. (*telescope_name*)
- **x_orientation**: *string* The orientation of the x-arm of a dipole antenna. It is assumed to be the same for all antennas in the dataset. Supported options are “east” or “north”. (*x_orientation*).
- **Nants_telescope**: *int* The number of antennas in the array. May be larger than the number of antennas with data corresponding to them. (*Nants_telescope*)
- **antenna_numbers**: *int* An array of the numbers of the antennas present in the radio telescope (note that these are not indices, they do not need to start at zero or be continuous). This is a one-dimensional array of size Nants_telescope. Note there must be one entry for every antenna in ant_array, but there may be additional entries. (*antenna_names*)
- **antenna_names**: *string* An array of the names of antennas present in the radio telescope. This is a one-dimensional array of size Nants_telescope. Note there must be one entry for every antenna in ant_array, but there may be additional entries. (*antenna_names*)

[5] Note that this is *not* the same as the `H5T_NATIVE_HBOOL` type; instead, it is an `H5Tenum` type, with an explicit `TRUE` and `FALSE` value. Such a type is created automatically when using `h5py`, both for Python `bool` and numpy `np.bool_` types. See the UVH5 memo, Appendix C for an example of how to define this in C. Such a definition should follow analogously in other languages.

- **Nants_data**: *int* The number of antennas that have calibration data in the file. May be smaller than the number of antennas in the array. (*Nants_data*)
- **ant_array**: *int* An array of the antenna numbers corresponding to calibration solutions present in the file. All entries in this array must exist in the antenna_numbers array. This is a one-dimensional array of size Nants_data. (*ant_array*)
- **Nspws**: *int* The number of spectral windows present in the data. (*Nspws*)
- **Nfreqs**: *int* The total number of frequency channels in the data across all spectral windows. Should be 1 for wide band calibration solutions. (*Nfreqs*)
- **spw_array**: *int* An array of the spectral windows in the file. This is a one-dimensional array of size Nspws. (*spw_array*)
- **Njones**: *int* Number of Jones calibration parameters in data. (*Njones*)
- **jones_array**: *int* An array giving the Jones calibration parameters contained in the file. This is a one-dimensional array of size Njones. Note that the Jones parameters should be stored as an integer with the following mapping:
  - linear pols: -5 to -8 denoting: jxx, jyy, jxy, jyx
  - circular pols: -1 to -4 denoting: jrr, jll. jrl, jlr
  - unknown: 0

  (*jones_array*)
- **Ntimes**: *int* The number of time samples present in the data. (*Ntimes*)
- **integration_time**: *float* Integration time of a calibration solution, units seconds. This is a one-dimensional array of size Ntimes. Should be the total integration time of the data that went into calculating the calibration solution (i.e. the visibility integration time for calibration solutions that are calculated per visibility integration, the sum of the integration times that go into a calibration solution that was calculated over a range of integration times). (*integration_time*)
- **history**: *string* The history of the data file. (*history*)

3.2 Optional Parameters

- **telescope_frame**: *string* The coordinate frame for the telescope. Supported options are "itrs" for telescopes on earth or "mcmf" for telescopes on the moon. Not required but encouraged, assumed to be "itrs" if not specified. (*telescope_frame*)

- **instrument**: *string* The name of the instrument, typically the telescope name. (*instrument*)
- **antenna_diameters**: *float* An array of the diameters of the antennas in meters. This is a one-dimensional array of size (Nants_telescope,). (*Nants_telescope*)
- **gain_scale**: *string* The gain scale of the calibration, which indicates the units of the calibrated visibilities. For example, Jy or K str. (*gain_scale*)
- **pol_convention**: *string* The convention for how instrumental polarizations (e.g. XX and YY) are converted to Stokes parameters. Options are “sum” and “avg”, corresponding to I=XX+YY and I=(XX+YY)/2 (for linear instrumental polarizations) respectively. This header item is not required, but is highly recommended. If pol_convention is present, gain_scale should also be present. (*pol_convention*)
- **freq_array**: *float* An array of all the frequencies (centers of the channel, for all spectral windows) stored in the file in Hertz. This is a one-dimensional array of size (Nfreqs,). Required for per-frequency calibration solutions, should not be present for wide band calibration solutions. (*freq_array*)
- **channel_width**: *float* The width of frequency channels in the file in Hertz. This is a one-dimensional array of size (Nfreqs,). Required for per-frequency calibration solutions, should not be present for wide band calibration solutions. (*channel_width*)
- **flex_spw_id_array**: *int* The mapping of individual channels along the frequency axis to individual spectral windows, as listed in the *spw_array.* This is a one-dimensional array of size (Nfreqs,). Required for per-frequency calibration solutions, should not be present for wide band calibration solutions. (*flex_spw_id_array*)
- **freq_range**: *float* Frequency range that the calibration solutions are valid for. This should be an array with shape (Nspws, 2) where the second axis gives the start frequency and end frequency (in that order) in Hertz. Required for wide band calibration solutions, should not be present for per-frequency calibration solutions. (*freq_range*)
- **flex_jones_array**: *int* Optional array that allows for labeling individual spectral windows with different polarizations. This is a one-dimensional array of size Nspws. If present, Njones must be set to 1 (i.e., only one Jones vector per spectral window is allowed). (*flex_jones_array*)
- **time_array**: *float* An array of the Julian Date corresponding to the temporal midpoint of the calibration solution. This is a one-dimensional array of size Ntimes. Should be present for calibration solutions calculated per visibility integration. Only one of time_range and time_array should be present. (*time_array*)

- **lst_array**: *float* An array corresponding to the local sidereal time of the temporal midpoint of each solution in units of radians. If it is not specified, it is calculated from the latitude/longitude and the time_array. Saving it in the file can be useful for files with many values in the time_array, which would expensive to recompute. This is a one-dimensional array of size Ntimes. Should only be present for calibration solutions calculated per visibility integration. Only one of lst_range and lst_array should be present. (*time_array*)

- **time_range**: *float* Time range in Julian Date that calibration solutions are valid for. This should be an array with shape (Ntimes, 2) where the second axis gives the start time and end time (in that order) in JD. Should be present if the calibration solutions apply over a range of times. Only one of time_range and time_array should be present. (*time_range*)

- **lst_range**: *float* Local sidereal time range in radians corresponding to the time_range. This should be an array with shape (Ntimes, 2) where the second axis gives the start LST and end LST (in that order). Should only be present if the calibration solutions apply over a range of times. Only one of lst_range and lst_array should be present. (*time_range*)

- **ref_antenna_name**: *string* Phase reference antenna name. If there are different reference antennas for different times, this will be "various" and the ref_antenna_array will be present. Required for sky based calibrations. (*ref_antenna_name*)

- **ref_antenna_array**: *int* Reference antenna number array, only used for sky-based calibration solutions if the reference antenna varies by time. This is a one-dimensional array of size Ntimes. (*ref_antenna_array*)

- **sky_catalog**: *string* Name of the sky catalog used in calibration, Required for sky based calibration solutions. (*sky_catalog*)

- **diffuse_model**: *string* The name of the diffuse model used in the calibration, only used for sky based calibration solutions. (*diffuse_model*)

- **Nsources**: *int* The number of sources used in the calibration, only used for sky based calibration solutions. (*Nsources*)

- **baseline_range**: *float* Range of baseline lengths used for calibration. This is a array of length 2 giving the shortest and longest baselines used in calculating the calibrations solutions. Only used for sky based calibration solutions. (*baseline_range*)

- **Nphase**: *int* The number of phase centers present in the phase_center_catalog. (*Nphase*)

- **phase_center_catalog**: A way to specify where the data where phased to when the calibration solutions were calculated (most commonly seen with calibration solutions derived from measurement sets). This is nearly identical to the dataset with the same name in UVH5 files. A series of nested datasets, similar to a dict in python (*phase_center_catalog*). The top level keys are integers giving the phase center catalog IDs which are used to identify which times are phased to which phase center via the *phase_center_id_array*. The next level keys must include:

  - **cat_name**: *string* The phase center catalog name. This does not have to be unique, non-unique values can be used to indicate sets of phase centers that make up a mosaic observation.
  - **cat_type**: *string* One of four allowed values: **(1)** sidereal, **(2)** ephem, **(3)** driftscan, **(4)** unprojected. Sidereal means a phase center that is fixed in RA and Dec in a given celestial frame. Ephem means a phase center that has an RA and Dec that moves with time. Driftscan means a phase center with a fixed azimuth and elevation (note that this includes w-projection, even at zenith). Unprojected means no phasing, including w-projection, has been applied.
  - **cat_lon**: *float* The longitudinal coordinate of the phase center, either a single value or a one dimensional array of length Npts (the number of ephemeris data points) for ephem type phase centers. This is commonly RA, but can also be galactic longitude. It is azimuth for driftscan phase centers.
  - **cat_lat**: *float* The latitudinal coordinate of the phase center, either a single value or a one dimensional array of length Npts (the number of ephemeris data points) for ephem type phase centers. This is commonly Dec, but can also be galactic latitude. It is elevation (altitude) for driftscan phase centers.
  - **cat_frame**: *string* The coordinate frame that the phase center coordinates are defined in. It must be an astropy supported frame (e.g. fk4, fk5, icrs, gcrs, cirs, galactic).

  And may include:

  - **cat_epoch**: *float* The epoch in years for the phase center coordinate. For most frames this is the Julian epoch (e.g. 2000.0 for j2000) but for the FK4 frame this will be treated as the Bessel-Newcomb epoch (e.g. 1950.0 for B1950). This parameter is not used for frames without an epoch (e.g. ICRS) unless the there is proper motion (specified in the cat_pm_ra and cat_pm_dec keys).
  - **cat_times**: *float* Time in Julian Date for ephemeris points, a one dimensional array of length Npts (the number of ephemeris data points). Only used for ephem type phase centers.
  - **cat_pm_ra**: *float* (sidereal only) Proper motion in RA in milliarcseconds per year for the source.

  - **cat_pm_dec**: *float* (sidereal only) Proper motion in Dec in milliarcseconds per year for the source
  - **cat_dist**: *float* Distance to the source in parsec (useful if parallax is important), either a single value or a one dimensional array of length Npts (the number of ephemeris data points) for ephem type phase centers.
  - **cat_vrad**: *float* Radial velocity of the source in km/sec, either a single value or a one dimensional array of length Npts (the number of ephemeris data points) for ephem type phase centers.
  - **info_source**: *string* Information about provenance of the source details. Typically this is set either to "file" if it originates from a file read operation, and "user" if it was added because of a call to the `phase()` method in `pyuvdata`. But it can also be set to contain more detailed information.

  (*phase_center_catalog*)

- **phase_center_id_array**: *int* A one dimensional array of length Ntimes containing the cat_id from the phase_center_catalog that the data were phased to for each calibration time.
  (*phase_center_id_array*)

- **observer**: *string* Name of observer who calculated solutions in this file. (*observer*)

- **git_origin_cal**: *string* Origin (e.g. on github) of calibration software. Url and branch. (*git_origin_cal*)

- **git_hash_cal**: *string* Commit hash of calibration software (from git_origin_cal) used to generate solutions. (*git_hash_cal*)

- **scan_number_array**: *int* Measurement set scan numbers. This is a one-dimensional array of size Ntimes. May be present if the calibration solutions derive from measurement sets. (*scan_number_array*)

3.3 Extra Keywords

UVData objects support "extra keywords", which are additional bits of arbitrary metadata useful to carry around with the data but which are not formally supported as a reserved keyword in the `Header`. In a UVH5 file, extra keywords are handled by creating a datagroup called `extra_keywords` inside the `Header` datagroup. In a UVData object, extra keywords are expected to be scalars, but UVH5 makes no formal restriction on this. Also, when possible, these quantities should be HDF5 datatypes, to support interoperability between UVH5 readers. Inside of the extra_keywords datagroup, each extra keyword is saved as a key-value pair using a dataset, where the name of the extra keyword is the name of the dataset and its corresponding value is saved in the dataset. Though the use of HDF5 attributes can also be used to save additional metadata, it is not recommended, due to the lack of support inside of pyuvdata for ensuring the attributes are properly saved when writing out.

4 Data

In addition to the `Header` datagroup in the root namespace, there must be one called `Data`. This datagroup saves the gain or delay calibration solutions, flags, and optionally, quality measure arrays. Either a delay or gain datasets and a flag dataset must be present in a valid CalH5 file. Per-frequency calibration solutions have arrays of shape: (Nants_data, Nfreqs, Ntimes, Njones) while wide band solutions have arrays of shape: (Nants_data, Nspws, Ntimes, Njones) (see the wide_band header item for more details). There can also be a total quality dataset that provides a quality across the entire telescope (so drops the Nants_data axis).

4.1 Gain Dataset

Gain data is saved as a dataset named `gains`, which must be present in the cal_type header item is "gain". It should be a 4-dimensional, complex-type dataset with shape (Nants_data, Nfreqs, Nfreqs, Npols) for a per-frequency solution (i.e. the wide_band header item is False) or shape (Nants_data, Nspws, Ntimes, Njones) for a wide band solution (i.e. the wide_band header item is True). Commonly this is saved as an 8-byte complex number (a 4-byte float for the real and imaginary parts), though some flexibility is possible and 16-byte complex floating point numbers (composed of two 8-byte floats) are also common. In all cases, a compound datatype is defined, with an `'r'` field and an `'i'` field, corresponding to the real and imaginary parts, respectively. The real and imaginary types must also be the same datatype. For instance, they should both be 8-byte floating point numbers. Mixing datatypes between the real and imaginary parts is not allowed.

Using `h5py`, the datatype for `gains` can be specified as `'c8'` (8-byte complex numbers, corresponding to the `np.complex64` datatype) or `'c16'` (16-byte complex numbers, corresponding to the `np.complex128` datatype) out-of-the-box, with no special handling by the user. `h5py` transparently handles the definition of the compound datatype.

4.2 Delay Dataset

Delay data is saved as a dataset named `delays`, which must be present in the cal_type header item is "delay". It should be a 4-dimensional, float-type dataset with shape (Nants_data, Nspws, Ntimes, Njones).

4.3 Flags Dataset

The flags corresponding to the calibration solutions are saved as a dataset named `flags`. It is a 4-dimensional, boolean-type dataset with shape (Nants_data, Nfreqs, Nfreqs, Npols) for a per-frequency solution (i.e. the wide_band header item is False) or shape (Nants_data, Nspws, Ntimes, Njones) for a wide band solution (i.e. the wide_band header item is True). Values of True correspond to instances of flagged data, and False is non-flagged. Note that the boolean type of the data is *not* the HDF5-provided `H5T_NATIVE_HBOOL`, and instead is defined to conform to the `h5py` implementation of the numpy boolean type. When creating this dataset from `h5py`, one can specify the datatype as `np.bool`. Behind the scenes, this defines an HDF5 enum datatype. See the UVH5 memo, Appendix C for an example of how to write a compatible dataset from C.

Compression is typically applied to the flags dataset. The LZF filter (included in all HDF5 libraries) provides a good compromise between speed and compression, and is the default for CalH5 files written with pyuvdata. Note that HDF5 supports many other types of filters, such as ZLIB, SZIP, and BZIP2.[6] In the special cases of single-valued arrays, the dataset occupies virtually no disk space.

4.4 Quality Dataset

The quality measure corresponding to the calibration solutions can optionally be saved as a dataset named `qualities`. It is a 4-dimensional, float-type dataset with shape (Nants_data, Nfreqs, Nfreqs, Npols) for a per-frequency solution (i.e. the wide_band header item is False) or shape (Nants_data, Nspws, Ntimes, Njones) for a wide band solution (i.e. the wide_band header item is True). The definition of the calibration quality measure depends on the calibration software, but χ^2 values are a common choice.

As with the flags dataset described above, it is common to apply compression to the qualities dataset.

4.5 Total Quality Dataset

A telescope array-wide quality measure corresponding to the calibration solutions can optionally be saved as a dataset named `total_qualities`. It is a 3-dimensional, float-type dataset with shape (Nfreqs, Nfreqs, Npols) for a per-frequency solution (i.e. the wide_band header item is False) or shape (Nspws, Ntimes, Njones) for a wide band solution (i.e. the wide_band header item is True). The definition of the calibration total quality measure depends on the calibration software, but array-averaged χ^2 values are a common choice.

As with the flags dataset described above, it is common to apply compression to the total_qualities dataset.

[6]For more information, see the documentation on using compression filters in HDF5.