Memo: UVH5 file format

Paul La Plante, and the pyuvdata team

November 28, 2018
Revised April 2, 2021
Revised July 14, 2022
Revised April 17, 2024

Contents

1 Introduction

This memo introduces a new HDF5[1]-based file format of a UVData object in `pyuvdata`[2], a python package that provides an interface to interferometric data. Here, we describe the required and optional elements and the structure of this file format, called *UVH5*.

Note that this file format is specifically designed to represent UVData objects. Other HDF5-based datasets for radio interferometers, such as katdal[3] or HDFITS[4] *are not compatible* with the standard as defined here. We refer the reader to the documentation of those other formats to find out more about them.

We assume that the user has a working knowledge of HDF5 and the associated python bindings in the package `h5py`[5], as well as UVData 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 UVData object, please visit https://pyuvdata.readthedocs.io/en/latest/uvdata.html. An example for how to interact with UVData objects in `pyuvdata` is available at http://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 UVH5 object contains the interferometric data from a radio telescope, as well as the associated metadata necessary to interpret it. A UVH5 file contains two primary HDF5

[1] https://www.hdfgroup.org/
[2] https://github.com/RadioAstronomySoftwareGroup/pyuvdata
[3] https://github.com/ska-sa/katdal
[4] https://github.com/telegraphic/fits2hdf
[5] https://www.h5py.org/

groups: the `Header` group, which contains the metadata, and the `Data` group, which contains the data itself, the flags, and information about the number of samples corresponding to the data. Datasets in the `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 UVH5 format, again because the HDF5 file is self-documenting in this regard. However, be warned that most UVH5 files "in the wild" typically feature compression of datasets in the `Data` group.

In the discussion below, we discuss required and optional datasets in the various groups. We note in parenthesis the corresponding attribute of a UVData 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 UVData 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 UVData attribute.

Note that string datatypes should be handled with care. See Appendix A for appropriately defining them for interoperability between different HDF5 implementations.

3.1 Required Parameters

- **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*)

- **instrument**: *string* The name of the instrument, typically the telescope name. (*instrument*)
- **history**: *string* The history of the data file. (*history*)
- **Nants_data**: *int* The number of antennas that have visibility data in the file. May be smaller than the number of antennas in the array. (*Nants_data*)
- **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*)
- **ant_1_array**: *int* An array of the first antenna numbers corresponding to baselines present in the data. All entries in this array must exist in the antenna_numbers array. This is a one-dimensional array of size Nblts. (*ant_1_array*)
- **ant_2_array**: *int* An array of the second antenna numbers corresponding to baselines present in the data. All entries in this array must exist in the antenna_numbers array. This is a one-dimensional array of size Nblts. (*ant_2_array*)
- **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 unique antenna in ant_1_array and ant_2_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 unique antenna in ant_1_array and ant_2_array, but there may be additional entries. (*antenna_names*)
- **Nbls**: *int* the number of baselines present in the data. For full cross-correlation data (including auto-correlations), this should be Nants_data×(Nants_data+1)/2. (*Nbls*)
- **Nblts**: *int* The number of baseline-times (i.e., the number of spectra) present in the data. Note that this value need not be equal to Nbls × Ntimes. (*Nblts*)
- **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. (*Nfreqs*)
- **Npols**: *int* The number of polarization products in the data. (*Npols*)
- **Ntimes**: *int* The number of time samples present in the data. (*Ntimes*)

- **uvw_array**: *float* An array of the uvw-coordinates corresponding to each observation in the data. Baselines are specified as from `ant_1_array` to `ant_2_array`, implying the position of antenna 2 minus the position of antenna 1. This is a two-dimensional array of size (Nblts, 3). Units are in meters. (*uvw_array*)

- **time_array**: *float* An array of the Julian Date corresponding to the temporal midpoint of the corresponding baseline's integration. This is a one-dimensional array of size Nblts. (*time_array*)

- **integration_time**: *float* An array of the duration in seconds of an integration. This is a one-dimensional array of size Nblts. (*integration_time*)

- **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). (*freq_array*)

- **channel_width**: *float* The width of frequency channels in the file in Hertz. This is a one-dimensional array of size (Nfreqs). (*channel_width*)

- **spw_array**: *int* An array of the spectral windows in the file. This is a one-dimensional array of size Nspws. (*spw_array*)

- **flex_spw**: *python bool*[6] Whether the data are saved using flexible spectral windows. If more than one spectral window is present in the data, this must be `True`. See Sec. 5.2.2 for a discussion of the details. (*flex_spw*)

- **polarization_array**: *int* An array of the polarizations contained in the file. This is a one-dimensional array of size Npols. Note that the polarizations should be stored as an integer, and use the convention defined in AIPS Memo 117. (*polarization_array*)

- **antenna_positions**: *float* An array of the antenna coordinates relative to the reference position of the radio telescope array, which is implicitly defined by the *latitude*, *longitude*, and *altitude* (LLA) parameters. More explicitly, these are the ECEF coordinates of individual antennas minus the ECEF coordinates of the reference telescope position, such that the telescope position plus the values stored in *antenna_positions* equals the position of individual elements in ECEF. The conversion between LLA and ECEF is given by WGS84. This is a two-dimensional array of size (Nants_telescope, 3). (*antenna_positions*)

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

- **phase_center_catalog**: 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

[6] See Appendix C

IDs which are used to identify which baseline-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 Nblts containing the cat_id from the phase_center_catalog that each baseline-time is phased to. (*phase_center_id_array*)
- **phase_center_app_ra**: *float* Apparent right ascension of the phase center in the topocentric frame of the observatory, in radians. This is a one-dimensional array of size Nblts. In the event that there are multiple phase centers, the phase_center_id_array can be used to identify which phase center is used for this calculation. For unprojected phase types, this is just the apparent LST (LAST). (*phase_center_app_ra*)
- **phase_center_app_dec**: *float* Apparent declination of the phase center in the topocentric frame of the observatory, in radians. This is a one-dimensional array of size Nblts. In the event that there are multiple phase centers, the phase_center_id_array can be used to identify which phase center is used for this calculation. For unprojected phase types, this is just the telescope latitude. (*phase_center_app_ra*)
- **phase_center_frame_pa**: *float* Position angle between the hour circle (which is a great circle that goes through the target postion and both poles) in the apparent/topocentric frame, and the frame given in the *phase_center_catalog* under the *cat_frame* dataset. This is a one dimensional array of length Nblts. In the event that there are multiple phase centers with different frames, the phase_center_id_array can be used to identify which frame is used for each baseline-time in this calculation. This is set to zero for unprojected phase types. (*phase_center_frame_pa*)
- **version**: *string* The version of the HDF5 file. The latest version (and the one described in this memo) is Version 1.1. Note it should be a string, such as `"1.1"`. See Sec. 5 for the version history of the HDF5 specification. (No corresponding UVData attribute)

3.2 Optional Parameters

- **vis_units**: *string* The units of the visibilities. Supported options are "Jy", "K str" or "uncalib" for uncalibrated data. Note that some older files may have "UNCALIB"

which `pyuvdata` supports for backwards compatibility, but all future files should use the lower case string. Not required but encouraged, assumed to be "uncalib" if not specified. (*vis_units*)

- **pol_ convention**: *string* The convention for how instrumental polarizations (e.g. XX and YY) are converted to Stokes parameters. Supported options are "sum" and "avg", corresponding to $I = XX + YY$ and $I = \frac{XX+YY}{2}$ (for linear instrumental polarizations) respectively. This only makes sense for calibrated data, so should only be present if vis_units is present and is not "uncalib". (*pol_ convention*)

- **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). Note this is **required** if the file uses flexible spectral windows (see Sec. 5.2.2). (*flex_spw_id_array*)

- **flex_spw_polarization_array**: *int* Allows for labeling individual spectral windows with different polarizations. If set, Npols must be 1 (i.e., only one polarization per spectral window allowed). This is a one-dimensional array of size (Nspws,). (*flex_spw_polarization_array*)

- **lst_array**: *float* An array corresponding to the local sidereal time of the center of each observation in the data 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. (*lst_array*)

- **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*)

- **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. For instance, "East" or "North" may be used. (*x_orientation*).

- **antenna_diameters**: *float* An array of the diameters of the antennas in meters. This is a one-dimensional array of size (Nants_telescope). (*antenna_diameters*)

- **dut1**: *float* difference between UT1 (defined with respect to the Earth's angle of rotation, which includes whole and partial "leap seconds") and UTC (which *only* includes whole leap seconds), in seconds, with typical precision of 1 ms. AIPS 117 calls it `UT1UTC`. Note that this is slightly different from the value DUT1 which is broadcast by various time signal services (e.g., NIST), which only supply this difference with precision of 0.1 seconds. (*dut1*)

- **earth_omega**: *float* Earth's rotation rate in degrees per day. Note the difference in units, which is inherited from the way this quantity is handled in UVFITS datasets (AIPS 117 calls it `DEGPDY`). (*earth_omega*)
- **gst0**: *float* Greenwich sidereal time at midnight on reference date, in degrees. AIPS 117 calls it `GSTIAO` (*gst0*)
- **rdate**: *string* Date for which GST0 (or whichever time saved in that field) applies. Note this is different from how UVFITS handles this quantity, which is saved as a float rather than a string. The user is encouraged to ensure it is being handled self-consistently for their desired application. (*rdate*)
- **timesys**: *string* Time system. pyuvdata currently only supports UTC. (*timesys*)
- **blts_are_rectangular**: *python bool*[7] Indicates whether the baseline-time axis is rectangular (i.e. each baseline is present for each time). This can be determined from the other metadata if it is not provided, but that can take time, so providing it can provide code efficiencies. (*blts_are_rectangular*)
- **time_axis_faster_than_bls**: *python bool*[8] If the baseline-time axis is rectangular, this indicates whether the time axis is the fastest-moving virtual axis. Should only be present if blts_are_rectangular is present and is True. This can be determined from the other metadata if it is not provided, but that can take time, so providing it can provide code efficiencies. (*time_axis_faster_than_bls*)
- **blt_order**: *string* Indicates the ordering of the data along the baseline-time axis. This can either be a single string two comma delimited strings giving the first and optionally second ordering criteria. Supported strings are: "time", "baseline", "ant1", "ant2", "bda". For example, data that is ordered first by time then by the first antenna number (so times are in order and change slowest and within each time, the first antenna numbers are in order and change next fastest) would be recorded here as "time, ant1". The "bda" option is for data that has been averaged to different integration times depending on baseline length and orientation and should only ever appear as a single string (in this case, the axis is ordered first by integration time, then by baseline number and then by time). Not required, but can allow for code efficiencies if known. (*blt_order*)
- **eq_coeffs**: *float* An array per-antenna and per-frequency equalization coefficients. This is a two-dimensional array of size (Nants_telescope, Nfreqs). (*eq_coeffs*)
- **eq_coeffs_convention**: *string* The convention for how to remove eq_coeffs from data. Supported options are "divide" and "multiply". (*eq_coeffs_convention*)

[7] See Appendix C
[8] See Appendix C

- **uvplane_reference_time**: *int* The time at which the phase center is normal to the chosen UV plane for phasing. Used for interoperability with the FHD package[9]. (*uvplane_reference_time*)

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 visibility data, flags, and number of samples corresponding to each entry. All three datasets must be present in a valid UVH5 file. They are also all expected to be the same shape: (Nblts, Nfreqs, Npols). Note that due to the intermixing of the baseline and time axes, it is *not* required for data to exist for every baseline and time in the file. This behavior is similar to UVFITS and MIRIAD file formats. Also note that there is no explicit ordering required for the baseline-time axis. A common ordering is to write the data in "correlator order", and have all baselines for a single time t_i, followed by all baselines for the next time t_{i+1}, etc. However, this is merely a convention, and is not explicitly required for the UVH5 format.

4.1 Visdata Dataset

The visibility data is saved as a dataset named `visdata`. It should be a 3-dimensional, complex-type dataset with shape (Nblts, Nfreqs, Npols). Most 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. 16-byte complex floating point numbers (composed of two 8-byte floats), as well as 8-byte complex integers (two 4-byte signed integers), are also common. In all cases, a compound datatype is defined, with an `'r'` field and an `'i'` field,

[9] https://github.com/EoRImaging/FHD

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, or 32-bit (4-byte) integers. Mixing datatypes between the real and imaginary parts is not allowed.

Using `h5py`, the datatype for `visdata` 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. For examples of how to handle complex integer datatypes in `h5py`, see Appendix B.

4.1.1 Conjugation Convention

A cross-correlation between two antennas is defined by the baseline connecting them, and the conjugation of one of the input data streams. Accordingly, the *uvw* coordinates and the conjugation of the visibility data are interconnected, based on the definition of one's coordinate system. For UVH5 files, it is assumed that the convention for the Radio Interferometer Measurement Equation (RIME) of a visibility $\mathcal{V}$ for antennas i and j is as follows [1]:

$$\mathcal{V}(u_j - u_i, v_j - v_i) = \int \mathrm{d}l\, \mathrm{d}m\, I(l,m) g_i(l,m) e^{-2\pi i(u_i l + v_i m)} g_j^*(l,m) e^{2\pi i(u_j l + v_j m)}. \quad (1)$$

That is, the baseline vector defined by the *uvw* coordinates is directed from antenna i to antenna j (so the baseline vector can be computed as $\mathbf{r_j} - \mathbf{r_i}$, where $\mathbf{r}$ is the position vector of a given antennas), and the data corresponding to antenna j is conjugated. Following the specification of the baselines, antenna i is given by `ant_1_array` and j by `ant_2_array`. Note that if a file is generated with the opposite convention, it is usually sufficient to multiply *uvw* coordinates by -1 to generate a self-consistent dataset, as well as conjugate the data in the `data_array`.

4.2 Flags Dataset

The flags corresponding to the data are saved as a dataset named `flags`. It is a 3-dimensional, boolean-type dataset with shape (Nblts, Nfreqs, Npols). 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 Appendix C for an example of how to write a compatible dataset from C.

As with the nsamples dataset discussed below, 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 used in most HERA datasets. Note that HDF5 supports many other types of filters, such as ZLIB, SZIP, and BZIP2.[10] In the special cases of single-valued arrays, the dataset occupies virtually no disk space.

4.3 Nsamples Dataset

The number of data points averaged into each data entry is saved as a dataset named `nsamples`. It is a 3-dimensional, floating-point type dataset with shape (Nblts, Nfreqs, Npols). Note that it is *not* required to be an integer, and should *not* be saved as an integer type. The product of the integration_time array and the data in the nsample array reflects the total amount of time that went into a visibility. The best practice is for the nsamples dataset to track flagging within an integration time (leading to a decrease of the nsamples array value to be less than 1) and LST averaging (leading to an increase in the nsamples array value). Datasets that have not been LST averaged should have values in nsamples that are less than or equal to 1. Although this convention is not adhered to by all data formats serviced by `pyuvdata`, it is recommended to follow it as closely as possible in UVH5 files. What *should* be true is the product of the integration_time array and nsamples array corresponding to the total amount of time included in a visibility.

5 Version History

The UVH5 specification has been through several minor version updates, and in the interest of maximizing interoperability between different readers and writers external to `pyuvdata`, it is useful to define a version history. This is not a strict semantic versioning scheme, but instead intended to capture some of the important changes that the specification has gone through. Note that, as much as possible, `pyuvdata` intends to be fully compatible, and be able to read any valid UVH5 file written. Those interested in writing fully compatible readers/writers may look there for further details.

It is strongly encouraged that independent UVH5 writers conform to the latest version (Version 1.1 at time of writing), while readers are encouraged to support backwards compatibility as much as possible. If readers cannot support all revisions, reading more recent versions should be prioritized.

5.1 `version` dataset

When present, the version information is stored in the Header as a string-based dataset with the key `version`. Note that files have not always contained this dataset, but as much as possible, new files written should contain this dataset to clarify.

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

5.2 Version 0.x/0.1

Historically, UVH5 files written by `pyuvdata` and the HERA correlator did not include the `version` dataset as part of the header. Implicitly, these files are v0.x. More recently, `pyuvdata` has begun writing the version information to files, and so the `version` dataset is present in these files. Below, we discuss some of the changes that occurred within the Version 0.1 generation, to make users aware of the different flavors of UVH5 files they may encounter "in the wild."

5.2.1 `integration_time` dataset

Initially, UVH5 files were written with a single value for `integration_time`. It has since been modified to its current length of `Nblts` to allow for data with varying integration time between time samples or baselines.

5.2.2 Flexible Spectral Windows

A significant update to how the frequency axis was handled in UVData objects was implemented to allow for a more flexible handling of data from different spectral windows. Initially, following the method of handling multiple spectral windows in UVFITS files, the spectral window (`spw`) axis was treated as a separate axis in metadata and data arrays. However, this approach is relatively inflexible, because it requires all spectral windows to have the same number of frequency channels to efficiently store the data (the alternatives being to use ragged-length arrays, which are inefficient for storing or accessing the data, or padded arrays which can contain a large amount of wasted storage to ensure arrays are regularly spaced).

To overcome these limitations, taking inspiration from how frequency data are stored in MIRIAD, the idea of "flexible spectral windows" was adopted to save the frequency information. Analogously to how baselines and times are collapsed to a "baseline-time axis", frequencies and spectral windows are collapsed to a "frequency-spectral window" axis. This allows for more versatility in how data from different spectral windows are stored inside of a single file, but it requires the change of several important components of metadata. We summarize these changes here.

- The value for `Nfreqs` is the total number of frequency channels saved in the data across all spectral windows.
- Where required, the number of spectral windows `Nspws` is required to be 1.
- The `channel_width` dataset was changed from a single number to a 1-d array of length `Nfreqs`.
- The `flex_spw` dataset was added to identify whether the file in question supports flexible spectral windows (if `True`) or not (if `False`).

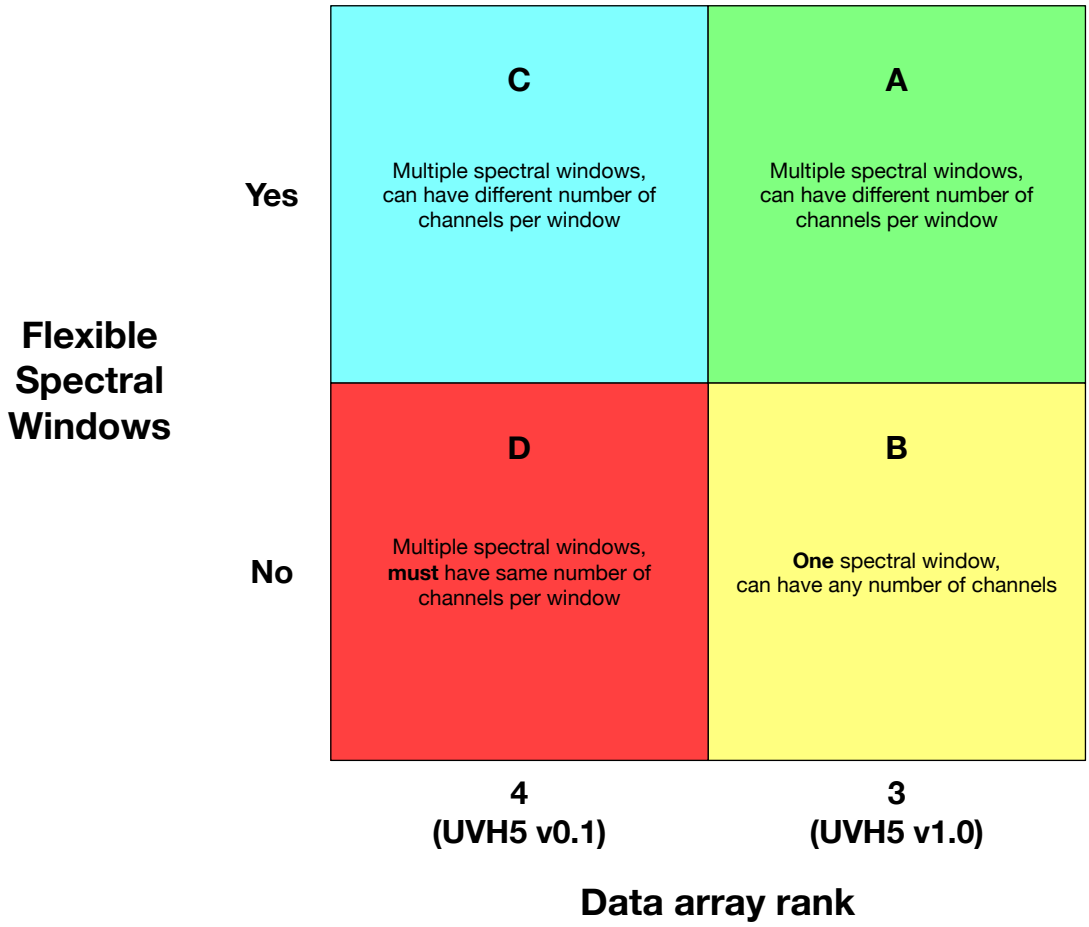

Figure 1: A summary of the different combinations of the rank of data arrays (reflected by UVH5 version), and flexible spectral windows. The various data and metadata values and ranks are listed in detail in Table 2.

- The `flex_spw_id_array` dataset was added to identify which spectral window a given channel belongs. This is required if `flex_spw` is `True`.

It is possible to save files self-consistently without using flexible spectral windows *if and only if there is a single spectral window*. We outline the various (valid) combinations below in Sec. 5.5.

5.3 Version 1.0

5.3.1 Rank-3 Array Convention

Version 1.0 of UVH5 represents a significant change in the way that the data arrays (`visdata`, `flags`, and `nsamples`) and metadata arrays are stored. The previously vestigial spectral-window axis is removed, meaning that data arrays are rank-3 instead of rank-4. Explicitly, these arrays have shape (Nblts, Nfreqs, Npols), where Nfreqs includes the number of channels across all spectral windows. This also affects the *freq_array* dataset, which went from a rank-2 array to rank-1 of size (Nfreqs). The description of data and metadata in the body of this memo assumes the Version 1.0 specification. Although `pyuvdata` plans to indefinitely support files written with the previous convention (i.e., having an explicit spectral window-axis), UVH5 files should be written such that they conform to Version 1.0.

5.4 Version 1.1

Historically, only a single phase center was supported and only sidereal or unprojected (zenith drift without w projection) phasing types were supported. When multiple phase center phasing was added, along with support for more types of phase centers, the following parameters were added (described in 3.1):

- **phase_center_catalog**
- **phase_center_id_array**
- **phase_center_app_ra**
- **phase_center_app_dec**
- **phase_center_frame_pa**

and the following header items (found in versions less than 1.1) were removed:

- **phase_center_ra**: *float* The right ascension of the phase center of the observation in radians. Required if phase_type is "phased". (*phase_center_ra*)
- **phase_center_dec**: *float* The declination of the phase center of the observation in radians. Required if phase_type is "phased". (*phase_center_dec*).
- **phase_center_epoch**: *float* The epoch year of the phase applied to the data (*e.g.*, 2000.). Required if phase_type is "phased". (*phase_center_epoch*)
- **phase_center_frame**: *string* The frame the data and uvw_array are phased to. Options are "gcrs" and "icrs", with default "icrs". These frames are defined as coordinate systems in astropy. (*phase_center_frame*)

Dataset	Current Convention	Previous Convention	Version Changed
`Header/version`	String corresponding to version	Not present	v0.1
`Header/integration_time`	Array of float, shape (Nblts)	Single float (assumed to apply to all baseline-times)	v0.1
`Header/phase_center_catalog`	nested datasets, similar to a dict in python	Not present	v1.1
`Header/phase_center_id_array`	Array of int, shape (Nblts)	Not present	v1.1
`Header/phase_center_app_ra`	Array of float, shape (Nblts)	Not present	v1.1
`Header/phase_center_app_dec`	Array of float, shape (Nblts)	Not present	v1.1
`Header/phase_center_app_pa`	Array of float, shape (Nblts)	Not present	v1.1

Table 1: A table summarizing changes that have occurred in the UVH5 specification.

- **object_name**: *string* The name of the object tracked by the telescope. For a drift-scan antenna, this is typically "zenith". (*object_name*)
- **phase_type**: *string* The phase type of the observation. Should be "phased" or "drift". Note that "drift" in this context more accurately means "unphased", in that baselines are computing using ENU coordinates, without any w-projection. Any other value is treated as an unrecognized type. (*phase_type*)

Prior to version 1.1, the new phase attributes were sometimes written to files along with the header items listed above. During this time, the **phase_center_catalog** was written as a python dict converted to a JSON-formatted string. This intermediate file format was undocumented and not widely used, but it is possible some files like this exist "in the wild".

5.5 Table Summarizing Changes

In the interest of summarizing all of the historical changes in a single place, we outline below the changes that have occurred in the UVH5 specification. We note what they are currently, along with how they were saved previously.

Dataset	Type A	Type B	Type C	Type D
`Header/Nspws`	Number of spectral windows	1	Number of spectral windows	Number of spectral windows
`Header/Nfreqs`	Number of frequencies across all spectral windows	Number of frequencies	Number of frequencies across all spectral windows	Number of frequencies *per* spectral window
`Header/channel_width`	Shape (Nfreqs)	Shape (Nfreqs)	Shape (Nfreqs)	Scalar (assumed to apply to all frequencies)
`Header/flex_spw_id_array`	Shape (Nfreqs)	Not present	Shape (Nfreqs)	Not present
`Header/flex_spw`	`True`	`False`	`True`	`False` **OR** not present
`Header/freq_array`	Shape (Nfreqs)	Shape (Nfreqs)	Shape (Nfreqs)	Shape (Nspws, Nfreqs)
`Data/visdata`	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, 1, Nfreqs, Npols)	Shape (Nblts, Nspws, Nfreqs, Npols)
`Data/flags`	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, 1, Nfreqs, Npols)	Shape (Nblts, Nspws, Nfreqs, Npols)
`Data/nsamples`	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, Nfreqs, Npols)	Shape (Nblts, 1, Nfreqs, Npols)	Shape (Nblts, Nspws, Nfreqs, Npols)

Table 2: A table summarizing the different data and metadata values for different file types. Type A, B, C, and D refer to the combinations of data array rank and flexible spectral windows in Figure 1. Note that UVH5 writers are strongly encouraged to write files compatible with Type A or B (i.e., UVH5 v1.0), whereas readers are encouraged to be as flexible as possible (within reason).

We also summarize the combination of data and metadata properties for the cases of: (A) rank-3 data arrays, flexible spectral windows; (B) rank-3 data arrays, no flexible spectral windows; (C) rank-4 data arrays, flexible spectral windows; (D) rank-4 data arrays, no flexible spectral windows. See Figure 1 for a visual representation. **Note that we include the following only as a reference! We encourage UVH5 writers to conform as much as possible to the v1.0 specification (options A or B).**

References

[1] A. Richard Thompson, James M. Moran, and George W. Swenson, Jr., "Interferometry and Synthesis in Radio Astronomy, 3rd Edition", 2017.

Appendix A Strings in HDF5

String datatypes are finicky, and require special handling to ensure that they are compatible with the HDF5 bindings in various languages. This is especially true for files written from `h5py`, which handles strings differently between python2 and python3. Though python2 is nearing its end-of-life, UVH5 should be backwards compatible with older versions of `h5py` as much as possible. To help service this, all string-type metadata in UVH5 files *must* be fixed-length ASCII type. Not only does this allow for interoperability between different `h5py` versions, but it also ensures that strings can be round-tripped through other HDF5 bindings, such as those in C, MATLAB, IDL, Fortran[11], etc. Note that the string should use one byte per character, and be null-terminated. This corresponds to the numpy `S` datatype in both versions of python2 and python3.

When writing a string-like dataset from `h5py`, scalar data should be written by casting a string to a `numpy.string_` object. Array data should be written as a `S<n>` dataset, where `<n>` represents the length of the strings to be saved. Upon reading, strings can be cast to bytes using the `tostring()` method, at which point the data is `<str>`-type (python2) or can be decoded as UTF-8 to become `<str>`-type (python3).

Below is an example for how to read and write string scalar and array-type datasets using `h5py` in python2 and python3.

A.1 Target String Type

The following is the output of `h5dump` for a string-like dataset in a UVH5 file. UVH5 writers are strongly encouraged (though not required) to follow the same convention. Although something like UTF-8 is more flexible, restricting strings to ASCII allows for greater interoperability with other file formats such as MIRIAD and UVFITS.

```
$ h5dump -V
h5dump: Version 1.12.0
$ h5dump -d Header/history -A simulated_bda_file.uvh5
HDF5 "simulated_bda_file.uvh5" {
DATASET "Header/history" {
   DATATYPE  H5T_STRING {
      STRSIZE 1035;
      STRPAD H5T_STR_NULLPAD;
      CSET H5T_CSET_ASCII;
      CTYPE H5T_C_S1;
   }
   DATASPACE  SCALAR
}
```

[11] Strings in Fortran are not null-terminated, so these require special handling.

```
}
```

A.2 Writing strings in python2

```
import numpy as np
import h5py
# open file and write string datasets
with h5py.File('test_file.uvh5', 'w') as f:
    header = f.create_group('Header')
    # scalar dataset
    header['scalar_string'] = np.bytes_('Hello world!')

    # array dataset
    str_array = np.array(['hello', 'world'])
    n_words = len(str_array)
    max_len_words = np.amax([len(n) for n in str_array])
    dtype = "S{:d}".format(max_len_words)
    header.create_dataset('array_string', (n_words,), dtype=dtype,
                          data=str_array)

# read the data back in again
with h5py.File('test_file.uvh5', 'r') as f:
    header = f['Header']
    # read scalar dataset
    scalar_string = header['scalar_string'][()].tobytes()
    assert scalar_string == 'Hello world!'

    # read array dataset
    str_array_file = [n.tobytes() for n in header['array_string'][()]]
    assert np.all(str_array_file == str_array)
```

A.3 Writing strings in python3

```
import numpy as np
import h5py
# open file and write string datasets
with h5py.File('test_file.uvh5', 'w') as f:
    header = f.create_group('Header')
    # scalar dataset
    header['scalar_string'] = np.bytes_('Hello world!')
```

```
    # array dataset
    str_array = ['hello', 'world']
    header['array_string'] = np.bytes_(str_array)

# read the data back in again
with h5py.File('test_file.uvh5', 'r') as f:
    header = f['Header']
    # read scalar dataset
    scalar_string = header['scalar_string'][()].tobytes().decode('UTF-8')
    assert scalar_string == 'Hello world!'

    # read array dataset
    str_array_file = [n.tobytes().decode('UTF-8')
                      for n in header['array_string'][()]]
    assert np.all(str_array_file == str_array)
```

Appendix B Integer Datatype Support for Visibility Data

The HERA correlator writes datasets which have 32-bit integer real and imaginary components. Due to the self-describing nature of HDF5 datasets, this information is captured by the file format. Nevertheless, special handling must be used to interpret these datasets as complex numbers. The `astype` context manager in `h5py` is used to convert the datatype on the fly from integers to complex numbers. Below is an example of how to do this.

```
import numpy as np
import h5py
# define integer datatype
int_dtype = np.dtype([('r', '<i4'), ('i', '<i4')])

# open file and read in the dataset
with h5py.File('test_file.uvh5', 'r') as f:
    visdata = f['Data/visdata']
    dshape = visdata.shape
    data = np.empty(dshape, dtype=np.complex128)
    with visdata.astype(int_dtype):
        data.real = visdata['r'][:, :, :]
        data.imag = visdata['i'][:, :, :]
```

Appendix C Defining Python Boolean Types in C

Several header items and the flags array (Sec. 4.2) are booleans, which are *not* encoded as the `H5T_NATIVE_HBOOL` type; instead, they are an `H5Tenum` type, with an explicit `TRUE` and `FALSE` value. When creating such a datatype using `h5py`, the user simply needs to ensure the datatype `np.bool_`. The building of the enum is transparent. When building the enum from a different language, the precise specification is necessary to ensure compatibility. The following code is a template for how to build the appropriate datatype using C. The construction in other languages, such as Fortran, should follow analogously.

```
#include <hdf5.h>

#define CPTR(VAR,CONST) ((VAR)=(CONST),&(VAR))

typedef enum {
  FALSE,
  TRUE
} bool_t;

int main() {
  bool_t val;
  static hid_t boolenumtype;
  hid_t file_id, dspace_id, flags_id;
  herr_t status;

  /* define enum type */
  boolenumtype = H5Tcreate(H5T_ENUM, sizeof(bool_t));
  H5Tenum_insert(boolenumtype, "FALSE", CPTR(val, FALSE ));
  H5Tenum_insert(boolenumtype, "TRUE" , CPTR(val, TRUE  ));

  /* open a new file */
  file_id = H5Fcreate("test_file.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

  /* define array dimensions */
  int Nblts = 10;
  int Nfreqs = 16;
  int Npols = 4;
  hsize_t dims[3] = {Nblts, Nfreqs, Npols};

  /* initialize data array with FALSE values */
  bool_t data[Nblts][Nfreqs][Npols];
  for (int i=0; i<Nblts; i++) {
```

```
    for (int j=0; j<Nfreqs; j++) {
      for (int k=0; k<Npols; k++) {
        data[i][j][k] = FALSE;
      }
    }
  }

  /* make dataspace and write out data */
  dspace_id = H5Screate_simple(3, dims, dims);
  flags_id = H5Dcreate(file_id, "flags", boolenumtype, dspace_id,
                       H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
  status = H5Dwrite(flags_id, boolenumtype, H5S_ALL, H5S_ALL,
                    H5P_DEFAULT, data);

  /* close down */
  H5Dclose(flags_id);
  H5Sclose(dspace_id);
  H5Fclose(file_id);
  return 0;
}
```