Memo: UVH5 file format

Paul La Plante, and the pyuvdata team

November 28, 2018

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 http://pyuvdata.readthedocs.io/en/latest/uvdata_parameters.html. An example for how to interact with UVData objects in pyuvdata is available at http://pyuvdata.readthedocs.io/en/latest/tutorial.html.

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 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

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

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 disucssion 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 **??** 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*)

- **object_name**: *string* The name of the object tracked by the telescope. For a drift-scan antenna, this is typically "zenith". (*object_name*)
- **history**: *string* The history of the data file. (*history*)
- **phase_type**: *string* The phase type of the observation. Should be "phased" or "drift". Any other value is treated as an unrecognized type. (*phase_type*)
- **Nants_data**: *int* The number of antennas that data in the file corresponds to. 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*)
- **Nfreqs**: *int* The number of frequency channels in the data. (*Nfreqs*)
- **Npols**: *int* The number of polarization products in the data. (*Npols*)
- **Ntimes**: *int* The number of time samples present in the data. (*Ntimes*)
- **Nspws**: *int* The number of spectral windows present in the data. (*Nspws*)

- **uvw_array**: *float* An array of the uvw-coordinates corresponding to each observation in the data. This is a two-dimensional array of size (Nblts, 3). (*uvw_array*)
- **time_array**: *float* An array of the Julian Date corresponding to the center of an integration. This is a one-dimensional array of size Nblts. (*time_array*)
- **integration_time**: *float* An array of the length of time in seconds of an integration. This is a one-dimensional array of size Nblts. (*time_array*)
- **freq_array**: *float* An array of the frequencies stored in the file in Hertz. This is a two-dimensional array of size (Nspws, Nfreqs). (*freq_array*)
- **channel_width**: *float* The width of frequency channels in the file in Hertz. (*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*)
- **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 *telescope_location* (in the ITRF frame). This is a two-dimensional array of size (Nants_telescope, 3). (*antenna_positions*)

3.2 Optional Parameters

- **dut1**: *float* DUT1 (google it). AIPS 117 calls it "UT1UTC". (*dut1*)
- **earth_omega**: *float* Earth's rotation rate in degrees per day. (*earth_omega*)
- **gst0**: *float* Greenwich sidereal time at midnight on reference date. (*gst0*)
- **rdate**: *string* Date for which GST0 (or whichever time saved in that field) applies. (*rdate*)
- **timesys**: *string* Time system. pyuvdata currently only supports UTC. (*timesys*)
- **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, "E" or "East" 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). (*Nants_telescope*)

- **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[6].
- **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". (*phase_center_frame*)
- **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. (*lst_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. 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, Nspws, 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

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

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 4-dimensional, complex-type dataset with shape (Nblts, Nspws, 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). 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, 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 **??**.

4.2 Flags Dataset

The flags corresponding to the data are saved as a dataset named `flags`. It is a 4-dimensional, boolean-type dataset with shape (Nblts, Nspws, 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 **??** 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. 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 4-dimensional, floating-point type dataset with shape (Nblts, Nspws, 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.

Appendix A Writing Strings from h5py

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[7], 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 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.string_('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:
```

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

```
    header = f['Header']
    # read scalar dataset
    scalar_string = header['scalar_string'].value.tostring()
    assert scalar_string == 'Hello world!'

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

A.2 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.string_('Hello world!')

    # array dataset
    str_array = ['hello', 'world']
    header['array_string'] = np.string_(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'].value.tostring().decode('UTF-8')
    assert scalar_string == 'Hello world!'

    # read array dataset
    str_array_file = [n.tostring().decode('UTF-8')
                      for n in header['array_string'].value]
    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 numpy Boolean Arrays in C

As mentioned in Sec. **??**, the flags array in a UVH5 file uses an HDF5 enum datatype to encode the numpy boolean type. When creating such a datatype using `h5py`, the user simply needs to ensure the datatype is `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 Nspws = 1;
  int Nfreqs = 16;
  int Npols = 4;
  hsize_t dims[4] = {Nblts, Nspws, Nfreqs, Npols};

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

  /* make dataspace and write out data */
  dspace_id = H5Screate_simple(4, 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;
}
```