HDF5eis: A storage and I/O solution for big, multidimensional time series data from environmental sensors

Malcolm C. A. White[*], *Zhendong Zhang*[*], *Tong Bai*[*], *Hongrui Qiu*[*], *Hilary Chang*[*] *and Nori Nakata*[*]

ABSTRACT

Modern high-performance computing (HPC) tasks overwhelm conventional geophysical data formats. We describe a new data schema named HDF5eis (read H-D-F-size) for handling big, multidimensional time series data from environmental sensors in HPC applications and implement a freely available Python Application Programming Interface (API) for building and processing HDF5eis files. HDF5eis augments the popular Hierarchical Data Format 5 (HDF5) with a minimal set of additional conventions that facilitate fast and flexible data input and output protocols for regularly sampled (in time) data with any number of dimensions. HDF5eis supports arbitrary ancillary data (e.g., metadata) storage in columnar format or as UTF-8 encoded byte streams alongside time series data. Our HDF5eis API enables simple and efficient access to big data sets distributed across a potentially large number of small, heterogeneous files through a single point of access. HDF5eis outperforms conventional seismic data formats by up to two orders of magnitude in terms of random read access times. We contribute HDF5eis as an operational tool and an experimental draft proposal that will help establish the next generation of data standards in the Earth Sciences.

INTRODUCTION

Seismology is a data-driven science, and the recent emergence within the earthquake seismology community of so-called large-N recording paradigms, such as distributed acoustic sensing (DAS) (e.g., Lindsey et al., 2019) and nodal geophone arrays (e.g., Lin et al., 2013; Ben-Zion et al., 2015), has simultaneously created new opportunities for discovery and inundated researchers with a glut of data. At the same time, exploration seismologists are increasingly interested in continuous recordings from the same dense recording technology for applications such as monitoring microseismicity during hydraulic fracturing treatments of unconventional hydrocarbon reservoirs (e.g., Li and van der Baan, 2021). New recording paradigms challenge traditional distinctions between conventions for acquiring, exchanging, archiving, and processing seismic data historically associated with either earthquake (passive-source) or exploration (active-source) seismology. Whereas earthquake seismologists are accustomed to data recorded by a relatively small number of sensors (small N) over a long period of time (large T), exploration seismologists are accustomed to the exact opposite: data recorded by a large number of sensors (large N) over a short period of time (small T). With continuous recordings from dense acquisition systems becoming increasingly commonplace, large-N, large-T data are posed to become status quo in the era of "Big Data Seismology" (Arrowsmith et al., 2022).

Many of the specialized data formats at the core of seismological research today, such as (mini)SEED (Ahern et al., 2014), SAC (Goldstein et al., 2003; Goldstein and Snoke, 2005), SEG-Y (Barry et al., 1975), and SEG-D (Barry et al., 1975; Allen et al., 1994), originated in the 1970s and 1980s following the advent of digital computing. Despite the modifications and revisions that have been made to the original format definitions, modern high-performance computing (HPC) tasks strain these conventional formats far beyond their intended uses. Data input/output (I/O) operations on conventional data formats for data-hungry processing procedures, such as training deep learning models (e.g., Ross et al., 2018; Mousavi et al., 2020; Zhu and Beroza, 2018) or the massive cross-correlation calculations necessary for seismic interferometry (e.g., Wapenaar et al., 2010a,b) and matched-filter processing (e.g., Gibbons and Ringdal, 2006; Ross et al., 2019; Shelly et al., 2016; Shelly, 2020), can quickly overwhelm computational infrastructure. Although each of these conventional formats offers critical functionality for particular tasks (e.g., acquiring real-time data in the field and archiving data with standardized metadata), seismologists need new formats if they wish to reduce the time needed to extract physical insights from large data sets.

This progression towards bigger data and more intensive computing is part of a much broader trend in science and society, which means that seismologists are fortunately not facing this technological development alone. Data scientists labour continuously to develop tools for effectively handling the increasing volume and variety of data. Such tools must be *optimized* to handle large volumes of data and *flexible* to handle a wide variety of data. Optimization and flexibility, however, are often opposed to one another, and an appropriate balance between them must be found (Figure 1). The Hierarchical Data Format 5 (HDF5) (The HDF Group, 2022) is a prominent solution striking this balance for scientific data—although its applications are by no means restricted to scientific data. The seismological community has already leveraged HDF5 in specialized formats such as IRIS PASSCAL's PH5 format (Hess et al., 2017) and the Adaptable Seismic Data Format (ASDF) (Krischer et al., 2016), as has the broader Earth Science community in formats such as netCDF (Unidata, 2016) and NASA Earth Observatory System's HDF-EOS (Habermann, 2015). PH5, netCDF, and HDF-EOS, however, are primarily intended for exchanging and archiving data. Although ASDF is intended for HPC applications, design decisions associated with its definition (detailed description of these is deferred to the "Motivation" section) limit its effectiveness in analysis of multidimensional large-N, large-T data.

In this paper we present a new data schema named HDF5eis (read H-D-F-Size) that is intended to mitigate some of the limitations of the formats alluded to above.

Figure 1: The Yin and Yang of data formatting: To gain widespread use, a data form must be sufficiently optimized for a particular set of tasks while simultaneously remaining flexible. These two objectives are often opposed to one another and must be held in proper balance.

HDF5eis is an HDF5-based solution for *handling big, multidimensional time series data from environmental sensors in HPC applications*. Our goal is to develop a workhorse for I/O intensive computing in desktop and HPC cluster environments with an Application Programming Interface (API) simple enough for novice programmers to learn quickly. We present HDF5eis not as a final, one-size-fits-all solution, but rather an experimental draft proposal intended to generate discussion of how existing technology might be leveraged by new data format standards in the Earth Sciences community. HDF5eis is mainly geared towards seismic data; however we aim to keep it flexible enough to accommodate generic time series data from any environmental sensor. HDF5eis refines and improves upon some of the concepts introduced primarily by ASDF.

This article is structured as follows: After elaborating on our motivations for developing HDF5eis, we specify our design specifications, the schema definition, and our implementation of a Python API for building and processing HDF5eis files. Then we describe a hypothetical experiment that demonstrates the usefulness of HDF5eis and compare its performance against conventional seismic formats using simulated data from a hypothetical survey comprising multiple acquisition systems. We conclude with a discussion of the merits and demerits of HDF5eis.

MOTIVATION

Although it may be clear that a key feature setting HDF5eis apart from conventional seismic data formats is that it leverages the HDF5 architecture, one may well wonder what sets it apart from the more comparable PH5 and ASDF formats, which also leverage HDF5. Indeed HDF5eis is inspired by ASDF in many ways and shares many similarities with it. Because PH5 is primarily designed for archiving and exchanging standardized data and metadata from highly structured array data recorded during temporary deployments—in fact data are often converted from PH5 to SEG-Y for analysis—we focus the remainder of this section on what differentiates HDF5eis from ASDF. Below we outline key lessons learned from ASDF and how we improve upon those aspects.

First and most importantly, the atomic storage unit for regularly sampled time series data in ASDF is a one-dimensional (1-D) data array—each 1-D array is considered a discrete data object representing a single-channel time series. Although many contiguous data channels can be stored in a single ASDF file, each one is stored as a separate data object. This severely limits the efficiency of certain access patterns to multidimensional array data. Consider, for example, reading a short segment of data recorded by a DAS fiber with 1000 channels. Because ASDF stores each data channel in an independent 1-D array, contiguous channels are liable to be stored on distant disk sectors, which increases the amount of time spent seeking the relevant data to fulfill multi-channel data queries. If, however, those data were stored contiguously on disk in a single 2-D array, I/O primitives could be used much more efficiently when reading data. HDF5's *hyperslab selection* functionality provides a seamless mechanism for efficiently accessing contiguous data volumes in such a scenario. A secondary side-effect of atomizing data at such a fine level of granularity is that ASDF files bloat with excessive metadata maintained by HDF5, which hinders performance. This is only a problem, however, when tens of thousands of data segments are stored in a single file. Multidimensional arrays are supported by ASDF as so-called *auxiliary data*, but these are—as the name implies—secondary data structures rather than the emphasis of the format. HDF5eis makes multidimensional arrays the atomic storage unit around which the schema is defined.

Second, to retrieve data from an ASDF file, one must exhaustively traverse every data object in the file and parse the associated string identifiers to check for query matches. This is fine in principle, but becomes prohibitively slow in practice, particularly when using the Python API to access data in a file with highly fragmented data (e.g., a large number of channels or discontinuous/gappy data). This problem is exacerbated by the highly granular structure of ASDF files. HDF5eis mitigates this inefficiency by (a) enabling grouping of coordinate data channels and (b) maintaining a tabular index of file contents, which enables efficient regular-expression parsing to rapidly fulfill queries. We elaborate on this indexing feature later.

Third and finally, ASDF operates on each file independently. Data can thus be either (a) stored in a single, large file—this option provides a convenient access pattern, but the large files can become unwieldy—or (b) stored in many relatively small files—this option yields manageable file sizes, but increases the user's bookkeeping burden. HDF5eis, however, leverages HDF5's hierarchical design and external linking functionality to enable external links to an arbitrary number of files from a single *master* file. This master file serves as a convenient, single point of access to data distributed across a potentially large number of files. This helps maintain manageable file sizes for large data sets without sacrificing the convenience of having a single point of access. Furthermore, a single data file can be linked to multiple master files, enabling users to conveniently create multiple virtual data sets with different subsets of data without having to copy any of it.

One salient novelty of ASDF that we retain and extend is the notion of integrating existing formats for storing ancillary data alongside primitive time series data. ASDF implements this feature by defining storage structures for standard formats based on the Extended Markup Language (XML), namely QuakeML (Schorlemmer et al., 2011), StationXML, and Seis-Prov (Krischer et al., 2015) for storing earthquake catalog data, network metadata, and data provenance, respectively. These formats, however, are particular to the earthquake seismology domain and are not amenable to the broad spectrum of alternative data domains in the Earth Sciences. We extend this feature to enable storage of data in columnar format and any UTF-8 encoded byte stream. Supporting columnar data provides a natural interface to the *DataFrame* structures that are the cornerstone of powerful data science software such as `pandas` for Python, `DataFrames.jl` for Julia, and `data.frame` in R, which are well-suited for operating on relational data and support optimized data queries, data transformations, and statistical analyses. Supporting arbitrary UTF-8 encoded data implies that any text-based format, such as simple text README data and derivatives of XML or JSON, can be integrated into HDF5eis files.

DESIGN SPECIFICATIONS

In service of our primary objective—i.e., to efficiently handle big, multidimensional time series data from environmental sensors in HPC applications—we design HDF5eis to

1. store primitive time series data with arbitrary dimensionality;
2. store arbitrary ancillary data in columnar format or as UTF-8 encoded byte streams;
3. provide a single point of access to diverse data distributed across many files; and
4. simultaneously leverage existing technology and minimize external dependencies.

It should be noted that the base HDF5 API satisfies our design specifications. However, by imposing a minimal set of additional conventions, we can significantly increase its efficiency for a particular class of tasks. By this, we do not mean that we will make HDF5 itself more efficient. Rather, we aim to define a minimal set of additional structures that will make HDF5's efficiency more accessible to Earth Scientists. Defining and imposing a set of conventions on HDF5 enables implementation of universal I/O protocols and reusable, shareable code that will help Earth Scientists shift their focus from the technicalities of data management to the scientific research problems the data pertain to.

By leveraging the HDF5 software stack, HDF5eis freely inherits

1. various lossless, in-flight compression filters (e.g., GZIP, LZF, SZIP);
2. an interface for defining custom compression filters (e.g., STEIM, STEIM2);
3. advanced data indexing and memory caching algorithms for accelerating I/O operations;
4. seamless functionality for integrating structured array data with arbitrary dimensionality; and
5. hierarchical data structures that permit symbolic links to external data objects.

SCHEMA DEFINITION

Having outlined our motivations and design specifications, we now move to the HDF5eis schema definition. We wish to make a distinction at this point between the schema *definition* and *implementation*. The schema definition is a set of rules defining which data structures are permissible, whereas its implementation is a set of software for tangibly manifesting and manipulating those structures. This section outlines the HDF5eis schema definition.

At the highest level, data in each HDF5eis file are separated into three main HDF5 groups (Groups): (a) `/timeseries`, (b) `/products`, and (c) `/metadata` (Figure 2). The root Group possesses an HDF5 attribute (Attribute) named `__VERSION` which records the schema version of the file for control against future schema revisions. The structure and function of each Group is elaborated on in a top-down fashion below.

The `/timeseries` Group

Primitive time series data are stored in a nested hierarchy beneath the `/timeseries` Group. The HDF5 data set (DataSet) at the terminus of each branch in the hierarchy stores a single, continuous block of multidimensional time series data with fixed sampling intervals along each dimension. These DataSets can have any number of dimensions; however, the time axis must be oriented along the last axis. Each DataSet is addressed by a unique identifier formatted like `/timeseries/{tag}/__{start_time}Z__{end_time}Z` in which `{tag}` is a user-specified *tag* intended to differentiate collections of data, and `{start_time}` and `{end_time}` are the Coordinated Universal Time (UTC) times corresponding to the first and last samples in the data block, respectively. The `{start_time}` and `{end_time}` fields are formatted like `%Y%m%dT%H:%M:%S.%n` (see Table 1 for string format specifications). Each DataSet is also assigned an Attribute

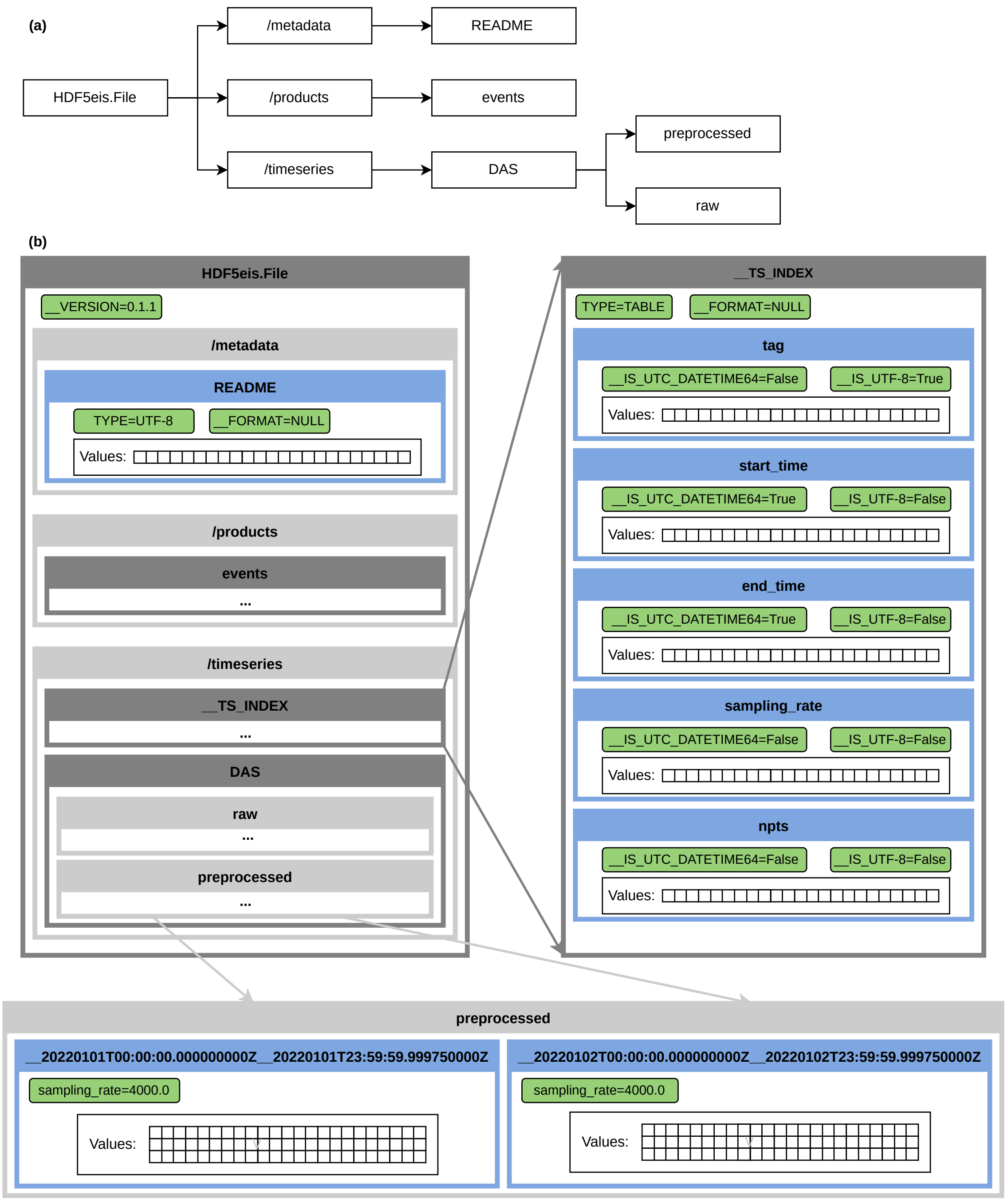

Figure 2: Schematic overview of example HDF5eis file. (a) shows the basic hierarchical layout of file contents. (b) shows a more detailed pictorial representation of file contents. Grey and blue containers represent Groups and DataSets, respectively. Green rectangles represent Attributes.
)

Table 1: String format specifications for naming data sets.

Formatter	Meaning	Range	Example
`%Y`	Four digit year	0-9999	2022
`%m`	Zero-padded, two digit month	1-12	03
`%d`	Zero-padded, two digit day of month	1-31	27
`%H`	Zero-padded, two digit hour of day	0-23	14
`%M`	Zero-padded, two digit minute of hour	0-59	19
`%S`	Zero-padded, two digit second of minute	0-59	01
`%n`	Zero-padded, nine digit nanosecond of second	0-999999999	030900120

with key `sampling_rate` and value equal to the fixed temporal sampling rate of the time series as a floating point number in units of samples per second (s^{-1}).

The `/timeseries` Group requires a nested data table under the key `__TS__INDEX` that maintains an index of all other DataSets in the `/timeseries` Group. Specifications for storing tabular data are discussed in the next subsection. For each DataSet in the `/timeseries` Group, there must be a corresponding row in the `__TS__INDEX` table with `tag`, `start_time`, `end_time`, `sampling_rate`, and `npts` columns. The `tag`, `start_time`, `end_time`, and `sampling_rate` columns record the variables by the same name in the paragraph above. The `npts` column must be the integer number of time samples comprising the DataSet. This `__TS__INDEX` table can be quickly parsed to find the DataSets needed to fulfill a user query. No other tables are permitted in the `/timeseries` Group.

The `/products` and `/metadata` Groups

The `/products` and `/metadata` Groups are fundamentally different from the `/timeseries` Group in that the former provide storage and I/O for tabular data comprising columnar data with potentially mixed data types, and UTF-8 encoded byte streams, whereas the latter provides storage and I/O for structured, multidimensional arrays of time series data with uniform type. The difference between the `/products` and `/metadata` Groups is merely an organizational convention: `/products` is intended for data products that are derived from or otherwise related to the data in `/timeseries` (e.g., earthquake catalog data), whereas `/metadata` is intended for static metadata associated with `/timeseries` data (e.g., recording geometry, instrument responses, and data provenance).

Tabular data

Tabular data are organized in Groups (one per table) nested within the `/products` and `/metadata` Groups (with the exception of the `/timeseries/__TS__INDEX` table) and addressed by `/{parent}/{key}` in which `{parent}` is either `/products` or

`/metadata`, as appropriate, and `{key}` is a user-specified *key* value. To differentiate tabular data from UTF-8 encoded byte streams, Groups corresponding to tabular data are each assigned an Attribute with key `__TYPE` and value `TABLE`. DataSets corresponding to UTF-8 encoded byte streams, on the other hand, are each assigned a similar Attribute with value `UTF-8`. Each column within a table is stored in an independent DataSet addressed by `/{parent}/{key}/{column}` in which `{parent}` and `{key}` are as above and `{column}` is the column name. Each column may have a different data type, but data within a single column must be of homogeneous type, and all columns within a single table must have the same length. Valid data types for columnar data are numeric types (integers and floats), character strings, and timestamp or *datetime* values.

Because the HDF5 backend does not support a native datetime data type, special care must be taken when storing tabular data with columns of datetime values. HDF5eis stores datetime data as 64-bit integers representing the number of nanoseconds since 1970-01-01 00:00:00 (UTC). Every DataSet nested within a Group for which the `__TYPE` Attribute is equal to `TABLE` requires an `__IS_UTC_DATETIME64` Attribute with boolean value `True` or `False`. This flag indicates how to properly interpret (and potentially convert) the data at read time.

Columnar data comprising character strings must similarly be handled carefully. Just as every DataSet nested within a Group for which the `__TYPE` Attribute is equal to `TABLE` requires an `__IS_UTC_DATETIME64` Attribute, so too do they require a similar `__IS_UTF-8` Attribute. Character strings are stored as raw UTF-8 encoded bytes with a sufficient number of bytes per string to represent the longest string in the column. Setting the `__IS_UTF-8` Attribute to `True` indicates that these bytes should be appropriately decoded at read time.

UTF-8 encoded data

UTF-8 encoded data can be stored in columnar format when they constitute part of a data table (e.g., alphanumeric station IDs in a network geometry metadata table), or they can simply be stored as individual byte streams of arbitrary length. Each byte stream is stored in a separate DataSet, which is addressed by `/{parent}/{key}` in which `/{parent}` and `{key}` are as above. Also as above, each such DataSet must have a `TYPE` Attribute with value `UTF-8`. The rationale for permitting arbitrary UTF-8 encoded data is to enable existing formats to be leveraged. For example the StationXML, QuakeML (Schorlemmer et al., 2011), and SEIS-PROV (Krischer et al., 2015) standards can be used to keep records of station metadata, seismic events, and data provenance, respectively, as is done in ASDF. Alternatively, any other text-based format, even a simple text README, can be stored as a UTF-8 encoded byte stream.

Groups corresponding to tabular data and DataSets corresponding to UTF-8 encoded byte streams each possess a `__FORMAT` Attribute that records the format of the stored data. As examples, setting `__FORMAT` to `CSS3.1-origin` could be used to indicate that a table records seismic event origin data using the Center for Seismological Studies CSS3.1 database schema, or `STATIONXML` might be used to indicate that a UTF-8 encoded byte stream can be parsed using the STATIONXML protocol.

APPLICATION PROGRAMMING INTERFACE (API) IMPLEMENTATION

We implement a Python API to build and process HDF5eis files, which is publicly available at `https://github.com/malcolmw/HDF5eis`. Although HDF5eis's schema definition depends only on HDF5, the Python API we implement introduces a few additional dependencies. We use the `h5py` API to interact directly with the underlying HDF5 files. In addition to `h5py`, we use the `pandas` package (McKinney, 2010) to manipulate tabular data in memory. In particular, we manipulate the `__TS__INDEX` table using `pandas` to rapidly fulfill time series queries—specifying the tag and start and end time of the desired samples is sufficient to determine precisely which DataSet(s) contain them and at which offsets they reside within their respective DataSet(s). Our API also serves tabular data from HDF5eis files as `pandas.DataFrame` objects. Finally, we use the `numpy` library (Harris et al., 2020) to manipulate time series data in memory and perform various other basic operations. These three dependencies (`h5py`, `pandas`, and `numpy`) are mature, stable packages that are unlikely to introduce breaking changes in the near future. Our API is most easily learned through the tutorials that are included with the distribution.

Because HDF5eis is designed to maximize flexibility, the mapping between existing formats and HDF5eis is non-unique. Multiple valid HDF5eis structures could be implemented to represent data in an existing format, and the optimal file structure will depend on the intended use case. The non-uniqueness of this mapping from conventional data formats to HDF5eis inhibits us from implementing a simple program for converting data. HDF5eis users, at least those creating HDF5eis files, need to have some understanding of the internal HDF5eis structures to convert data from existing formats. Although it is a fairly straightforward exercise, users must write a script to convert their data to HDF5eis. To facilitate this process and demonstrate additional nuances of the API not covered in this paper, we include a few tutorials in the distribution.

EXAMPLE USE CASE

To illustrate the advantages of implementing a multidimensional array data structure and basic features of our API, we now consider a hypothetical experiment monitoring microseismic activity during a hydraulic fracturing treatment. In this section we describe the experimental design and a basic structure for handling the data using HDF5eis. In the "Comparison Against Alternative Formats" section we simulate a small subset of the data that would be recorded during such an experiment and

Table 2: Summary of data acquisition parameters in hypothetical 30 d experiment monitoring microseismicity during hydraulic fracturing treatment.

Component	# of channels	Sampling rate (s^{-1})	Data size
DAS	1024	4000	38.62 TB
DTS	1024	60^{-1}	168.8 MB
Downhole geophones	$24 \times 3 = 72$	500	347.6 GB
Surface geophones	$16 \times 16 \times 3 = 768$	500	3.621 TB
All	2888	NA	42.58 TB

compare the performance of various data formats for handling the data.

The data acquisition system for our hypothetical experiment comprises four separate components: a) a downhole DAS fiber, b) a downhole distributed temperature sensing (DTS) fiber, c) a 1-D array of downhole geophones, and d) a 2-D array of surface geophones. Data are acquired from all four of these components continuously over a 30 d period. The DAS and DTS fibers both record 1024 channels of data. The DAS data are sampled at a rate of 4000 s^{-1}, whereas DTS data are sampled at a rate of 1 min^{-1}. The downhole geophone array comprises 24 three-component (3C) sensors sampled at a rate of 500 s^{-1}. The 2-D array of surface geophones comprises a 16×16 array of 3C sensors also sampled at a rate of 500 s^{-1}. Data recorded in this configuration and stored with 32-bit floats will require 43 TB of storage (Table 2). Simply organizing storage for such a data set is a significant technical challenge, but this challenge can be handled efficiently using HDF5eis as follows.

DAS data are amenable to being stored on disk using 2-D arrays with shape `(nchannels, npts)` in which `nchannels` is the number of channels being recorded and `npts` is the number of temporal samples. Allocating one file for every 5 min of uncompressed DAS data requires 8640 files consuming roughly 4.6 GB each, which is manageable on any modern computer with enough storage. It is also important to consider the *chunk layout* and compression algorithm when creating HDF5 files, as they will significantly impact storage and I/O performance, but both of these aspects are optional and a discussion of them is beyond the scope of this thought experiment. DAS data can be associated with tag `DAS`.

DTS data are also amenable to being stored on disk using 2-D arrays with shape `(nchannels, npts)`. Because DTS data are sampled infrequently, they amount to less than 1 GB of data and can easily be stored in a single file. DTS data can be associated with tag `DTS`.

Downhole geophone data are amenable to being stored on disk using 3-D arrays with shape `(nsensors, ncomponents, npts)` in which `nsensors = 24` is the number of recording sensors and `ncomponents = 3` is the number of components per sensor. Allocating one file for every 6 h of downhole geophone data requires 120 files consuming roughly 2.9 GB of storage each. Downhole geophone data can be associated with tag `geophones/downhole`.

Surface geophone data are amenable to being stored on disk using 4-D arrays with shape `(nrows, ncolumns, ncomponents, npts)` in which `nrows = ncolumns = 16` are the number of sensor rows and columns, respectively. Allocating one file for every 30 min of surface geophone data requires 1440 files consuming roughly 2.6 GB of storage each. Surface geophone data can be associated with tag `geophones/surface`.

All 10 201 files created in the configuration above can be linked to and subsequently accessed through a single master file using our Python API. Array geometry can also be stored in tabular form under the `/metadata` Group. The user no longer needs to maintain a record of the location of each individual file, even if they are distributed across multiple disks on a network file system. Furthermore, the user can request data spanning multiple files without any knowledge of which samples come from which files; the necessary bookkeeping is done by the API. The structure of the resultant master file may look as shown in Figure **??**.

Not only do such data structures simplify bookkeeping for such large data sets, they also enable efficient I/O for many common access patterns via hyperslab selection (a.k.a. array slicing). Data subsets specified by start and end indices and stride length along each storage dimension can be extracted in a single call to the HDF5 library. For instance, each of the following subsets of data can be extracted in a single API call that leverages HDF5 hyperslab selection functionality for efficient I/O (Figure 3):

(a) DAS data from all channels.

(b) DTS data from channels 256 through 511.

(c) Vertical-component data from downhole geophones 8 through 11.

(d) Three-component data from every second geophone in row 12 of the surface array.

The above storage requirements are based on the assumption that data are uncompressed; however, one may wish to compress the data. HDF5eis inherits and exposes various compression algorithms, which can be configured to suit one's particular needs. For example, the GZIP compression algorithm can be used with compression level 9 (maximum compression) for long-term archival or data exchange purposes. Entire files can be retrieved and decompressed for processing as needed. Alternatively, one may compress the data with a more moderate compression level and decompress it in an on-the-fly manner. In this scenario, small chunks of compressed data can be transmitted from a network file system to a processing node where it can be decompressed for processing. This only requires decompressing the requested chunks of data (not the entire file) and can be optimized to balance demands on storage, network, and compute resources.

```python
import hdf5eis

with hdf5eis.File("demo.hdf5", mode="r") as f5:
    # (a) Extract one second of DAS data for all channels.
    start_time, end_time = "2022-01-01T00:00:00Z", "2022-01-01T00:00:01Z"
    das = f5.timeseries["DAS", :, start_time, end_time]

    # (b) Extract one hour of DTS data from channels 256 through 511.
    start_time, end_time = "2022-01-01T00:00:00Z", "2022-01-01T01:00:00Z"
    dts = f5.timeseries["DTS", 256: 512, start_time: end_time]

    # (c) Extract ten seconds of vertical-component data from downhole
    #     geophones 8 through 11.
    start_time, end_time = "2022-01-01T00:00:00Z", "2022-01-01T00:00:10Z"
    dwn = f5.timeseries["geophones/downhole", 8: 12, 0, start_time: end_time]

    # (d) Extract four seconds of three-component data from every second
    #     geophone in row 12 of the surface array.
    start_time, end_time = "2022-01-01T00:00:00Z", "2022-01-01T00:00:04Z"
    srf = f5.timeseries["geophones/surface", 12, ::2, :, start_time: end_time]

    # (e) Extract geometry metadata for DAS data.
    das_geom = f5.metadata["geometry/DAS"]
```

Figure 3: Illustrative API syntax for efficiently reading various subsets of data from a hypothetical multi-system survey (described in the “Example Use Case” section): (a) one second of DAS data for all channels, (b) one hour of DTS data for channels 256 through 511, (c) ten seconds of vertical-component data from downhole geophones 8 through 11, (d) four seconds of 3C data from every other geophone in row 12 of the surface array, and (e) geometry metadata for DAS fiber.

COMPARISON AGAINST ALTERNATIVE FORMATS

In this section, we describe results from two experiments designed to assess the relative performance of HDF5eis against SEG-Y, miniSEED, and ASDF. SEG-Y is chosen as a representative of exploration seismic data formats. MiniSEED is chosen as a representative of earthquake seismic data formats. ASDF is chosen as a comparable format to HDF5eis because it also leverages HDF5. The code for these experiments is included as a Jupyter Notebook in the Supplementary Materials for readers who wish to validate these results. The `segyio` (Equinor ASA, 2022), `obspy` (Beyreuther et al., 2010), and `pyasdf` (Krischer et al., 2016) Python packages are used to read and write SEG-Y, miniSEED, and ASDF data, respectively.

The first experiment is designed to simulate random reads from the DAS data set described in the previous section. We generate random sample values to simulate real data recorded by 1024 channels with $4000\,\mathrm{s}^{-1}$ sampling rate over a 30 min period. Data are stored in uncompressed 32-bit floating point representation for all formats. HDF5eis data are stored in a single file per 5 min period as described in the previous section. SEG-Y data are also stored in a single file per 5 min period. MiniSEED data are stored in one file per channel. All ASDF data are stored in a single file. We record the average amount of time required to read random segments of data.

At each realization, we read data from a random subset of sequential channels between random start and end times. The number and position of the channels, the start time, and the duration of the data read are all drawn from uniform random distributions. The duration of data read at each iteration is between 0.5 and 2 s. The same random subset is read from each data format and corresponding read times are recorded for 1000 iterations. Figure 4a shows the read times in multiples of the corresponding HDF5eis read time (values greater than one indicate slower reads than HDF5eis, whereas values less than one indicate faster reads). SEG-Y is substantially more efficient than miniSEED and ASDF, which is unsurprising given that SEG-Y is intended for such large-N array data. The median relative read time for SEG-Y, however, is 6.5 times that of HDF5eis. Median relative read times for miniSEED and ASDF are 22.9 and 40.6, respectively. HDF5eis is an order of magnitude faster than miniSEED and ASDF, and it achieves this performance with the fewest lines of client code. Although high-level APIs could be developed to read data from other formats using similar amounts of client code, pithy code is characteristic of HDF5eis because its multidimensional array storage structure naturally leverages HDF5's hyperslab selection functionality. Notably, ASDF is the slowest of the formats. This is because of the first and second design features outlined in the "Motivations" section.

The second experiment is designed to simulate random reads from the data set recorded by the 2-D array of geophones described in the "Example Use Case" section. We again use random sample values to simulate recorded data, this time for a 2-D array of 3C geophones with 16 rows, 16 columns, and $500\,\mathrm{s}^{-1}$ sampling rate over a 1 d period. MiniSEED and ASDF data are stored in one file per channel and a single file, respectively, as in the first experiment. SEG-Y and HDF5eis data are stored in

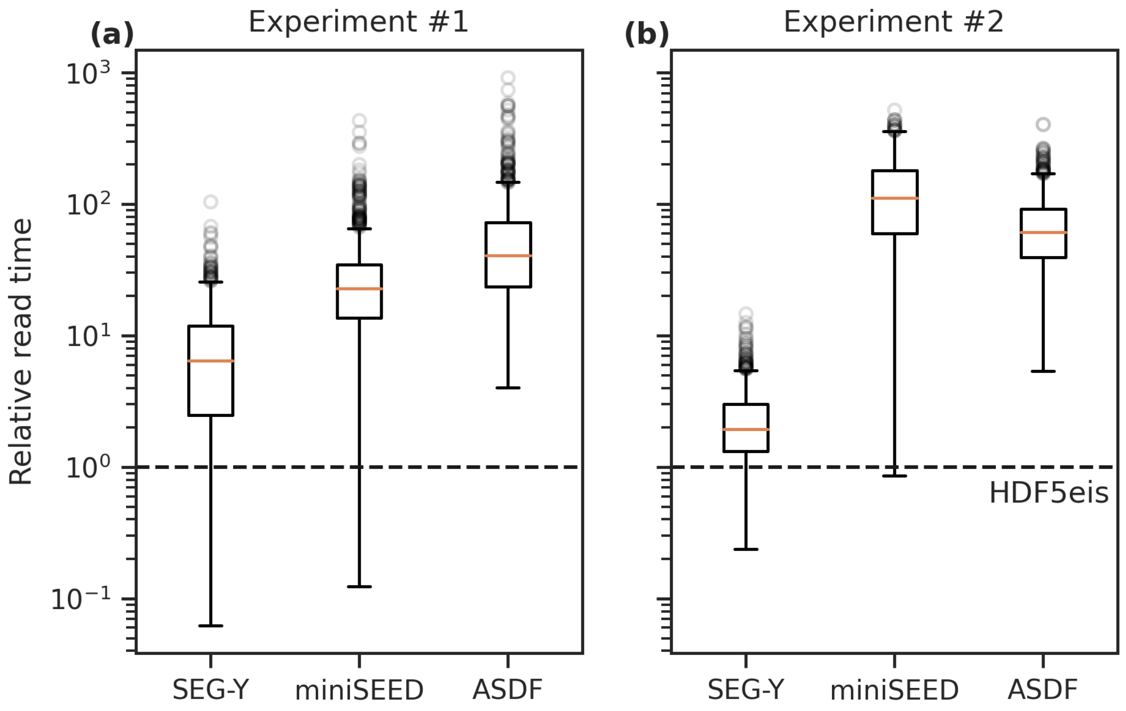

Figure 4: Comparison of random read access times for representative seismic data formats. (a) shows results for simulated DAS data. (b) shows results for simulated 2-D geophone array. Experimental designs are described in the "Example Use Case" section. Read times are reported in multiples of corresponding HDF5eis read time. Horizontal orange lines represent median relative read time. Boxes span the first and third quartiles. Whisker lengths are 1.5 times the interquartile range. Outliers are shown as translucent black circles. Horizontal black dashed line represents HDF5eis read time, which is unity by definition.

one file per 30 min period.

At each realization, a random segment of data between 2 and 6 s long is read from a random subset of rows, columns, and channels starting at a random start time. As in the first experiment, all random variables are drawn from uniform random distributions. 1000 iterations are performed and the results summarized in Figure 4b. All median relative read times are again greater than one, however in this experiment, the median relative read time for SEG-Y is only 1.9. Median relative read times for miniSEED and ASDF are 105.1 and 63.0, respectively. HDF5eis is nearly twice as fast as SEG-Y and two full orders of magnitude faster than miniSEED in this context.

DISCUSSION

In this article we present a data schema, which we name HDF5eis, for archiving, exchanging, and processing big, multidimensional time series data from environmental sensors in HPC applications using HDF5. We contribute it as (a) a functional tool and (b) an experimental draft proposal to provide a basis on which to discuss the merits and demerits of existing technology and how they might be leveraged in future data formats for the Earth Sciences. Although our focus is seismic data, we recognize that time series data from many disciplines within the Earth Sciences share similarities, and we have aimed to accommodate a wide variety of data types by maintaining schema flexibility. To promote interdisciplinary interoperability and collaboration, standards for future data formats in the Earth Sciences should be the product of inter-generational, interdisciplinary committees. In contribution to such potential future efforts, we now discuss the merits and demerits of HDF5eis.

First, HDF5eis inherits impressive I/O performance from the highly optimized HDF5 library (Figure 4). Because extensive effort has been invested by the HDF Group in implementing and optimizing advanced storage layout schemas, memory caching algorithms, and data indexing protocols, it is likely that only a highly skilled team of data scientists will be able to improve on these. Whereas conventional seismic data formats rely on their own schemas and protocols, future formats will benefit greatly from leveraging advances made by data specialists.

Second, HDF5eis is flexible. In fact, its I/O performance owes considerably to its flexibility. Supporting arrays of arbitrary dimensionality enables users to conform storage structures to the characteristics of their specific data set. Such multidimensional array structures also enable efficient retrieval of coordinate data channels via hyperslab selection. Furthermore, because HDF5eis implements storage for columnar data and UTF-8 encoded byte streams, many existing data format standards can be internally maintained. For example, QuakeML (Schorlemmer et al., 2011) can be used to store earthquake catalog data, and because HDF5eis is independent of QuakeML, it is future proof against changes to the QuakeML schema. The same can be said of any analogous format. Alternatively, one could store earthquake catalog data in tabular format as part of a relational database. This flexibility and interoperability with existing formats will reduce the overhead associated with adapting software to operate on HDF5eis files.

Third, HDF5eis freely inherits multiple lossless, in-flight compression algorithms and functionality to define custom filters. These enable the user to tailor the level of compression to their specific needs and balance demands placed on network, storage, and processing resources. Users do not need to decompress an entire file when they only need a small subset of its contents; however users may choose to heavily compress data files (e.g., for transfer or long-term, cold storage) and decompress files wholesale for processing.

Fourth, HDF5eis files are self describing and platform independent which makes them universally portable. A user in one computational environment will have no problem reading data written in an entirely different environment. Moreover, HDF5eis files can be manipulated in a wide variety of languages because HDF5 APIs exist for many of them including C/C++, Fortran, Java, Python, Julia, and MATLAB. Although comparable tools to HDF5 exist, such as *zarr* and *xarray*, HDF5 is significantly more mature, has a wide user base, and has APIs for the greatest number of languages.

Fifth and finally, use of external links in HDF5eis minimizes the bookkeeping burden borne by the user. Users of our Python API can create zero-copy virtual data sets and seamlessly read data across file boundaries through a single point of access without worrying about which files samples reside in. Users can simultaneously access diverse ancillary data (e.g., metadata) through the same single point of access using a unified programming interface. This allows users to rise above the mundanities of data management and focus instead on scientific research.

HDF5eis is not, however, without limitations.

First, HDF5eis is not cloud native, meaning that it is not optimized for interacting with data on cloud resources. It is our opinion that future data formats in the Earth Sciences should work at least as well on the cloud as they do on a local workstation or computing cluster. The Highly Scalable Data Service (HSDS) from the HDF Group, is one technology that could be leveraged in a cloud-native version of HDF5eis. HSDS is designed to maximize interoperability with existing HDF5 data and libraries, which means that migrating to HSDS from HDF5 would engender relatively little overhead. However this also means that HSDS design decisions are influenced by non-cloud-native technology, which may ultimately detract from cloud performance. As cloud-computing paradigms mature, so do the tools designed specifically for the cloud. Emerging cloud technologies, such as TileDB (Papadopoulos et al., 2016), offer great promise for providing dyed-in-the-wool cloud-native backend data formats that can be leveraged in the Earth Sciences.

Second, HDF5eis currently only supports the UTC time standard. To be more broadly useful, formats for time series data from environmental sensors should implement multiple time standards including Mean Solar Time, Atomic Time, and GPS Time. In addition to these, Elapsed Time, representing the time elapsed since some arbitrary datum, would be useful for storing data without an absolute time frame, such as Green functions and (cross-)correlograms, or data for which the elapsed time is more important than the absolute time, such as in typical exploration seismology. Adding support for these time standards is fairly straightforward in principle and may be the subject of future development, depending on user interest.

Third, HDF5eis only supports data sampled at regular time intervals. Because many sensor types sample at irregular intervals, supporting such data is important. Again, adding support for irregularly sampled time series is fairly straightforward and may be the subject of future development.

Fourth, HDF5eis is not designed for real-time data acquisition and may not be suitable in certain contexts. For instance, data packets transmitted by regional seismic networks cannot be assumed to arrive in proper time order. Recording such data using HDF5eis, particularly when gaps in data transmission occur, is likely to be cumbersome and may produce inefficient data structures.

CONCLUSION

HDF5 is a suitable substrate on which to build efficient and flexible data formats for handling big, multidimensional time series data from environmental sensors in HPC applications. Implementing a simple indexing protocol, as is done by HDF5eis, can yield superior I/O performance in comparison with conventional seismic data formats. The time required to extract physical insights from large data sets can thus be effectively reduced when processing is I/O bound. HDF5eis is a functional tool for simplifying complex access patterns to diverse data in distributed computing environments. As data volumes swell and cloud technology matures, data-driven research in the Earth Sciences will increasingly demand data standards beyond the scope of those that exist today. HDF5eis sheds needed light on the path towards establishing standards for the next generation.

REFERENCES

Ahern, T. K., R. Casey, D. Barnes, R. Benson, and T. Knight, 2014, Seed Reference Manual v2.4.

Allen, R., G. Crews, W. Guyton, C. A. McLemore, B. Peterson, C. S. Rapp, L. Walker, L. R. Whigham, D. A. White, and G. Wood, 1994, Digital field tape format standards — SEG-D, Revision 1: Geophysics, **59**, 668–684.

Arrowsmith, S. J., D. T. Trugman, J. MacCarthy, K. J. Bergen, D. Lumley, and M. B. Magnani, 2022, Big Data Seismology: Reviews of Geophysics, **60**, 1–55.

Barry, K. M., D. A. Cavers, and C. W. Kneale, 1975, Recommended standards for digital tape formats: Geophysics, **40**, 344–352.

Ben-Zion, Y., F. L. Vernon, Y. Ozakin, D. Zigone, Z. E. Ross, H. Meng, M. White, J. Reyes, D. Hollis, and M. Barklage, 2015, Basic data features and results from a spatially dense seismic array on the San Jacinto fault zone: Geophysical Journal International, **202**, 370–380.

Beyreuther, M., R. Barsch, L. Krischer, T. Megies, Y. Behr, and J. Wassermann, 2010, ObsPy: A Python toolbox for seismology: Seismological Research Letters, **81**, 530–533.

Equinor ASA, 2022, segyio [software].

Gibbons, S. J., and F. Ringdal, 2006, The detection of low magnitude seismic events using array-based waveform correlation: Geophysical Journal International, **165**, 149–166.

Goldstein, P., D. Dodge, M. Firpo, L. Minner, W. Lee, H. Kanamori, P. Jennings, and C. Kisslinger, 2003, SAC2000: Signal processing and analysis tools for seismologists and engineers: The IASPEI international handbook of earthquake and engineering seismology, **81**, 1613–1620.

Goldstein, P., and A. Snoke, 2005, SAC availability for the IRIS community: Incorporated Institutions for Seismology Data Management Center Electronic Newsletter, **7**.

Habermann, T., 2015, HDF Earth Science Program.

Harris, C. R., K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, 2020, Array programming with NumPy: Nature, **585**, 357–362.

Hess, D., S. Azevedo, N. Falco, and B. C. Beaudoin, 2017, PH5: HDF5 based format for integrating and archiving seismic data: AGU Fall Meeting Abstracts, IN42B–03.

Krischer, L., J. Smith, W. Lei, M. Lefebvre, Y. Ruan, E. S. de Andrade, N. Podhorszki, E. Bozdağ, and J. Tromp, 2016, An adaptable seismic data format: Geophysical Journal International, **207**, 1003–1011.

Krischer, L., J. A. Smith, and J. Tromp, 2015, SEIS-PROV: Practical provenance for seismological data: AGU Fall Meeting Abstracts, S53A–2768.

Li, Z., and M. van der Baan, 2021, Reverse-time imaging of DAS microseismic data: First International Meeting for Applied Geoscience Energy Expanded Abstracts, Society of Exploration Geophysicists, 392–396.

Lin, F.-C., D. Li, R. W. Clayton, and D. Hollis, 2013, High-resolution 3D shallow crustal structure in Long Beach, California: Application of ambient noise tomography on a dense seismic array: Geophysics, **78**, no. 4, Q45–Q56.

Lindsey, N. J., T. C. Dawe, and J. B. Ajo-Franklin, 2019, Illuminating seafloor faults and ocean dynamics with dark fiber distributed acoustic sensing: Science, **366**, 1103–1107.

McKinney, W., 2010, Data structures for statistical computing in Python: Proceedings of the 9th Python in Science Conference, 56–61.

Mousavi, S. M., W. L. Ellsworth, W. Zhu, L. Y. Chuang, and G. C. Beroza, 2020, Earthquake transformer—an attentive deep-learning model for simultaneous earthquake detection and phase picking: Nature Communications, **11**, 3952.

Papadopoulos, S., K. Datta, S. Madden, and T. Mattson, 2016, The TileDB array data storage manager: Proceedings of the VLDB Endowment, **10**, 349–360.

Ross, Z. E., M.-A. Meier, E. Hauksson, and T. H. Heaton, 2018, Generalized seismic phase detection with deep learning: Bulletin of the Seismological Society of America, **108**, 2894–2901.

Ross, Z. E., D. T. Trugman, E. Hauksson, and P. M. Shearer, 2019, Searching for hidden earthquakes in Southern California: Science, **364**, 767–771.

Schorlemmer, D., F. Euchner, P. Kästli, and S. Joachim, 2011, QuakeML: Status of the XML-based seismological data exchange format: Annals of Geophysics, **54**, 59–65.

Shelly, D. R., 2020, A high-resolution seismic catalog for the initial 2019 Ridgecrest earthquake sequence: Foreshocks, aftershocks, and faulting complexity: Seismological Research Letters, **91**, 1971–1978.

Shelly, D. R., W. L. Ellsworth, and D. P. Hill, 2016, Fluid-faulting evolution in high definition: Connecting fault structure and frequency-magnitude variations during the 2014 Long Valley Caldera, California, earthquake swarm: Journal of Geophysical Research: Solid Earth, **121**, 1776–1795.

The HDF Group, 1997-2022, Hierarchical Data Format, version 5. (https://www.hdfgroup.org/HDF5/).

Unidata, 2016, Network Common Data Form (netCDF).

Wapenaar, K., D. Draganov, R. Snieder, X. Campman, and A. Verdel, 2010a, Tutorial on seismic interferometry: Part 1 — Basic principles and applications: Geophysics, **75**, 75A195–75A209.

Wapenaar, K., E. Slob, R. Snieder, and A. Curtis, 2010b, Tutorial on seismic interferometry: Part 2 — Underlying theory and new advances: Geophysics, **75**, 75A211–75A227.

Zhu, W., and G. C. Beroza, 2018, PhaseNet: A deep-neural-network-based seismic arrival time picking method: Geophysical Journal International, 261–273.