Frames and their representations

Brett Viren

July 23, 2024

Contents

1 Intro **1**

2 Initial I/O with FrameFileSink/FrameFileSource **1**

3 Tag solutions **2**

4 Frame decomposition **2**
4.1 Set metadata 3
4.2 Tensors 3
4.3 Trace 3
4.4 Index 4
4.5 Summary 4

1 Intro

A WCT "frame" provides a simple representation of data from different types of "readout" of a LArTPC detector with some additional concepts bolted on. This describes the `IFrame` representation and a more generic representation based on `ITensorSet` which is a useful intermediate for sharing frame information with non-WCT things like files.

A WCT `IFrame` holds an `ident` number, a reference `time` and a sample period duration `tick`. It also holds one or more "frame tags" which are simple strings providing application-defined identifiers. A frame also holds three collections. The most important is `traces` which is a vector of `ITrace`. It also holds a "trace tag map" which maps from a string (the "trace tag") to a vector of indices into the `traces` vector. This provides application-defined identification of a subset of traces. Finally, it provides a "channel mask map" which again maps a string "mask tag" to a mapping from channel ident to a list of tick ranges.

The `ITrace` holds an offset from the frame reference `time` measured in number of `ticks`, a channel ident and a floating point array, typically identified as holding an ADC or a signal waveform (or fragment thereof). The `traces` array can be thought of as a 2D sparse or dense array spanning the channels and ticks of the readout.

At least two persistent representations of `IFrame` are supported by WCT. The rest of this d

2 Initial I/O with FrameFileSink/FrameFileSource

WCT `sio` subpackage provides support for sending `IFrame` through file I/O based on `boost::iostreams` and `custard` to support streams through `.tar` with optional `.gz`, `.xz` or `.bz2` compression or `.zip`

files. The `FrameFile{Sink,Source}` uses `pigenc` to stream `IFrame` in the form of Python Numpy `.npy` files. The `IFrame` is decomposed into a number of arrays and a file naming scheme is used to encode array type and frame `ident` as well as tag information.

This initial sink/source pair introduced problems. The sink produced a stream of arrays with file names like:

```
frame_<tag1>_<ident>.npy
channels_<tag1>_<ident>.npy
tickinfo_<tag1>_<ident>.npy
summary_<tag1>_<ident>.npy  (optional)
```

In deciding what to write, the `FrameFileSink` matches each tag in the union of the `IFrame` trace and frame tags against a configured list. Upon first match, the scope that is tagged is written in a trio of files as shown above. Thus the frame vs trace nature of the tag was lost.

As only Numpy `.npy` files are supported in `FrameFile{Sink/Source}` files, the nested structure of the `IFrame` channel mask map has to be flattened into a number of arrays named like:

```
chanmask_<tag1>_<ident>.npy
chanmask_<tag2>_<ident>.npy
```

The tags in this case are the channel mask map keys (eg, `"bad"`).

Despite these problems, this format is convenient for producing per-tag dense arrays for quick debugging using Python/Numpy. However, this breakage of data fidelity makes this format unsuitable for lossless I/O. And the array ordering requirements of `FrameFileSource` make producing compatible files by software other than `FrameFileSink` awkward.

3 Tag solutions

To fix the above, it was attempted to modify the `FF{Sink,Source}` to work in a new mode, which retaining backward compatibility and to add support for sending channel mask maps. This quickly became a mess, especially when one must fit any schema into the confines of flat arrays an file naming conventions.

At the same time, development was ongoing to add point-cloud support and in particular a generic `TensorFileSink` was developed. Given that, it became clear that the problems in a new `IFrame` I/O can be decoupled into two parts: 1) define an `ITensorSet` based schema to represent `IFrame` data with high fidelity and 2) use `TensorFileSink` as-is and develop the anyway required `TensorFileSource`.

This approach is attractive for two other reasons. `ITensor` and `ITensorSet` both support structured, JSON-like metadata objects. This relieves the burden on using naming schemes to hold metadata. Second, `ITensor` representation is (intentionally) natural to use for HDF5 files and ZeroMQ transport and thus by converting between `IFrame` and `ITensor` representations, frames get new I/O methods "for free".

4 Frame decomposition

The `IFrame` info is decomposed into an `ITensorSet/ITensor` representation.

4.1 Set metadata

The `IFrame::ident()` is mapped directly to `ITensorSet::ident()`.

The correspondence between `ITensorSet`-level metadata attribute names and the `IFrame` methods providing the metadata value are listed along with their value type.

`time` `IFrame::time()` float

`tick` `IFrame::tick()` float

`masks` `IFrame::masks()` structure

`tags` `IFrame::frame_tags()` array of string

When the set-level metadata is represented as a JSON file its name is assumed to take the form `frame_<ident>.json`. When `IFrame` data in file representation are provided as a stream, this file is expected to be prior to any other files representing the frame. The remaining files are expected to hold tensors and must be contiguous in the stream but otherwise their order is not defined. These tensors are described in the remaining sections.

4.2 Tensors

An `ITensor` represents some aspect of an `IFrame` not already represented in the set-level metadata. Each tensor provides at least these two metadata attributes:

`type` a label in the set `{trace, index, summary}` identifying the aspect of the frame it represents.

`name` an instance identifier that is unique in the context of all `ITensor` in the set of the same `type`.

The values for both attributes must be suitable for use as components of a file name. File names holding tensor level array or metadata information are assumed to take the forms, respectively `frame_<ident>_<type>_<name>.{json,npy}`.

The remaining sections describe each accepted type of tensor.

4.3 Trace

A trace tensor provides waveform samples from a number of channels. Its array spans a single or an ordered collection of channels. A single-channel trace array is 1D of shape `(nticks)` while a multi-channel trace array is 2D of shape `(nchans,nticks)`. Samples may be zero-padded and may be of type `float` or `short`. The ident numbers of the channels is provided by the `chid` metadata which is scalar for a single channel trace tensor and 1D of shape `(nchans)` for a multi-channel trace tensor.

- `tbin=N` the number of ticks prior to the first tensor column
- `chid=<int-or-array-of-int>` the channel ident numbers
- `tag="tag"` an optional trace tag defining an implicit index tensor

If `tag` is given it implies the existence of an index of tagged traces spans the trace tensor. See below for other ways to indicate tagged traces.

`IFrame` represents traces as a flat, ordered collection of traces. When more than one trace tensor is encountered, its traces are appended to this collection. This allows sparse or dense or a hybrid mix of trace information. It also allows a collection of tagged traces to have their associated waveforms represented together.

4.4 Index

A subset of traces held by the frame is identified by a string ("trace tag") and its associated collection of indices into the collection of traces. Such an index may be represented implicitly with a `tag` attribute of a trace tensor metadata or explicitly with an index tensor. An optional `traces` metadata attribute may be given which names a trace tensor (its `name` not its `tag`). In such case, the array is interpreted as indexing relative to the rows from that trace tensor. If `traces` is omitted or its value is the empty string, its indices are considered relative to the frame's entire collection of traces

`tag="tag"` a unique string ("trace tag") identifying this subset

`traces=<name-or-empty-"">` a trace tensor name or the empty string.

4.5 Summary

A trace summary tensor provides values associated to indexed (tagged) traces. The tensor array elements are assumed to map one-to-one with indices provided by an index tensor with the matching tag. The additional metadata:

`tag="tag"` the associated index trace tag.

Note, it is undefined behavior if no matching index tensor exists.