User Software :: CR-Tools :: DataReader

Overview

The `DataReader` class implements the processing framework, which can be applied to data before entering further processing.

Fields in the Header Record

This is the list of (mandatory) fields in the header record (as accessed with `dr.header()`) of a DataReader object. The mandatory fields have to be set by all child classes of the DataReader in order to be usable by the (upcoming) standard tools.

Field Name	M*	Data Type	Desctription
Date	☒	uInt	Date of the observation. Standard unix date i.e. (GMT-)seconds since 1.1.1970
AntennaIDs	☒	Vector<Int>	The IDs of the antennas, i.e. an unique number for each channel.
Observatory	☒	String	Name of the Observatory, e.g. LOFAR, LOPES, LORUN, ITS, etc.
Filesize	☒	Int	Size (number of samples) of the file(s).
dDate	☒	Double	Like “Date” but with sub-second precision. Either time when the first sample was taken, or time when sample with `delay==0` was taken.
presync	☒	Int	Number of samples taken before the trigger (currently LOPES and LORUN only)
TL	☒	Int	LOPES/KASCADE style timelabel (number of 5MHz ticks inside the second).
LTL	☒	Int	LOPES-time-label (number of 40MHz ticks inside the second).
EventClass	☒	Int	Class of the event (1=cosmic ray event, 2=simulation event, 3=test, 4=solar)
SampleFreq	☒	uchar	Sample frequency in MHz
StartSample	☒	uInt	Number of the first sample in this second

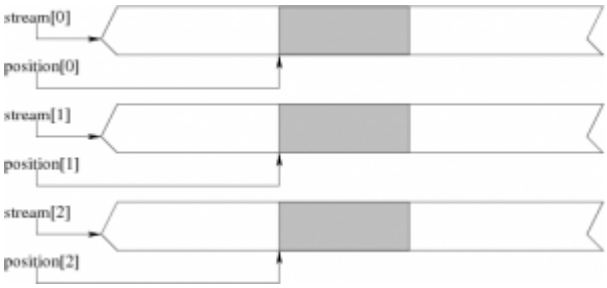
(* Mandatory)

The field-names are case sensitive, and should be put into the record exactly as they are written here.

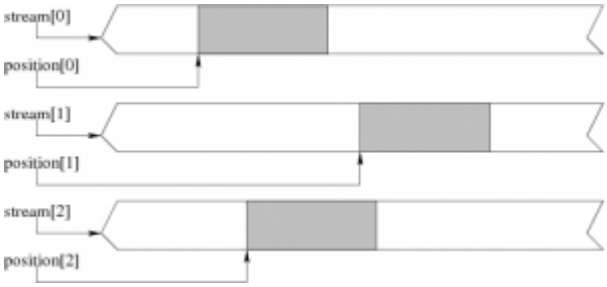
Filestream positions

The DataReader handles progression through the data volume via a set of DataIterator objects, providing *N* positions for *N* data streams. These stream- and position pointers allow a variety of access schemes:

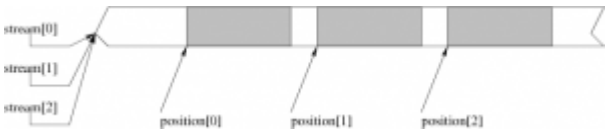
- Access to the same segment in multiple streams, e.g. when reading raw data recorded with the *LOFAR ITS*.



- Access to different segments in multiple streams

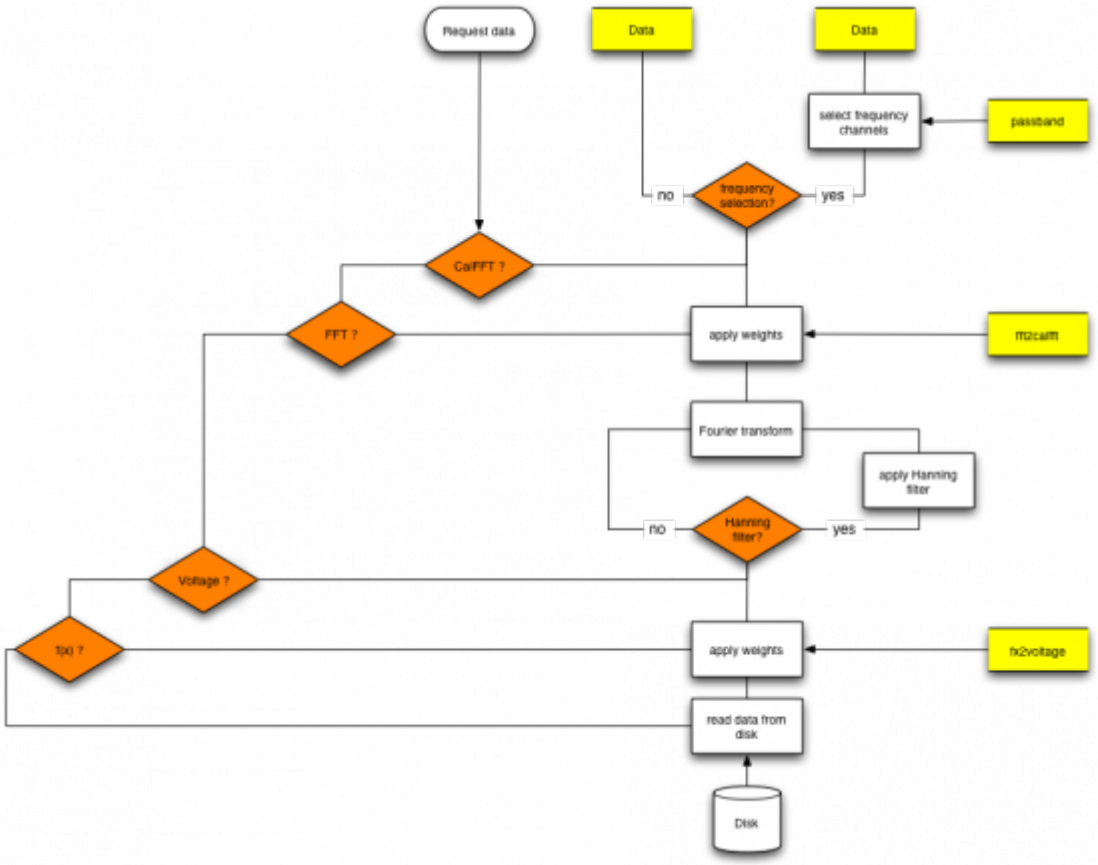


- Access to different segments withhin a single stream, e.g. when reading data from a LopesEvent file.



Data flow

The figure below illustrates the data flow inside the DataReader:



There is the option to insert a Hanning Filter step before performing the Fourier transform; this can be used to reduce the sidelobes in the frequency domain, originating from cutting out a block of data (which is equivalent to the multiplication of the data with a box function).

Performance

A clear trend can be seen when going towards smaller blocksizes, by which data are read from disk. One possible approach for tuning the performance would be read multiple blocks from disk and then dispatch them subsequently to the requesting routine; this of course requires some intelligence to be build into the data reading code, in order to do the bookkeeping.

Development

Adding a new data format

The DataReader framework has been set up in such a way, that adding the capability do read in data from new data formats should be kept as simple as possible:

- `DataReader` works as base class, from which all data type specific classes are inherited; by this the internal data processing framework is kept.
- Only reimplement the function performing the actual input from the data file, returning a standard product to the internal pipeline.

At the present time, the following classes are part of the data input framework:

Example

1. In the *header file* of the new class (here: `ITSBeam.h`) define a private variable `datatype_p` which is of the type as which the data are stored in the data file.

```
class ITSBeam : public DataReader {

  //! Information contained in experiment.meta are stored in their own
object
  ITSMetadata metadata_p;

  //! Type as which the data are stored in the data file
  float datatype_p;

 public:

  //! Get the raw time series after ADC conversion
  Matrix<Float> fx ();

 protected:

  //! Connect the data streams used for reading in the data
  Bool setStreams ();

};
```

   The two methods/functions are reimplemented from the `DataReader` class; a detailed description is given below.

2. In the *implementation file* (here: `ITSBeam.cc`) we need to reimplement two functions, which are already defined as virtual functions in the `DataReader` class:
   1. `setStreams()` – connect the data streams used for reading in the data from disk

```
Bool ITSBeam::setStreams ()
{
  bool status (true);

  uint blocksize (blocksize_p);
  Vector<uint> antennas (metadata_p.antennas());
  Vector<Float> adc2voltage (DataReader::adc2voltage());
  Matrix<Complex> fft2calfft (DataReader::fft2calfft());
  Vector<String> filenames (metadata_p.datafiles(true));
  DataIterator *iterator;

  /*
    Configure the DataIterator objects: for ITSBeam data, the
values are
    stored as short integer without any header information
```

```
preceeding the
    data within the data file.
  */

  uint nofStreams (filenames.nelements());
  iterator = new DataIterator[nofStreams];

  for (uint file (0); file<nofStreams; file++) {
    // data are stored as short integer
    iterator[file].setStepWidth(sizeof(datatype_p));
    // no header preceeding data
    iterator[file].setDataStart(0);
  }

  /*
    Setup of the conversion arrays
  */

  uint nofAntennas (antennas.nelements());
  uint fftLength (blocksize/2+1);
  IPosition shape (fft2calfft.shape());

  if (adc2voltage.nelements() != nofAntennas) {
    double weight (adc2voltage(0));
    adc2voltage.resize (nofAntennas);
    adc2voltage = weight;
  }

  if (uint(shape(0)) != fftLength ||
      uint(shape(1)) != nofAntennas) {
    fft2calfft.resize (fftLength,nofAntennas);
    fft2calfft = 1.0;
  }

  // -- call to DataReader::init(...) ----------------------------
--------------

  DataReader::init (blocksize,
            antennas,
            adc2voltage,
            fft2calfft,
            filenames,
            iterator);
  DataReader::setNyquistZone (1);

  return status;
}
```

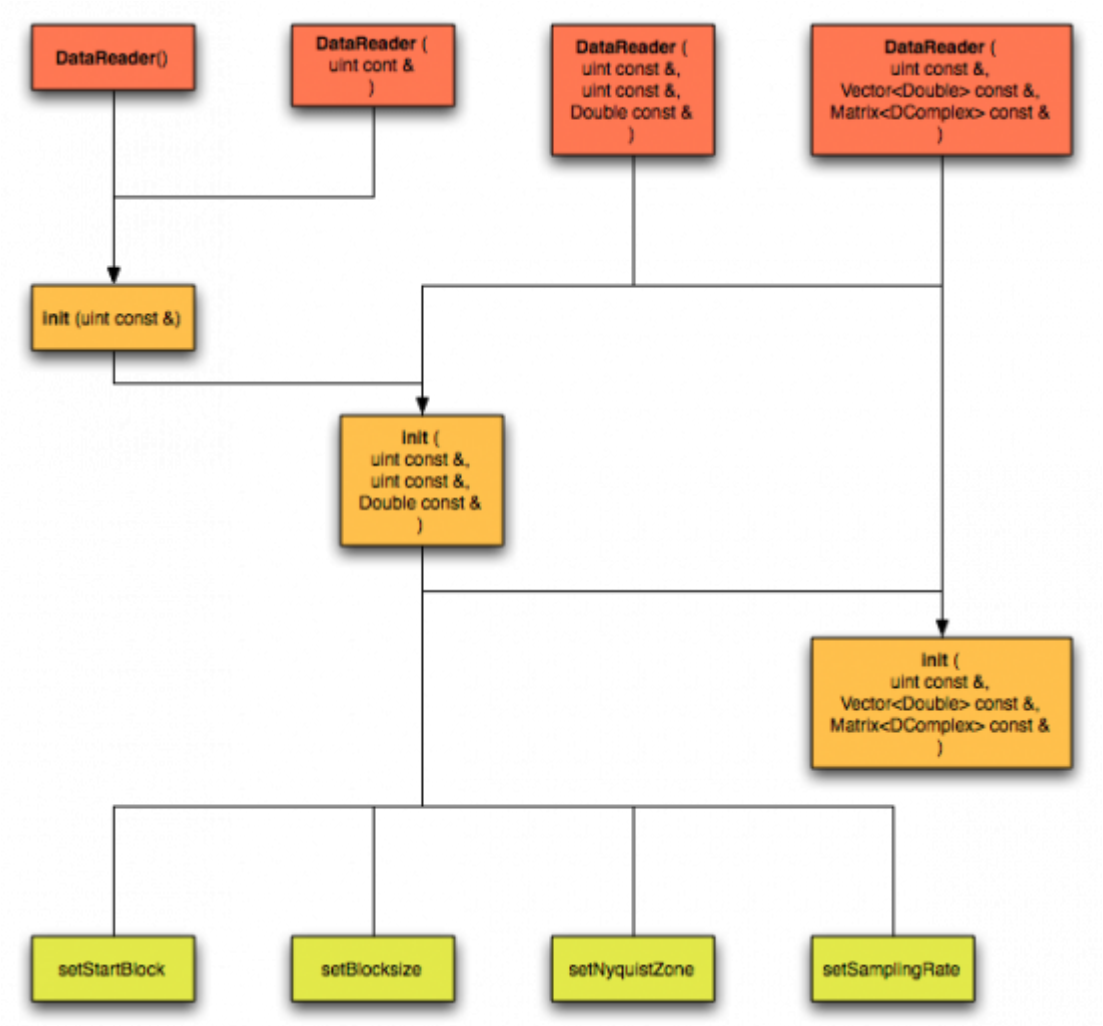
Even though the setup of the `DataIterator` can be quite different for your specific data format, the single-most important instruction – that needs to be issued before calling `DataReader::init` – is

```
iterator[file].setStepWidth(sizeof(datatype_p));
```

which takes care of adjusting the width by which the stepping through the data volume is done. Once all parameters have been set up correctly, they are passed to the base class.

2. `fx()` – Reading in of the data and formatting to one of the standard products within the processing chain internal to the `DataReader`; keep in mind that your data may be something else but the raw time series after ADC, such that you will need to re-implement another method (e.g. `fft()`).

Internal data initialization



Usage

... with C/C++ code

Usage within an application

```
#include <lopes/IO/DataReader.h>
#include <lopes/Data/LopesEvent.h>

DataReader *dr;
LopesEvent *le = new LopesEvent (eventfile,      // location the LopesEvent
file
                                 blocksize,      // nof. samples per block
of data
                                 adc2voltage,    // conversion weights: ADC
values to voltages [optional]
                                 fft2calfft);    // calibration weights: raw
to calibrated FFT [optional]
dr = le;

// processing of the data
for (int block(0); block<nofBlocks; block++) {
  fft = dr->fft();
}
```

With the conversion arrays optional you can even use the simpler construction method

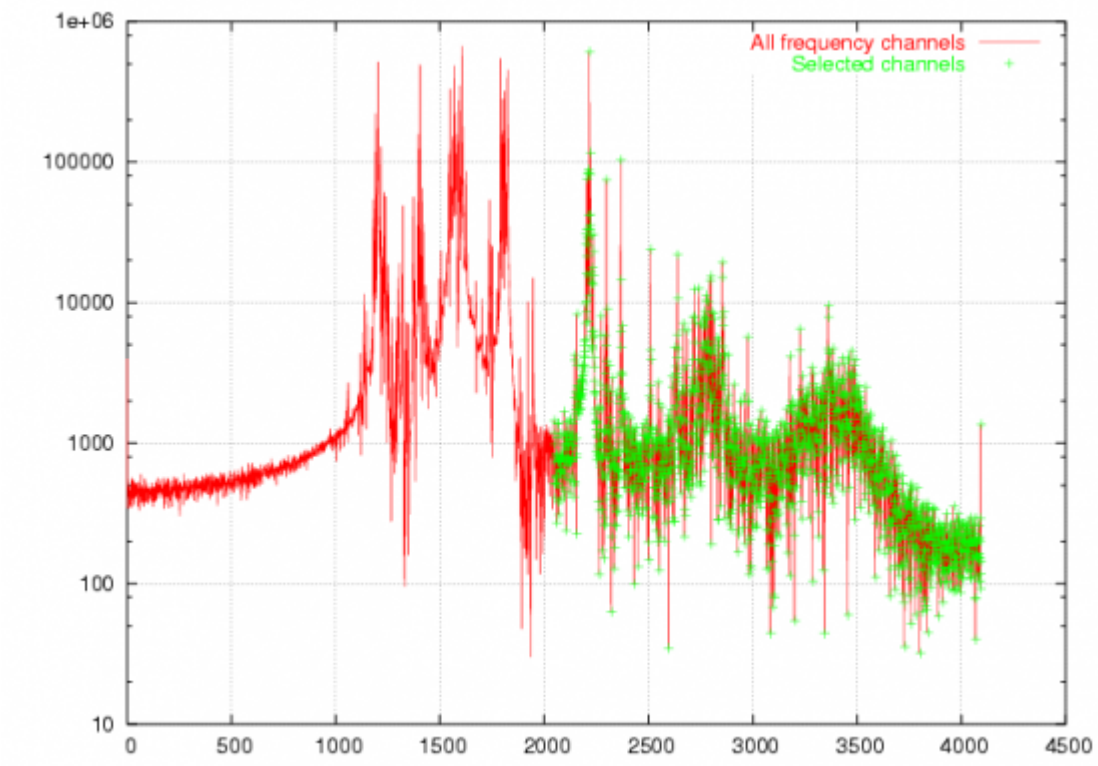
```
LopesEvent *le = new LopesEvent (eventfile,      // location the LopesEvent
file
                                 blocksize)      // nof. samples per block
of data
```

Data selection

Selection of frequency channels and antennas can be performed directly within the DataReader, e.g.

```
    dr->setSelectedChannels (selection);
```

where `selection` is an array of boolians of length `fftLength`.



← User Software • CR-Tools