Reconstruction framework user guide

Guy Nir

March 30, 2016

1 Introduction

2 Installation

3 Quick start

This section covers the basic operating modes for the reconstruction framework. We encourage new users to open a ROOT session and try to keep up. We will go over some basic examples, and refer the user to more advanced options in later sections.

3.1 Getting events into the framework

To start an analysis session first load the reconstruction library:

```
>> gSystem->Load("$RECO/libReco.so");
```

Where `$RECO` points to the location of the reconstruction folder. If everything is properly installed it should load all the classes needed. It is best to begin the session with:

```
>> Reconstruction reco;
```

This generates a `StationGeomery` and `OpticalIce` objects, and ties the relevant static pointers that need to know about them[1].

There are other functions to reco class, but right now they are not working...

Then there are three input methods:

- Built-in simple `EventGenerator`.
- Reading files from AraSim using `AraSimFileReader`.
- Reading data files from AraRoot framework using `L0FileReader`.

To generate simple events for the most basic testing (and if you want to run lots of parameters and want to generate millions of events on the fly), start up an object:

```
>> EventGenerator gen;
```

The generator is created with default parameters and generates an event on startup. If you want to change some settings look up Section 4.2. The basic command to include at the start of the loop is:

```
>> gen.generateEvent();
```

The generator will happily produce endless events for testing. You can play around with the position range for the vertex, the amplitude of the waveforms (at the source or antennas), the number of antennas receiving signal, and the polarization.

[1] This includes `Channel::setGeomety()` and `CurvedRay::setOpticalIce()`.

To get the waveforms into the framework, the basic tool is `Channel` objects. Each one holds a waveform, timing data, and the position of the antennas. For more detail see Section 4.4. Try

```
>> ChannelCollection channels=gen.getChannelCollection();
```

This gives you a `ChannelCollection` object, that is little more than a `vector<Channel>` object. This object holds the full input of the detector or simulation, which is relevant for reconstruction.

Another way to get events is to read ROOT files generated by AraSim or by AraRoot from real data. The two classes that are required for this are `AraSimFileReader` and `L0FileReader`, that work very similarly.

```
>> vector<string> filenames; filenames.append("AraOut.root");

>> AraSimFileReader reader(filenames);
```

or alternatively

```
>> L0FileReader reader(filenames);
```

Either way it is possible to retrieve the waveforms as before:

```
>> ChannelCollection channels=reader.getChannelCollection();
```

It is also useful to get the geometry saved in AraSim or AraRoot instead of the default station setup in `Reconstruction`:

```
>> reco.setStationGeometry(reader->getStationGeometry());
```

This also assigns the relevant static pointer. To get the next event from file, keep using

```
>> reader.getNextEvent();
```

That returns `true` so long as there are new events to process. This function may be placed in a breaking statement, or inside a `while` header.

3.2 Calculating timing data

Waveforms are primarily used to determine timing data, and the `Channel` objects have several functions that calculate it. The user is never exposed to the actual timings, without an implicit call to

```
>> channels.printout();
```

In analyis, the channels are given any command to find the timing results, and the relevant classes are called internally. All results are saved in the channels, that are given as a whole to the next phase of analyis.

There are three output types for time finding:

- Hit times. The point in time each waveform reaches its peak.
- Time delays. The difference between each pair in the collection.
- Correlation graphs. The relative strength of the wavefroms multiplied and summed, for different time offsets between each pair of channels.

For a system with 16 channels, the hit times will include 16 timing results. For time delays, there are 120 independent pairs of timing results. For correlation graphs, there are 120 graphs, each containing 2048 (interpolated) points, the position of the peak of these graphs is equivalent to the time delays.

There are several ways to calculate each of these timing results. For example, you could do

```
>> channels.applyFinder(TimeFinder::GAUS);
```

Which smooths the waveform and finds its peak. Other methods are discussed further in Section 4.4. The results of any call to `applyFinder` are kept inside the channels and are lazy loaded when called again.

If you want to produce the time delays from the hit times, a simple class exists that calculates all the pairwise subtractions:

```
>> channels.applyDeltaFinder(DeltaFinder::SUBTRACT);
```

Another option is to do the slightly slower

```
>> channels.applyDeltaFinder(DeltaFinder::CROS);
```

That calculates the correlation graphs for each pair. The results of calls to `applyDeltaFinder` are also kept in the channels, and are lazy loaded when called again. This method also generates the full graphs, that are kept as `CorrGraph` objects for each pair.

The results that can be used by the following analyis are always those that were implemented last. In the examples above, the hit time are still given by the results of the call to `GausTimeFinder`, while the time delays are given by the results from `CrosDeltaFinder` rather than the earlier call to `SubtractDeltaFinder`. If you want to swap them, just re-apply the previous finders, they are lazy loaded, so once they are called, all subsequent calls are fast.

3.3 Finding the vertex position

Getting vertex localization is the more complicated part. The easiest way to keep track of the different levels of analyis is to generate several instances of the subclasses of `VertexFinder` and work with them as inputs and outputs.

First off generate several different types of finders:

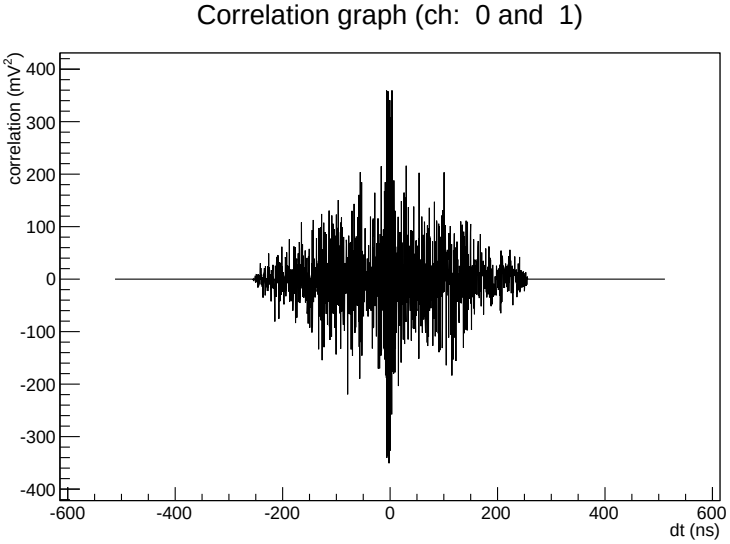

Figure 1: Example correlation graph from `CorrGraph`, produced from simulations of `EventGenerator`. In this graph the absolute maximum power is at $dt = 3.5\,\mathrm{ns}$, which is also the result of the `CrosDeltaFinder` object.

```
>> EarliestVertexFinder start_finder;
```

This finder puts the starting position at the position of the antenna that was hit first (hence, earliest). Vertex finders look for maximum likelihood for each point in space, and they usually require a good starting position. Some more practical finders can be initialized:

```
>> IntMapVertexFinder intmat_finder(0,0,0,0,1,36,72);
```

or

```
>> MCMCVertexFinder mcmc_finder(0,0,0,0,1e5,100)
```

Where the different parameters specify general settings like coordinate system, hit times or time delay inputs, parameter locks or more specific input parameters (see discussion in Section 3.3).

Once all these finders are initialized, they can be used multiple times for different events (or even for the same event with other starting positions). The way to activate them is to call

```
>> start_finder.findVertex(channels);
```

So the first finder looks at the results from the `channels` object, and puts the vertex position at the earliest antenna. The next finder on the list can be called

based on the results from the first finder

```
>> intmap_finder.findVertex(channels, start_finder);
```

Along with the starting position, the previous finder also gives the new finder some more restrictive error bounds in which to look for the vertex position. You can even call the same finder several times with itself as reference, so that the search narrows down to smaller intervals with better resolution.

Some finders don't need starting positions, since they scan the whole space anyway. It is still useful to give them another finder, to get updated error bounds and real positions (see below). For example:

```
>> mcmc_finder.findVertex(channels, intmap_finder);
```

To access the vertex results you can use

```
>> mcmc_finder.printout();
```

All classes in the framework have this method, to make interactive work easier. If you need to pass this information around use,

```
>> intmap_finder.getVertexPos();
```

Which returns a pointer to a `VertexPos` that is a subclass of `Pos` class that holds both x, y, z and R, θ, ϕ. The advanced `VertexPos` class also has errors on these sizes, the previous position it was given, and the true position, if known.

If the input source is simulation (either from generator or AraSim), or if the data is known to be cal-pulser events, the true position of the vertex can be given to the first finder in the sequence using

```
>> start_finder.setRealPosition(gen.getSourcePosition());
```

The position is set through a `Pos` object, or as x, y, z coordinates. Once the first finder knowns the true position, it passes it to all other finders when they are linked together. This lets you access the "residuals" for each finder result, that are just an easy way to check the distance between the true position and the found position.

3.4 Plotting and comparing results

Soon to be updated with the finalization of `THX` class.

3.5 Saving results

To write analysis results to root files for further work, we provide the `TreeWriter` class. A single object is given the relevat objects, i.e. the channels and vertex

finders:

```
>> TreeWrite writer("filename.root");
```

```
>> writer.addChannels(channels);
```

Followed by adding any or all finders:

```
>> writer.addFinder(intmap_finder);
```

```
>> writer.addFinder(mcmc_finder);
```

These additions are done within each loop, followed by a command to fill the `TTree` with the relevant branches:

```
>> writer.fill();
```

Note that by default the writer clears the inputs after filling the tree, unless the user sets `writer.setAutoclear(0);` in which case the user must call `writer.clearAll();` at the end of each loop.

Similarly to the commands in ROOT, when the loop is done, a call is made to

```
>> writer.write();
```

to transfer the data from the tree to the open root file.

3.6 List of time-finders

Time finders divide into two distinct groups: hit-time finders and delay-finders. Hit times will calculate the absolute timing (relative to the WF's time axis), generally by finding the time of the peak of the WF. Delay finders will provide relative timing between channels, e.g. the cross correlations. Since reconstruction can be made based on either approach both these "time inputs" can be used for vertex finding.

- `TimeFinder` is an abstract base class for all hit-time finders.
- `SimpleTimeFinder` just uses the maximum of the WF.
- `ThreshTimeFinder` finds when the WF goes over a threshold as the hit time, and calculates the error based on the number of bins above threshold.
- `GausTimeFinder` uses a gaussian smoothing before finding the peak (match filter).
- `CSWTimeFinder` uses coherently summed wave to get something like cross-correlation hit times.
- `RealTimeFinder` used for simulations when the real hit times are know. Can give true time positions plus random error, for debugging purposes.

- `DeltaFinder` inherits from `TimeFinder` and is an abstract base class for the pair-wise delay finders.
- `CrosDeltaFinder` calculates the correlation graphs and finds the peaks to calculate the delays. This is (in most cases) the best method for finding the timing data.
- `SubtractDeltaFinder` takes the hit-times and cros-subtracts them to produce delays. Only useful when delays are needed but no real delta-finder has been called.

In addition, each pair of `Channel` objects can have a `CorrGraph` object calculated, which holds the full correlation plot for that pair. This is useful e.g. for building the interference maps.

All finders can be called using `ChannelCollection::applyFinder` or `ChannelCollection::applyDeltaFinder`. Each time a finder is called it is lazy loaded (i.e. only calculate if the result has not been calculated and saved before). This includes the generation of `CorrGraph` objects, that are calculated when calling `CrosDeltaFinder` or `IntMapVertexFinder`. So a script can be written that applies a different finder before each step in the vertex-finding process, without needing to recalculate anything.

3.7 List of vertex-finders

The most important thing about vertex finding is to remember to give a starting position to the next vertex finder on the list, e.g.

$>>$ `finder`$_b$.$findVertex(channels, finder_a)$;

Some finders do not use a starting position at all, some will benefit from reducing the parameter space (based on the previous finder's confidence intervals), and some will fail miserably without a good starting guess.

- VertexPos inherits from Pos and holds the position and also errors for a vertex. All vertex finding results are contained in these objects, and initial positions should be given in this format.
- VertexFinder inherits from VertexPos and is an abstract base class for all vertex finders. Contains the code for basic operations such as calculating χ^2, deciding on timing parameter and coordinates etc.
- ScanVertexFinder simply scans over all parameters and calculates the χ^2 for each point. This is the slowest and least accurate method, but also the most reliable. It is useful for sanity checks, for scanning a small region in space or when locking one or two parameters.

- MinuitVertexFinder uses minimization techniques to find the χ^2 minimum. This method will converge quickly but will need a very good starting guess otherwise it will get stuck at a local minimum.
- AnalyticVertexFinder uses the Analytical Sphere Method that takes four antennas and calculates the solution to the four timing equations. This is very quick but can only use hit times and simple ice models.
- IntMapVertexFinder generates a few interference maps at different radii and finds the highest correlation value. At high resolution this method is very slow but extremely accurate (in and ϕ at least).
- MCMCVertexFinder takes a random walk around the position space based on the Metropolis-Hastings algorithm, giving a 3D map of the probability distribution for the vertex position. This method is very slow, but is very reliable and gives a good view of the degeneracy, local minima and uncertainties of the probability distribution for each event.
- EarliestVertexFinder puts the vertex at the position of the antenna with the earliest hit time. Useful to give as starting position for other finders. Can only get hit-times as inputs.
- AnyWhereVertexFinder puts the vertex at any location (chosen by the user). Useful for debugging.
- RealVertexFinder puts the vertex at the real position (simulation or calpulser only), optionally adding errors around that point. Useful for debugging.

3.8 Vertex finders parameters

Vertex finders have several parameters that can be defined for each finder, that allow the user to implement very different results based on the same core algorithms. This list includes the basic parameters for most finders, though some finders will happily ignore incompatible parameters (perhaps giving a warning).

These parameters can be passed via constructor (old method soon to be removed) or via a simple text parser.

- coordinates: choose CARTESIAN or 0 to use x, y, z. Choose SPHERICAL or 1 for $R, , \phi$.
- times or inputs: Choose TIMES or 0 to reconstruct based on hit-times. Choose PAIRS or 1 to use pairwise time delays.

- locked or unlocked: and specify X or Y or Z, or 0,1,2 respectively to lock or unlock a parameter. In any coordinate choice you can also input R or THETA or PHI.

This list will updated as more options are added. Also note that each finder will have additional parameters that can be set for their specific needs, e.g. AnyWhereVertexFinder will need three coordinates to be supplied by the user.

3.9 Vertex finding notes

changing coordinates and error intervals. locking parameters. initial positions. lazy loading time finders internally.

3.10 Vertex finding plotting tools: THX

4 Advanced users

note: this section is out-dated and needs serious review.

4.1 Module Overveiw

Following is a top-down description of the data and command flow, which is based on manager classes (e.g Reconstruction), data classes (e.g. Channel), worker classes (e.g. VertexFinder) and container classes that unite several data objects (e.g. EventData). These distinctions are not absolute. Some of the data classes also do work when required, while some containers also hold information in addition to their contained objects.

Top level classes

The top-most level for analysis is Reconstruction class. This class is given the files to be read, and the output location to write to. The user can supply any objects needed for reconstruction or allow them to be generated by Reconstruction. Analysis parameters can be given to Reconstruction either as function calls (e.g. useCartesian()) or through the Parser class.

Reconstruction is a *persistent* object, that is created once and does not need to be changed during an analysis of many events or runs. It contains several other *persistent* classes:

- L0FileReader or AraSimFileReader or EventGenerator
- OpticalIce and StationGeomery
- Output

The Reconstruction object currently only initializes the required objects and points the right static pointers to them. Future work will give it more functionality.

InputOutput module

The input classes, L0FileReader and AraSimFileReader are used in a similar manner, they are given a file name (or a list of files) and they parse the root files and extract the needed information as objects within the framework. EventGenerator can be used instead, to generate simple events without needing any input files.

The OpticalIce and StationGeomery classes are used to contain the environmental data used in the analysis. OpticalIce determines the ice model used, ice curvature etc. StationGeomery is a container for AntPos objects that store the positions and properties of the antennas in the station.

```
The Output class is used to store the location of the output file,
and different switches for what portions of the data would be saved
to file.  It can also output root files as class hierarchy or as simple
data members.  It can be used to write debugging data and basic reconstruction
results as text files.
```

Geometry module

TimeFinding module

VertexFinding module

4.2 Simulation basics

Using EventGenerator

Using StationGeomery

Using Pos

Using OpticalIce

Using CurvedRay

4.3 ROOT files input/output

Using AraSimFileReader

Using L0FileReader

Using Output

Using EventData

4.4 Time finding basics

Using Channel etc.

Using Timing etc.

Using TimeFinder etc.

Using DeltaFinder etc.

Using CorrGraph

4.5 Vertex finding basics

Using VertexPos

Using VertexFinder etc.

Using THX

5 Class reference

6 Improving the framework

In the following sections we present some coding standards and conventions that should be maintained when adding code to the framework.

6.1 New TimeFinder subclasses

When implementing new subclasses of TimeFinder it is important to implement a copy constructor that will call the base class' copy constructor:

>> SubTimeFinder::SubTimeFinder(const SubTimeFinder other) : TimeFinder(other) {...}

Make sure to copy in the body any new members added in this class.

If you have any such new data members, it is important to initialize them in a dedicated call to void initialize() and also add an implementation of operator=() that calls TimeFinder::operator=(other) internally, so that TimeFinder's members are all copied over.

The other critical point is to implement two (otherwise abstract) functions, the first:

codeTimeFinder *SubTimeFinder::getDeepCopy(){ return new SubTimeFinder(*this); }

That calls the copy constructor of the subclass but outputs a pointer to the base class. This allows a copying of polymorphized vectors (such as vector<TimeFinder*> inside Channel);

The second function is

virtual void SubTimeFinder::calculateTime()

That does the actual work, and is called whenever the finder is applied.

7 Committing code

8 Benchmarks