# Processing Contexts for Experimental HEP Data

Marc Paterno, Chris Green (*SCD/SSA/SSI/TAC*)
FERMILAB-TM-2647-CD

## Contents



## 1 Purpose of this document

This document provides, for those not closely associated with the experimental High Energy Physics (HEP) community, an introduction to data input and output requirements for a variety of data processing tasks. Examples in it are drawn from the *art* event processing framework, and from experiments and projects using *art*, most notably the LArSoft and NuTools projects.

## 2 Introduction

This document describes several different event-data processing contexts that are common in experimental HEP. We are describing the most important contexts for event-data processing in order to make sure we enumerate all the important access patterns for our data. The usage patterns described are those currently seen in HEP, using the commodity cluster programming model (grid computing) that dominates the field. An important feature of this model is the breaking up of large workflow tasks into separate event-processing framework program jobs, and the use of files to communicate data between these steps in the larger workflow. The path to exascale usage may drive us to combining several, maybe all but the final analysis, steps into a single program execution.

We characterize the usage patterns based on the following criteria:

1. The typical number of modules in the workflow. Here, a *module* means one of the modular elements common in HEP event-processing frameworks. A module

typically performs one well-defined task, and encapsulates an atomic unit of work. We distinguish between *processing modules*, which do physics tasks, and *output modules*, which write data to storage.

2. The computational intensity of the task. This is a very rough estimate of the ratio of time spent in processing modules, compared to the time spent in output modules. Tasks of "high" computational intensity may spend spend 50 times as much time in computation as input and output; tasks of "low" computational intensity can be 1:1 or lower.
3. The number and complexity of the event-data products involved in the processing. Simple data products are essentially tabular in nature. They may be fixed-size arrays of scalar types, or structures composed of fixed numbers of scalars and fixed-size arrays. Complex data types include variable-size arrays, and nested containers of types that are themselves complex.
4. The pattern of access to data read into the program. What is the unit of processing—the event, or something smaller? Are many or few data products in the event read? For those data products that are read, is the entire product read or only a portions of the product?

# 3 Simulation

Simulation processing is compute-intensive, involving detailed simulation of a detector's response to Monte Carlo (MC) "events". These processes often do no reading of event data. They typically write large volumes of both simple and complex products. These products include both descriptions of "MC truth", which means the knowledge of the particles and interactions of the simulation, as well as the final product of simulated raw data of the detector.

Simulation tasks often include relatively few processing modules. They may be organized into more than one processing step. One common organization is to separate the process that simulates analog energy or ionization deposits in the detector from the step that determines what the (digitized) response of the detector electronics to such deposits would be.

These processes may read large ancillary files of non-event data, such as neutrino flux files used by event generators, tables of data needed by GEANT simulations of detector response, and descriptions of detector geometry. These ancillary files may be read (and written) using different I/O libraries than are the experiment's event data.

Simulation processes access data by run, subrun, and event, processing each event atomically; events are mutually independent. Runs and subruns may be highly artificial in simulation processing, since there is no physical detector for which relevant conditions are recorded. Changing detector conditions are rarely (but not never) simulated. Runs and subruns may be used for other bookkeeping purposes, such as to identify the class of simulation represented by a given run.

## Output data product examples

An example simple product is the information used for GENIE-based event re-weighting, `std::vector<simb::GTruth>`. Another is the detector raw data, for example `std::vector<raw::RawDigit>`. In the class `raw::RawDigit` it is important to note that, while the class has a data member that is a `std::vector`, that the vector is always of fixed length, for all waveforms in a given data product.

An example complex data product is the `std::vector<MCTruth>`. A UML diagram describing this class is shown in figure 1. This class contains MC truth information about the (possibly many) sources of primary particles in a simulated event, including the interaction types and the particles that are part of the interaction, including both stable and unstable particles.

Each event tends to be large (1–10 MB/event).

## Simulation with mixing

In some simulation jobs, ancillary event data files are read, to achieve what is called "mixing". The event data read from these ancillary files are similar to the event data read by reconstruction jobs. They have the notable difference in that it is sometimes necessary to read the data for many (tens, hundreds or thousands) of ancillary events in the processing of one primary event, and that the choice of what events to read is often randomized, thus making random access within the ancillary file critical.

# 4 Raw reconstruction

Reconstruction is compute-intensive: the detector's raw data are processed to produce quantities important for physics analysis. The entirety of the available raw data for each event is read as input; if available, simulation data may also be read for correlation and associated with reconstructed items.

Uncompressed and non zero-subtracted data are generally relatively simple in structure, but can be huge. Zero-subtraction and compression can reduce data size by 2 orders of magnitude for an experiment like DUNE.

Raw reconstruction tasks typically include many (tens to hundreds) of processing modules. Each module typically creates one, or a few, different types of data products.

The raw data are typically processed in conjunction with other sources of information such as calibration data, detector geometry information, or beam luminosity information, which data may be imported from different sources such as external databases or flat text or binary data files. Generally all raw data are used, along with relevant associated simulation data where applicable.

New output product types are invented frequently by reconstruction algorithm authors. Existing types change frequently, and so schema evolution is essential.

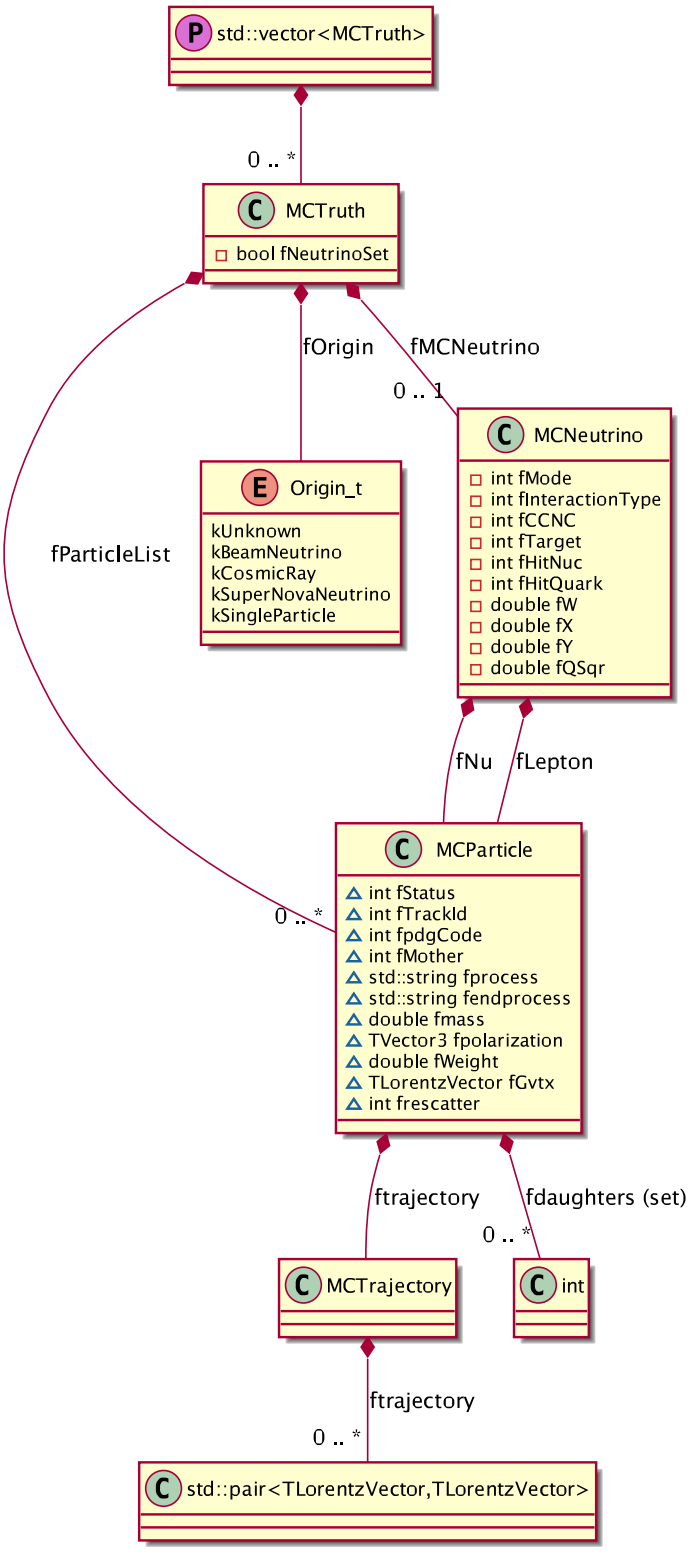


Figure 1: `std::vector<simb::MCTruth>` is a typical complex data product, created in LArSoft simulation programs. It carries the description of the (possibly many) source(s) of particles in a single detector readout. Boxes marked with a circled 'P' represent top-level event-data products; those marked with a circled 'E' represent enumeration types, and those marked with a circled 'C' represent other classes or types.

Data are read on an event-by-event basis. While not every data product in each event is read, most products are read. Whenever a data product is read, it is necessary to read the entire product.

### Input data product examples

- Simulation output (*e.g.* `std::vector<simb::GTruth>`, `std::vector<simb::MCTruth>`).
- Raw data (`std::vector<raw::RawDigit>`).

### Output data product examples

- Reconstructed intermediate objects: calibrated wire information (`std::vector<recob::Wire>`), tracker hits, reconstructed calorimeter deposits.
- Reconstructed physics objects: tracks (`std::vector<recob::Track>`), clusters (`std::vector<recob::Cluster>`).
- Many types of associations, critical to understanding physics objects (*e.g.* `art::Assns<raw::RawDigit, recob::Hit>`, `art::Assns<recob::Cluster, recob::Vertex, unsigned short>`, `art::Assns<recob::Hit, recob::Track, recob::TrackHitMeta>`).

## 5 Second pass reconstruction

Second (and later) pass reconstruction is done when improved calibrations, or new algorithms, or improvements in previous algorithms, are available. Such reconstruction passes are typically compute intensive; they may be less so than first-pass reconstruction in specific cases, *e.g.*, if only a small fraction of the first-pass reconstruction work is being replaced by newer work. Typically tens to hundreds of processing modules are used.

Second pass reconstruction differs from raw reconstruction in that it is common to *not* read all of the input event. In particular, those data products in the input file that are to be replaced by improved versions (for whatever reason) do not need to be read.

Products that are read, are read in their entirety, on an event-by-event basis.

### Output data product examples

The data products output by second (and later) pass reconstruction are the same as those output by first pass reconstruction.

## 6 Ntuple-making

Ntuple-making starts from full reconstruction output and produces summary objects suitable for final, typically interactive, analysis. It is not computing intensive. Complex

products are read from reconstruction output and simple products are written by up to a handful of writers (usually one module per analysis ntuple format being supported).

## 6.1 Input data product examples

Typical input data products include selected parts of most simulation, reconstruction output (not raw data), all events.

## 6.2 Output data product examples

Ntuple-making code does not produce event-data products. Instead, it produces relatively simple tabular data. These data are mostly a subset of input data, with limited production of derived quantities by additional algorithms.

Types are fixed-size; different "tables" may be required to express the variability of some entities (*e.g.* electron candidates, muon candidates) and a table will span the whole data set, not just one event.

Examples include reconstructed and MC particle IDs, four-vectors and vertex information per event.

# 7 Ntuple analysis

Analysis of produced ntuples is generally an individual, often interactive exercise and is typically not computing intensive. Only the simple ntuple data types are read and none are written. Data may be imported into data analysis systems such as ROOT, Pandas or R.

Processing need not be event-oriented, and is often not. Processing is frequently oriented toward independent processing of physics objects, *e.g.*, tracks or vertices. Frequently only a few of the data members of a physics object are needed for a particular task. This processing is typically not done in an event-processing framework, so there is generally no concept of processing modules.

Accidental features of a common use in the prevalent storage format (ROOT "trees", instances of class `TTree`) sometimes make a task (such as looping over all reconstructed particles hypotheses of a given type) appear to be event-oriented. The result is that code is more complex than necessary; doubly-nested loops over events-then-particles would be used where a more task-oriented format would result in a direct loop over particles. Frequently, application of selection criteria ("cuts") may direct which rows need to be read.

Ntuple analysis tasks do not write out framework-format files.

# 8 Summary

Most of the major event-processing contexts in HEP read data on an event-by-event basis, and require the reading and reconstitution of complete high-level (C++) objects. These

contexts include the most computation-intensive processing in the current paradigm of C++-object based simulation and reconstruction tasks.

Only the final “analysis ntuple” task typically consumes data on a non-event basis, and does not require the reconstitution of (C++) objects.