Common format

Summary of discussion

We have identified the goal:

- ❑ **Motivation** A common output format will facilitate common benchmark and validation tools.
- ❑ **Disclaimer** The idea is not to force the same output for any of the tools; the idea is to be able to create an output in addition to the existing output. So we will not break anything and should be easy to implement for everyone.
- ❑ **Proposal** We propose to use a flat ROOT file, i.e., a ROOT file that can be edited also with uproot. Please see also an earlier discussion where Torre summarizes the ATLAS experience in using flatter and simpler interfaces.

This is the status of our discussion of the last weeks:

- Andrea has put a simple (starting) tree for DIS together.
- Miguel has pointed us to the Delphes output format.
- Sylvester and Whit share a YAML file with variable definition.

Next steps

Andrea, Dmitry, Markus, Sylvester, and Whit have volunteered to come up with a first draft for the common output format the rest of our working group can comment on August 5.

Naming conventions

1. All fields of a root file should be named using snake_case with all lower case letters.

   Rationale: Many people that get used to one pattern are pretty unhappy to work in environments where the opposite pattern is adopted. Scientific studies show (1, 2, 3) that humans perceive words written in snake_case faster than with CamelCase, especially as a number of stacked words increases. Still there should be some convention selected.

   Examples:
   ```
   evt_weight          # good
   evt_prt_count       # good
   evtMyValue          # bad - camel case
   evt_W2              # bad - a capital letter
   ```

2. The name should fully describe the context. When a field name consists of many words (e.g. “Particle momentum x”), the most general context goes left.

   Explanation: If one has “Particle momentum x”, the “Particle” is the most general context as particles have many properties and “x” is the most specific, thus the name should be **prt_p_x**.

   Rationale: When fields are named this way, a simple alphabetical sorting allows one to also sort out fields by context. It greatly improves reading, file introspection and also helps with autocompletion in editors, etc.

   In the root library one can request reading data with wildcards. E.g `evt_*` - would mean: “read all fields whose names start with `evt_`” - reading all event level information for processing. Another good example would be `prt_[gen,reco]_p*` - process reconstructed and generated momentums -

   Examples
   ```
   prt_gen_p_x        # good
   prt_gen_x_vtx      # bad - should be ..._vtx_x
   p_x                # bad - not clear of what momentum it is
   ```

3. Common names:

- All event level information (i.e. once per event) starts with evt_*
- All information coming from event generator has *_gen_* in its name

Design considerations

CERN ROOT file layout.

Particles start with *prt_**. All types for particles are std::vector<type> or arrays. So it is vectors/arrays of fields rather than vectors of structures. Where each vector.size() corresponds to *evt_prt_count*

There are several possibilities how this can be realized. Pseudo root code looks like this:

```
tree->Branch( "evt_id",  &event_record.id, "evt_id/I");
tree->Branch( "evt_prt_count ",  &event_record.prt_count , "evt_prt_count/I");
...
tree->Branch("prt_pdg", event_record.pdg, "pdg[evt_prt_count]/L");
```

With std::vector<type> the code above will look different, but the idea is the same.

TBD

Metadata and schema evolution

For any public API is very important to have a proposed way of schema evolution and detection. Having metadata about what packages produced the data is also important

Schema versioning.

We will use semantic versioning with two numbers: <major>.<minor> versioning for tree and metadata.

Where:

- <minor> number changes when backward compatibility is NOT broken - old code can read new file and results of processing are the same
- <major> number changes when the compatibility is broken.

Example:

1. Initially we have a 1.00 version. After adding the “prt_pid_weight” field to the tree the version is changed to 1.01. Old software can read new files.
2. Imagine we have a 1.01 version. If we change the name from “prt_pid_weight” to “prt_total_pid_weight” the version is changed to 2.00. Old software can’t read new files

Metadata and run info

Metadata is written as a single record per file.
It is good to have some metadata of what was used on the way of data processing. I would imagine that it would be good to know:

- Processing software - maybe array. It would be good to know from file that originally data was generated by Beagle v… and processed with … and then with …
- Date-time of creation / last processing
- Description: whatever and whoever

Markus proposes the following information (we can add more fields later):

- Beam species
    - Electron or positron (for those who studying the possibility of positron beams at the EIC)
    - Ion beam species
- Beam energies
    - Electron beam energy
    - Ion beam energy (per nucleon)
- Beam polarization
    - Degree of electron beam polarization
    - Degree of ion beam polarization
    - Ion beam polarization: transverse or longitudinal
- Crossing angle
- Event generator
- Event generator version
- Fast or full simulation software
- Fast or full simulation software version

Redundancy

While we try to minimize the redundancy, some redundancy is allowed in favor of convenience.

Rationale: There are some extra values, like ..._p, ..._px, ..._py, ..._pz. It is done to simplify the processing of the resulting files with bare ROOT. At the same time, such values may be a source of difficult to find mistakes. Imagine if total momentum (p) in the example above doesn't correspond to px, py, pz. So the collective decision is to use such values but to try to minimize their number.

Ambiguosity

It is better to keep one ambiguous but clear value and put more effort into the documentation rather than have a lot of copying fields fitting all possible contexts.

Rationale: There might be ambiguities of how different sources can fit into one value. event_weight is a good example of such a problem, as there are different ways to normalize the event; prt_is_stable is another example as some generators can even provide several statuses to define "stable". The recommendation for such cases is to store the most appropriate value as event_weitgh/prt_is_stable, save generator data on event level and provide metadata specifying the generator. It would be then the task of documentation how to use the event weight of a given generator in the analysis. Otherwise the format would depend on a growing collection of generators

Original information

Original and reconstructed (or smeared) information is stored in the same tree. Naming and structure of ture/reconstructed values should be as close as possible.

prt_gen_… - fields means information that came from generator
prt_rec_… - fields means information that came from simulation/reconstruction

Rationale: It is a subject of many user requests to have an ability to compare original and smeared/reconstructed particle information without opening additional files (like reading generated file in addition to result file). The original information duplicates the resulting to some degree. There are several approaches of how to optimally organize such data, including using identical friend tries for generated and reconstructed information. The collective decision is to add fields from true and reconstructed/smeared data to one tree trying to keep their structure similar. There are several reasons for this: only a fraction of fields are being copied, single files are easier to manage in terms of processing and long term storage, since long term data conservation is one of the main priorities of this effort.

Comment: at this stage it might look that there are too many ..._gen_… fields. But imagine if the file format will grow including more detector details (recalling how Hera data looks), then those ..._gen_… fields will not be the majority and labeling of what comes from a generator will be especially important.

Values for missing fields

Double NaN-s for all uninitialized fields, -1 for integer indexes.

Rationale: Simulating the detector some particles may have only some sort of fields, e.g. gamma in a calorimeter that has only E. The decision is to use double not-a-number (NaN) for all not initialized/set double fields instead of having some special numbers like -9999 or 0. Before C++11 C++ had problems dealing with nans depending on compiler or realization and there are still inconsistencies about signaling NaNs. But we will use non-signaling nans which are consistent, cern root deals correctly with them when plotting the variables directly, and one can filter values by A!=A in the in-line analysis. In almost all other modern languages (python, go, java) NaNs are the first class citizens and their clarity and support is on the high level. If some values are missing from reconstruction/smearing double NaN is used.

Units

All values have consistent units: GeV, mm, radians

Event information

Field name	Type	Description
evt_id	uint64_t	Unique event ID in run aka “event number”
evt_weight	double	Event weight
evt_prt_count	uint64_t	Number of particles
evt_cross	double	Crossing angle in rad.
evt_ion_id	int64_t	Id of Ion beam particle in particles list
evt_ion_z	int64_t	Z of the target nucleus
evt_ion_a	int64_t	A of the target nucleus
evt_ion_p	double	Ion total momentum p/A

evt_lep_id	int64_t	Id of lepton beam particle in particles list
evt_lep_p	double	Lepton total momentum
evt_lep_pdg	int64_t	Beam lepton PDG code
evt_gen_recoil_id	int64_t	Recoil lepron id in particles (correspond to prt_id)
evt_gen_q2	double	Q2
evt_gen_x	double	x
evt_gen_y	double	y
evt_gen_w2	double	W2
evt_gen_nu	double	nu
evt_gen_t_hat	double	t^hat
evt_gen_phi_s	double	phi_S
evt_gen_vtx_x	double	Primary vertex information from generator
evt_gen_vtx_y	double	
evt_gen_vtx_z	double	
evt_gen_vtx_t	double	
evt_gen_ion_pol_x	double	Ion beam polarization generator information
evt_gen_ion_pol_y	double	
evt_gen_ion_pol_z	double	
evt_gen_lep_pol_x	double	Lepton beam polarization generator information
evt_gen_lep_pol_y	double	
evt_gen_lep_pol_z	double	
evt_rec_vtx_x	double	Smeared/reconstructed primary vertex information
evt_rec_vtx_y	double	
evt_rec_vtx_z	double	

evt_rec_vtx_t	double	

Missing: phi_S, polarizations

Particle information.

All starts with *prt_**. Now type, actually all types for particles are std::vector<type>. Where vector.size() corresponds to *evt_prt_count*

Field name	**Type**	**Description**
prt_id	uint64_t	Unique id inside event
prt_gen_pdg	int64_t	PDG value
prt_gen_charge	double	
prt_trk_id	uint64_t	ID of related track/hypo in reconstruction or full simulation
prt_gen_p	double	Full momentum
prt_gen_px	double	Momentum projections
prt_gen_py	double	
prt_gen_pz	double	
prt_gen_tot_e	double	Total energy
prt_gen_m	double	M for mass.. or murder… or mass mu...
prt_gen_z	double	Fractional hadron energy
prt_gen_phi_h	double	Azimuthal angle between lepton scattering plane and hadron production plane
prt_gen_time	double	
prt_is_stable	bool	It is a stable particle (redundant with prt_gen_code)
prt_is_recoil_e	bool	true if this particle is true recoil electron
prt_gen_code	int64_t	Generator particle code, also “status code”

We also need z and phi (w.r.t. to the virtual-boson direction).

Field name	**Type**	**Description**
prt_mother_first_id	uint64_t	First mother id

prt_mother_second_id	uint64_t	Second mother id (if applies)
prt_has_pol_info	bool	Has valid data in the next 3 fields
prt_pol_x	double	polarization info
prt_pol_y	double	
prt_pol_z	double	
prt_is_prim_vtx	bool	Does the particle belongs to the primary vtx
prt_is_sec_vtx		Does the particle belongs to a secondary vtx
prt_vtx_id	uint64_t	Unique vertex id in event
prt_gen_vtx_x	double	Vertex coordinates and time
prt_gen_vtx_y	double	
prt_gen_vtx_z	double	
prt_gen_vtx_t	double	

Smearing information

Field name	**Type**	**Description**
prt_has_smear_info	bool	
prt_smear_has_e	bool	Energy was smeared
prt_smear_has_p	bool	Momentum was smeared
prt_smear_has_pid	bool	PID is smeared
prt_smear_has_vtx	bool	Vertex is smeared
prt_smear_is_smeared	bool	Has smeared e, p, or pid
prt_smear_has_mom_theta	bool	Theta was smeared by a tracker
prt_smear_has_mom_phi	bool	Phi was smeared by a tracker
prt_smear_has_calo_theta	bool	Theta was smeared by a calorimeter

prt_smear_has_calo_phi	bool	Phi was smeared by a calorimeter

Smeared/reconstruction information

Field name	Type	Description
prt_rec_pid	int64_t	
prt_rec_tot_e	double	original Energy
prt_rec_p	double	double original total momentum
prt_rec_px	double	Original momentum components
prt_rec_py	double	
prt_rec_pz	double	
prt_rec_vtx_x	double	Original vertex components
prt_rec_vtx_y	double	
prt_rec_vtx_z	double	

Supporting software

What should be the minimal set of software around this specification.

1. Methods to extract DIS values from such tree (proposed by Andrea) for python and C++
2. Self validation test - tests if the tree corresponds to the specification and if values such as px,y,z and p are consistent.
3. Common test framework which should help establish testing (including unit testing) of code which uses this file formats:
    a. Provide examples of files (with different versions when versions will exceed 1.0.0)
    b. Provide code stubs to check values (for each field the file has)
    c. Unit test point 1 above

4. Provide examples and tutorials of reading the file in bare ROOT/uproot
5. Provide continuous integration and run tests that examples and test stubs always work
6. Spack support?