MAUS Analysis User System
User Guide

Chapter 1

Data Structure

1.1 Data Structure

The event in MAUS is the MICE spill. The major part of the MAUS data structure therefore is a tree of which each entry corresponds to the data associated with one spill. The spill is separated into three main sections: the MCEventArray contains an array of data each member of which represents the Monte Carlo of a single primary particle crossing the system; the ReconEventArray contains an array of data each member of which corresponds to a particle event (i.e. set of DAQ triggers); and the DAQData corresponds to the raw data readout. Additionally there are branches for reconstructed scalars, which are handled spill by spill and EMR data, which also read out on the spill rather than event by event.

The MCEvent is subdivided into sensitive detector hits and some pure Monte Carlo outputs. The primary that led to data being created is held in the Primary branch. Here the random seed, primary position momentum and so forth is stored. Sensitive detector hits have Hit data (energy deposited, position, momentum, etc) and a detector specific ChannelId that represents the channel of the detector that was hit - e.g. for TOF this indexes the slab, plane and station. Virtual hits are also stored - these are not sensitive detector hits, rather output position, momenta etc of particles that cross a particular plane in space, time or proper time is recorded. Note virtual hits do not inherit from the Hit class and have a slightly different data structure.

The ReconEvent and DAQEvents are subdivided by detector. ReconEvents contain reconstructed particle data for each detector and the trigger. There is an additional branch that contains global reconstruction output, that is the track fitting between detectors.

The data can be written in two formats. The main data format is a ROOT binary format. This requires the ROOT package to read and write, which is a standard analysis/plotting package in High Energy Physics and is installed by the MAUS build script. The secondary data format is JSON. This is an ascii data-tree format that in principle can be read by any text editor. Specific JSON parsers are also available - for example, the python *json* module is available and comes prepackaged with MAUS.

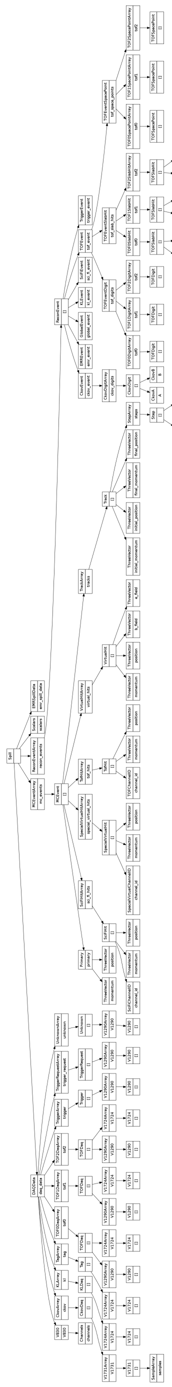

Figure 1.1: The MAUS output data structure

1.1.1 Loading ROOT Files in Python Using PyROOT

The standard scripting tool in MAUS is python. The ROOT data structure can be loaded in python using the PyROOT package. An example of how to perform a simple analysis with PyROOT is available in `bin/examples/load_root_file.py`.

1.1.2 Loading ROOT Files on the ROOT Command Line

One can load ROOT files from the command line using the ROOT interactive display. It is first necessary to load the MAUS class dictionary. Then The TBrowser ROOT GUI can be used to browse to the desired location and interrogate the data structure interactively. For example,

```
$ source env.sh
$ root

  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.30/03   20 October 2011   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

ROOT 5.30/03 (tags/v5-30-03@41540, Oct 24 2011, 11:51:36 on linuxx8664gcc)

CINT/ROOT C/C++ Interpreter version 5.18.00, July 2, 2010
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .L $MAUS_ROOT_DIR/build/libMausCpp.so
root [1] TBrowser b
```

1.1.3 Conversion to, and Working With, JSON

MAUS also provides output in the JSON data format. This is an ascii format with IO libraries available for C++, Python and other languages. Two utilities are provided to perform conversions, `bin/utilities/json_to_root.py` and `bin/utilities/root_to_json.py` for conversion from and to JSON format respectively. JSON Input and Output modules are provided, `InputPyJson` and `OutputPyJson`.

Accessing JSON is possible using the `json` python module that comes built-in to python 2.7. Each DAQ Event (real data) or Spill (Monte Carlo) in MAUS is written to a separate line, comprising a single json document, which can be loaded as a series of python dictionaries and arrays using `json.loads(a_line)` or written as a string using `json.dumps(a_line)`

1.1.4 Extending the Data Structure

The data structure can be extended in MAUS by adding extra classes to the existing data structure. The data classes are in `src/common_cpp/DataStructure`. In order to make these classes accessible to ROOT, the following steps must be taken:

- Add a new class
- Ensure that default constructor, copy constructor, equality operator and destructor is present
- Make a typedef for each type of STL object you wish to use. ROOT does not handle STL objects terribly well otherwise (and even then there are limitations).
- Add a call to the ClassDef() macro at the end of the class definition before the closing braces.
- Add the class to the list of classes in `src/common_cpp/DataStructure/LinkDef.hh`

In order to make these classes accessible to JSON, it is necessary to add a new processor in `src/common_cpp/JsonCppProcessors`. There are a few default processors available.

- `src/common_cpp/JsonCppProcessors/ProcessorBase.hh` contains IProcessor pure interface class for all processors and ProcessorBase base class (which may contain some implementation)
- `src/common_cpp/JsonCppProcessors/PrimitivesProcessors.hh` contains processors for primitive types; BoolProcessor, IntProcessor, UIntProcessor, StringProcessor, DoubleProcessor
- `src/common_cpp/JsonCppProcessors/ArrayProcessors.hh` contains processors for array types. Two processors are available: PointerArrayProcessor which converts an STL vector of pointers to data; and ValueArrayProcessor which converts an STL vector of values to data.
- `src/common_cpp/JsonCppProcessors/ObjectProcessor.hh` contains a processor for object types. Most of the classes in the MAUS data structure are represented in JSON as objects (string value pairs) where each string names a branch and each value contains data, which may be another class.
- `src/common_cpp/JsonCppProcessors/ObjectMapProcessors.hh` contains a processor for converting from JSON objects to STL maps. This is useful for JSON objects that contain lots of branches all of the same type.

A script, `bin/user/json_branch_to_data_structure_and_cpp_processor.py` is available that analyses a JSON object or JSON tree of nested objects and converts to C++ classes. The script is provided "as-is" and it is expected that developers will check the output, adding comments and tests where appropriate.

Chapter 2

Monte Carlo

The simulation module provides particle generation routines, GEANT4 bindings to track particles through the geometry and routines to convert modelled energy loss in detectors into digitised signals from the MICE DAQ. The Digitisation models are documented under each detector. Here we describe the beam generation and GEANT4 interface.

2.1 Beam Generation

Beam generation is handled by the MapPyBeamMaker module. Beam generation is separated into two classes. The MapPyBeamGenerator has routines to assign particles to a number of individual beam classes, each of which samples particle data from a predefined parent distribution. Beam generation is handled by the `beam` datacard.

The MapPyBeamMaker can either take particles from an external file, overwrite existing particles in the spill, add a specified number of particles from each beam definition, or sample particles from a binomial distribution. The random seed is controlled at the top level and different algorithms can be selected influencing how this is used to generate random seeds on each particle.

Each beam definition has routines for sampling from a multivariate gaussian distribution or generating ensembles of identical particles (called "pencil" beams here). Additionally it is possible to produce time distributions that are either rectangular or triangular in time to give a simplistic representation of the MICE time distribution.

The beam definition controls are split into four parts. The `reference` branch defines the centroid of the distribution; the `transverse` branch defines the transverse coordinates, x, y, px, py; the `longitudinal` branch defines the longitudinal coordinates - time and energy/momentum and the `coupling` branch defines correlations between longitudinal and transverse. Additionally a couple of parameters are available to control random seed generation and relative weighting between different beam definitions.

In transverse, beams are typically sampled from a multivariate gaussian.

The Twiss beam ellipse is defined by

$$\mathbf{B}_\perp = m \begin{pmatrix} \epsilon_x\beta_x/p & -\epsilon_x\alpha_x & 0 & 0 \\ -\epsilon_x\alpha_x & \epsilon_x\gamma_x p & 0 & 0 \\ 0 & 0 & \epsilon_y\beta_y/p & -\epsilon_y\alpha_y \\ 0 & 0 & -\epsilon_y\alpha_y & \epsilon_y\gamma_y p \end{pmatrix} \tag{2.1}$$

The Penn beam ellipse is defined by,

$$\mathbf{B}_\perp = m\epsilon_\perp \begin{pmatrix} \beta_\perp/p & -\alpha_\perp & 0 & -\mathcal{L}+\beta_\perp B_0/2p \\ -\alpha_\perp & \gamma_\perp p & \mathcal{L}-\beta_\perp B_0/2p & 0 \\ 0 & \mathcal{L}-\beta_\perp B_0/2p & \beta_\perp/p & -\alpha_\perp \\ -\mathcal{L}+\beta_\perp B_0/2p & 0 & -\alpha_\perp & \gamma_\perp p \end{pmatrix} \tag{2.2}$$

where parameters can be controlled in datacards

2.2 GEANT4 Bindings

The GEANT4 bindings are encoded in the Simulation module. GEANT4 groups particles by run, event and track. A GEANT4 run maps to a MICE spill; a GEANT4 event maps to a single inbound particle from the beamline; and a GEANT4 track corresponds to a single particle in the experiment.

A number of classes are provided for basic initialisation of GEANT4.

- MAUSGeant4Manager: is responsible for handling interface to GEANT4. MAUSGeant4Manager handles initialisation of the GEANT4 bindings as well as accessors for individual GEANT4 objects (see below). Interfaces are provided to run one or many particles through the geometry, returning the relevant event data. The MAUSGeant4Manager sets and clears the event action before each run.

- MAUSPhysicsList: contains routines to set up the GEANT4 physical processes. Datacards settings are provided to disable stochastic processes or all processes and set a few parameters.

- FieldPhaser: the field phaser is a MAUS-specific tool for automatically phasing fields, for example RF cavities, such that they ramp coincidentally with incoming particles. The FieldPhaser contains routines to fire test ("reference") particles through the accelerator lattice and phase fields appropriately. The FieldPhaser phasing routines are called after GEANT4 is first initialised.

- VirtualPlanes: the VirtualPlanes routines are designed to extract particle data from the GEANT4 tracking independently of the GEANT4 geometry. The VirtualPlanes routines watches for steps that step across some plane in physical space, or some time, or some proper time, and then interpolates from the step ends to the plane in question.

- FillMaterials: (legacy) the FillMaterials routines are used to initialise a number of specific

Table 2.1: Multiple beam control parameters.

Name	Meaning
`beam`	dict containing beam definition parameters.
The following cards should all be defined within the `beam` *dict.*	
`particle_generator`	Set to `binomial` to choose the number of particles by sampling from a binomial distribution. Set to `counter` to choose the number of particles in each beam definition explicitly. Set to `file` to generate particles by reading an input file. Set to `overwrite_existing` to generate particles by overwriting existing primaries.
`binomial_n`	When using a binomial `particle_generator`, this controls the number of trials to make. Otherwise ignored.
`binomial_p`	When using a binomial `particle_generator`, this controls the probability a trial yields a particle. Otherwise ignored.
`beam_file_format`	When using a file `particle_generator`, set the input file format - options are • `icool_for009` • `icool_for003`, • `g4beamline_bl_track_file` • `g4mice_special_hit` • `g4mice_virtual_hit` • `mars_1` • `maus_virtual_hit` • `maus_primary`
`beam_file`	When using a file `particle_generator`, set the input file name.
`file_particles_per_spill`	When using a file `particle_generator`, this controls the number of particles per spill that will be read from the file.
`random_seed`	Set the random seed, which is used to generate individual random seeds for each primary (see below).
`definitions`	A list of dicts, each item of which is a dict defining the distribution from which to sample individual particles.

Table 2.2: Individual beam distribution parameters.

Name	Meaning
The following cards should be inside a dict in the beam `definitions` list.	
`random_seed_algorithm`	Choose from the following options • `beam_seed`: use the `random_seed` for all particles • `random`: use a different randomly determined seed for each particle • `incrementing`: use the `random_seed` but increment by one each time a new particle is generated • `incrementing_random`: determine a seed at random before any particles are generated; increment this by one each time a new particle is generated
`weight`	When `particle_generator` is `binomial` or `overwrite_existing`, the probability that a particle will be sampled from this distribution is given by $weight/(sumofweights)$.
`n_particles_per_spill`	When `particle_generator` is `counter`, this sets the number of particles that will be generated in each spill.
`reference`	Dict containing the reference particle definition.
`transverse`	Dict defining the longitudinal phase space distribution.
`longitudinal`	Dict defining the longitudinal phase space distribution.
`coupling`	Dict defining any correlations between transverse and longitudinal.

Table 2.3: Beam distribution reference definition.

Name	Meaning
The following cards should be defined in each beam definition `reference` dict.	
`position`	dict with elements `x`, `y` and `z` that define the reference position (mm).
`momentum`	dict with elements `x`, `y` and `z` that define the reference momentum direction. Normalised to 1 at runtime.
`particle_id`	PDG particle ID of the reference particle.
`energy`	Reference energy.
`time`	Reference time (ns).
`random_seed`	Set to 0 - this parameter is ignored.

Table 2.4: Beam definition transverse parameters.

Name	Meaning
The following cards should be defined in each beam definition `transverse` *dict.*	
`transverse_mode`	Options are • `pencil`: x, py, y, py taken from `reference` • `penn`: cylindrical beam symmetric in x and y • `constant_solenoid`: cylindrical beam symmetric in x and y, with beam radius calculated from on-axis B-field to give constant beam radius along a solenoid. • `twiss`: beam with decoupled x and y beam ellipses.
`normalised_angular_momentum`	if `transverse_mode` is `penn` or `constant_solenoid`, set $\mathcal{L}$.
`emittance_4d`	if `transverse_mode` is `penn` or `constant_solenoid`, set $\epsilon_\perp$.
`beta_4d`	if `transverse_mode` is `penn`, set $\beta_\perp$.
`alpha_4d`	if `transverse_mode` is `penn`, set $\alpha_\perp$.
`bz`	if `transverse_mode` is `constant_solenoid`, set the B-field used to calculate $\beta_\perp$ and $\alpha_\perp$.
`beta_x`	if `transverse_mode` is `twiss`, set β_x.
`alpha_x`	if `transverse_mode` is `twiss`, set α_x.
`emittance_x`	if `transverse_mode` is `twiss`, set ϵ_x.
`beta_y`	if `transverse_mode` is `twiss`, set β_y.
`alpha_y`	if `transverse_mode` is `twiss`, set α_y.
`emittance_y`	if `transverse_mode` is `twiss`, set ϵ_y.

Table 2.5: Beam definition longitudinal parameters.

Name	Meaning
The following cards should be defined in each beam definition ***longitudinal*** *dict.*	
momentum_variable	In all modes, set this variable to control which longitudinal variable will be used to control the input beam. Options are energy, p, pz.
longitudinal_mode	Options are • pencil: time, energy/p/pz taken from reference • gaussian: uncorrelated gaussians in time and energy/p/pz • twiss: multivariate gaussian in time and energy/p/pz • uniform_time: gaussian in energy/p/pz and uniform in time. • sawtooth_time: gaussian in energy/p/pz and sawtooth in time.
beta_l	In Twiss mode, set β_l
alpha_l	In Twiss mode, set α_l
emittance_l	In Twiss mode, set ϵ_l
sigma_t	In gaussian mode, set the RMS time.
sigma_p sigma_energy sigma_pz	In gaussian, uniform_time, sawtooth_time mode, set the RMS energy/p/pz.
t_start	In uniform_time and sawtooth_time mode, set the start time of the parent distribution
t_end	In uniform_time and sawtooth_time mode, set the end time of the parent distribution

Table 2.6: Beam definition coupling parameters.

Name	Meaning
The following cards should be defined in each beam definition ***coupling*** *dict.*	
coupling_mode	Set to none - not implemented yet.

- MICEDetectorConstruction: (legacy) the MICEDetectorConstruction routines provide an interface between the MAUS internal geometry representation encoded in MiceModules and GEANT4. MICEDetectorConstruction is responsible for calling the relevant routines for setting up the general engineering geometry, calling detector-specific geometry set-up routines and calling the field map set-up routines.

- MAUSVisManager the MAUSVisManager is responsible for handling interfaces with the GEANT4 visualisation.

The GEANT4 *Action* objects provide interfaces for MAUS-specific function calls at certain points in the tracking.

- MAUSRunAction: sets up the running for a particular spill. In MAUS, it just reinitialises the visualisation.

- MAUSEventAction: sets up the running for a particular inbound particle. At the beginning of each event, the virtual planes, tracking, detectors and stepping are all cleared. After the event the event data is pulled into the event data from each element.

- MAUSTrackingAction: is called when a new track is created or destroyed. If `keep_tracks` datacard is set to True, on particle creation, MAUSTrackingAction writes the initial and final track position and momentum to the output data tree. If `keep_steps` is set to True MAUSTrackingAction gets step data from MAUSSteppingAction and writes this also.

- MAUSSteppingAction: is called at each step of the particle. If `keep_steps` datacard is set to True, output step data is recorded. MAUSSteppingAction kills particles if they exceed the `maximum_number_of_steps` datacard. MAUSSteppingAction calls the VirtualPlanes routines on each step.

- MAUSPrimaryGeneratorAction: is called at the start of every event and sets the particle data for each event. In MAUS, this particle generation is handled externally and so the MAUSPrimaryGeneratorAction role is to look for the primary object on the Monte Carlo event and convert this into a GEANT4 event object.

- MAUSStackingActionKillNonMuons: is never initialised and should be removed.

Table 2.7: Monte Carlo control parameters.

Name	Meaning
General Monte Carlo controls.	
`simulation_geometry_filename`	Filename for the simulation geometry - searches first in files tagged by environment variable `${MICEFILES}`, then in the local directory.
`simulation_reference_particle`	Reference particle used for phasing fields.
`keep_tracks`	Set to boolean true to store the initial and final position/momentum of each track generated by MAUS.
`keep_steps`	Set to boolean true to store every step generated by MAUS - warning this can lead to large output files.
`maximum_number_of_steps`	Set to an integer value. Tracks taking more steps are assumed to be looping.

Table 2.8: Physics list control parameters.

Physics list controls.	
`physics_model`	GEANT4 physics model used to set up the physics list.
`physics_processes`	Choose which physics processes normal particles observe during tracking. Options are • `normal` particles will obey normal physics processes, scattering and energy straggling will be active. • `mean_energy_loss` particles will lose a deterministic amount of energy during interaction with materials and will never decay. • `none` Particles will never lose energy or scatter during tracking and will never decay.
`reference_physics_processes`	Choose which physics processes the reference particle observes during tracking. Options are `mean_energy_loss` and `none`. The reference particle can never have stochastic processes enabled.
`particle_decay`	Set to boolean true to enable particle decay; set to boolean false to disable.
`charged_pion_half_life`	Set the half life for charged pions.
`muon_half_life`	Set the half life for muons.
`production_threshold`	Set the geant4 production threshold.

Table 2.9: Visualisation control parameters.

Visualisation controls.	
geant4_visualisation	Set to boolean true to activate GEANT4 visualisation.
visualisation_viewer	Control which viewer to use to visualise GEANT4 tracks. Currently only vrmlviewer is compiled into GEANT4 by default. Users can recompile GEANT4 with additional viewers enabled at their own risk.
visualisation_theta	Set the theta angle of the camera.
visualisation_phi	Set the phi angle of the camera.
visualisation_zoom	Set the camera zoom.
accumulate_tracks	Not implemented.
default_vis_colour	Not implemented.
pi_plus_vis_colour	Not implemented.
pi_minus_vis_colour	Not implemented.
mu_plus_vis_colour	Not implemented.
mu_minus_vis_colour	Not implemented.
mu_plus_vis_colour	Not implemented.
mu_minus_vis_colour	Not implemented.
e_plus_vis_colour	Not implemented.
e_minus_vis_colour	Not implemented.
gamma_vis_colour	Not implemented.
neutron_vis_colour	Not implemented.
photon_vis_colour	Not implemented.