Cexmc User's Manual

Index

1 Motivation....3
2 Setup geometry....3
3 Run modes....5
4 Physics list....5
5 Fast simulation model....7
6 Charge exchange production model....9
7 Sensitive detectors, hits and digits....9
8 Physics reconstruction....10
9 Persistency....11
10 Custom filter....11
11 Histograming....13
12 User commands....13
13 Utilities....13

1 Motivation

Program **Cexmc** (which stands for **C**harge **EX**change **M**onte **C**arlo) was used to find best geometry of a physical setup and calculate *acceptances* of this setup for 2 types of hadronic interactions:

- $\boldsymbol{\pi^- p \rightarrow \pi^0 n}$
- $\boldsymbol{\pi^- p \rightarrow \eta n}$

The interactions were studied for different energies of $\boldsymbol{\pi^-}$ in momentum range ~ 400 - 720 MeV/c for $\boldsymbol{\pi^0}$ production and ~ 700 - 730 MeV/c for $\boldsymbol{\eta}$ production.

The physical setup can have different degrees of freedom (calorimeters are freely moved combining different distances and angles in respect to the target: see ***Fig.1***) and its best geometry is roughly such a configuration when acceptances have maximum values.

Setup acceptances can be calculated using Monte Carlo method. One might define them as a set of numbers that give a fraction of triggered events from the whole number of simulated interactions when output particle is scattered in a given solid angle in the Center of Mass system. Thus setup acceptances are parameterized by angular ranges and strictly bound to physical setup, trigger logic and studied interaction. Knowing setup acceptances makes it possible to find differential cross-sections of the studied interaction.

The value of differential cross-section in a given solid angle can be deduced from experimental data and the value of acceptance for this angle:

$$\frac{d\sigma}{d\Omega} \propto \frac{N_\Omega}{A_\Omega}$$

where N_Ω is number of experimental events registered in a given solid angle and A_Ω is calculated value of the setup acceptance for this solid angle.

Despite the program was developed for a concrete physical experiment its structure is flexible enough to support different setup geometries and interaction models. This flexibility is a result of using GDML for description of setup geometry and a special fast simulation engine that provides easy replacement of studied interactions.

2 Setup geometry

The physical setup is constructed from a cylindrical liquid hydrogen target with radius 5cm, two calorimeters that make 2 arms of the setup and contain 6x4 sets of CsI crystals, a monitor counter before the target, 2 veto counters before the calorimeters and various target constructive elements. The calorimeters will detect decay products of the output particle ($\boldsymbol{\pi^0}$ or $\boldsymbol{\eta}$). The most intensive decay mode for the both particles is $2\boldsymbol{\gamma}$. The 2 arms of the setup are responsible for catching the gammas and collect their energies. The setup is depicted on ***Fig.1***.

The setup is completely described in file *lht.gdml*. Besides the geometry this file contains information about sensitive detectors using tag *auxiliary* with attributes *auxtype* equal to *EnergyDepositDetector* or *TrackPointsDetector*. Tag *auxiliary* with attribute *auxtype* equal to *SpecialVolume* marks logical volumes that play important roles in the program: the monitor counter, the target and the calorimeters. Tag *auxiliary* with attribute *auxtype* equal to *SensitiveRegion* marks logical volumes where user can define custom production cuts (particularly, setting custom cuts is

supported in calorimeters).

Using *lht.gdml* is not mandatory. In fact, geometry file is set by command */cexmc/geometry/gdmlFile* in preinit phase. In principle user can completely change geometry description of the setup, but any description must meet following requirements:

1. A logical volume of the target must be tagged by tag *auxiliary* with *auxtype SpecialVolume* and *auxvalue 3*. The corresponding physical volume is referenced in user *stepping* and *tracking* actions.
2. The setup must have 2 arms: *left* and *right*. The arms contain detectors that collect energies of particles produced in studied interactions. Also, names of physical volumes in the right arm, that have been made sensitive detectors, must contain *Right*.
3. Three pairs of *positions* and *rotations* must be defined: *TargetPos*, *TargetRot*, *CalorimeterLeftPos*, *CalorimeterLeftRot*, *CalorimeterRightPos* and *CalorimeterRightRot*. They must correspond to *absolute* positions and rotations of the target and the calorimeters in the *world* volume. Their values are used in reconstruction procedures.
4. A logical volume of the calorimeter must be tagged by tag *auxiliary* with *auxtype SpecialVolume* and *auxvalue 2*. The calorimeters must be replicated horizontally and vertically which makes rows and columns of separate *crystals*. Exact number of replicated volumes is not important. Crystals may be wrapped in paper or other materials.
5. Following constraints exist, but they can be removed in future (currently they are mostly referenced in *CexmcHistoManger*): solids of logical volumes of calorimeters, crystals and the monitor counter must be *boxes*, solid of logical volume of the target must be a *tube*; a logical volume of the calorimeter must be tagged as *SensitiveRegion* with *auxvalue 0*; logical volumes of the monitor counter and veto counters must be tagged as *SpecialVolume* with *auxvalue 0* and *1* respectively.

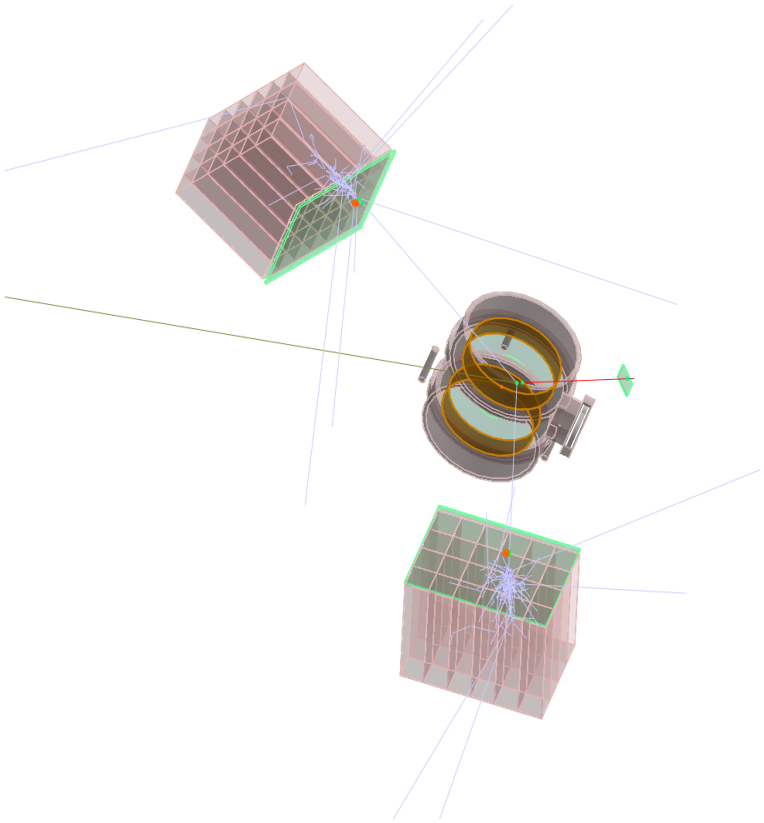

Figure 1: the physical setup

3 Run modes

Cexmc has 3 major operational modes:

1. *Straight mode* (or *Monte Carlo mode*). The program will read *preinit* and *init* macros, then calculate acceptances and (optionally) save data in *project* files. Project files are saved in a directory defined by environment variable **CEXMC_PROJECTS_DIR** (or in the current directory if it is not defined), name of the project is specified by option *-w*. *Preinit* and *init* macros are set by options *-p* and *-m* respectively. In the straight mode preinit macro must be specified explicitly, as far as desired production model can be instantiated only in preinit phase.
2. *Replay mode* (or *Read project mode*). In this mode the program will not use common Geant4's event loop. Instead, it will sequentially read event data from an existing project and pass them into *CexmcEventAction::EndOfEventAction()*. The read project is specified by option *-r*. This mode is useful when user wants to recalculate data from an existing project with different conditions (for example with different reconstruction parameters). The results of run can be written again into another project. In fact, user can make long chain of *write-read-write* runs sequentially tuning different parameters, but validity of results of a *long chain run* depends on event data verbose level specified in init macro in in-between runs.
3. *Show results mode* (or *Output mode*). The program will output various data from an existing project (specified by option *-r*). Type(s) of data are specified in option *-o*. For example, to show results of a run user can specify *-orun* in command line. To show events, geometry and run results user can specify *-oevents,geom,run*.

There are also 3 different run modes classified by user interaction type:

1. *Batch mode*. The program will run without any interaction with user.
2. *Interactive mode*. The program provides an interactive shell. To run in the interactive mode command line option *-i* must be specified.
3. *Graphical Qt mode*. This mode is specified by command line option *-g*.

In Cexmc number of events specified by command */run/beamOn* may have different meanings which depends on the value set by command */cexmc/run/eventCountPolicy*. If *eventCountPolicy* is *all* or not specified then the run manager will generate the specified number of events as usual, if it is *interaction* then the run manager will generate the specified number of the studied interactions, and if it is *trigger* then the run manager will generate the specified number of the setup triggers.

4 Physics list

Cexmc physics list consists of two parts: one of Geant4's reference physics list and a production model provided by user for fast simulation in the target. Both reference physics list and production model are instantiated as template arguments of class template *CexmcProductionModelFactory* which is defined as

```
template  < typename  BasePhysics,
            template  < typename > class  StudiedPhysics,
            template  < typename > class  ProductionModel >
class  CexmcProductionModelFactory
{
    public:
        static G4VUserPhysicsList *    Create(
```

```
                                    CexmcProductionModelType  productionModelType );

        static CexmcBasePhysicsUsed    GetBasePhysics( void );

    private:
        CexmcProductionModelFactory();
};
```

Template argument *BasePhysics* is to be substituted by a reference physics list; argument *StudiedPhysics* corresponds to user's physics and must be implemented as a class template parameterized by a *G4VProcess* subclass and derived from *G4VPhysicsConstructor*; argument *ProductionModel* corresponds to user's interaction model: this is a class template parameterized by an arbitrary type (in case of Charge Exchange production model it is parameterized by output particle's type, e.g. π^0 or η). Production model must inherit class *CexmcProductionModel* which provides methods to access production model data and methods to set and access triggered angular ranges.

Concrete production model factory is instantiated in *CexmcBasicPhysicsSettings.hh*:

```
#ifdef CEXMC_USE_QGSP_BIC_EMY
typedef QGSP_BIC_EMY                    CexmcBasePhysics;
#else
typedef QGSP_BERT                       CexmcBasePhysics;
#endif

typedef CexmcProductionModelFactory< CexmcBasePhysics,
                                     CexmcHadronicPhysics,
                                     CexmcChargeExchangeProductionModel >
                                      CexmcChargeExchangePMFactory;

typedef CexmcChargeExchangePMFactory  CexmcPMFactoryInstance;
```

Two reference physics list are supported from the makefile: **QGSP_BERT** and **QGSP_BIC_EMY,** they are controlled by macro **CEXMC_USE_QGSP_BIC_EMY**. If user wants to use a different reference physics list he can edit *CexmcBasicPhysicsSettings.hh*.

CexmcProductionModelFactory has method *Create()* which creates and returns an object of class template *CexmcPhysicsList* instantiated with template arguments supplied by the factory. *CexmcPhysicsList* inherits reference physics list and *CexmcPhysicsManager*:

```
template  < typename  BasePhysics, template  < typename > class  StudiedPhysics,
            typename  ProductionModel >
class  CexmcPhysicsList : public BasePhysics, public CexmcPhysicsManager
{
    public:
        CexmcPhysicsList();

    public:
        CexmcProductionModel *  GetProductionModel( void );

        G4bool                  IsStudiedProcessAllowed( void ) const;

        void                    ResampleTrackLengthInTarget(
                                    const G4Track *  track,
                                    const G4StepPoint *  stepPoint );

    private:
        StudiedPhysics< ProductionModel > *  studiedPhysics;
```

```
        G4bool                                  proposedMaxILInitialized;

        G4double                                proposedMaxIL;
};
```

Class *CexmcPhysicsManager* implements a number of methods related to fast simulation model and has interface method *GetProductionModel()*. Created object is passed as *physics manager* into constructors of a number of management objects like *CexmcEventAction*, *CexmcRunAction*, *CexmcTrackingAction* and *CexmcSteppingAction* which enables them to access the production model object.

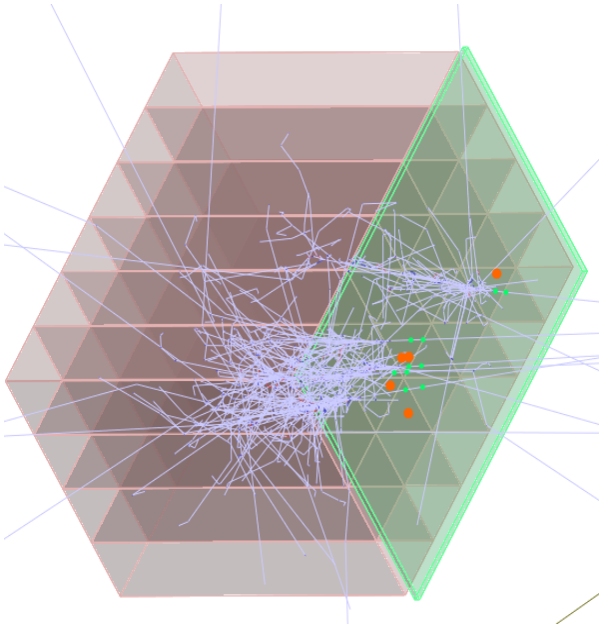

Figure 2a: electro-magnetic interactions caused by reference physics list

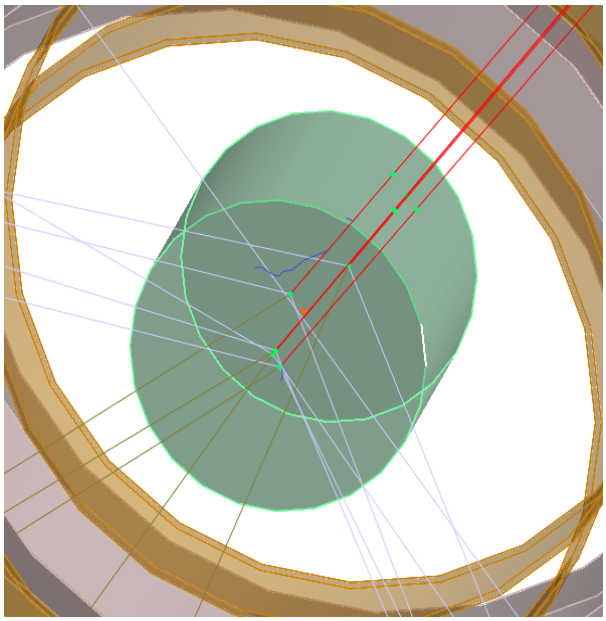

Figure 2b: studied interactions in the target caused by instantiated charge exchange production model

5 Fast simulation model

Fast simulation model implements production model simulation in the target. The idea of fast simulation is to make studied interaction occur in the target (and *only* in the target) as often as possible. Actually it makes them occur *almost* in every event generated by Geant4 engine. This *almost* arises from 2 major facts:

1. Not all incident beam particles visit the target (for example, if simulated beam is very wide or a beam particle was scattered and missed the target).
2. There is one important restriction on the distribution of studied interaction vertices: it must be uniform along the beam path in the target. From the point of view of an engine which samples vertices of studied interaction in the target, it means that if geometry of the target makes path length of beam particles depend on position of the particle on the beam projection, then probabilities of interaction sampling must differ for different positions of the particle on the beam projection. For physical setup on ***Fig.1*** particles on left and right edges of a wide beam projection will only scratch left or right edges of the cylindrical target, and an interaction sampling engine must miss most of interactions for such edgy particles to ensure uniform distribution of interaction vertices in the target.

Many methods related to fast simulation engine are located in classes *CexmcPhysicsManager* and *CexmcStudiedProcess* (this class inherits *G4WrapperProcess* and registers a physical process supplied in template argument of class template *CexmcStudiedPhysics* in its method *ConstructProcess()*).

However the most interesting parts occur in *CexmcTrackingAction::PreUserTrackingAction()* and *CexmcSteppingAction::UserSteppingAction()*. The *PreUserTrackingAction() tags* particles which definition matches the *incident* particle of instantiated production model with an object of type *CexmcIncidentParticleTrackInfo* (derived from *G4VUserTrackInformation*). This object will hold information related to fast simulation engine for this particle (for example current track length in the target and sampled track length).

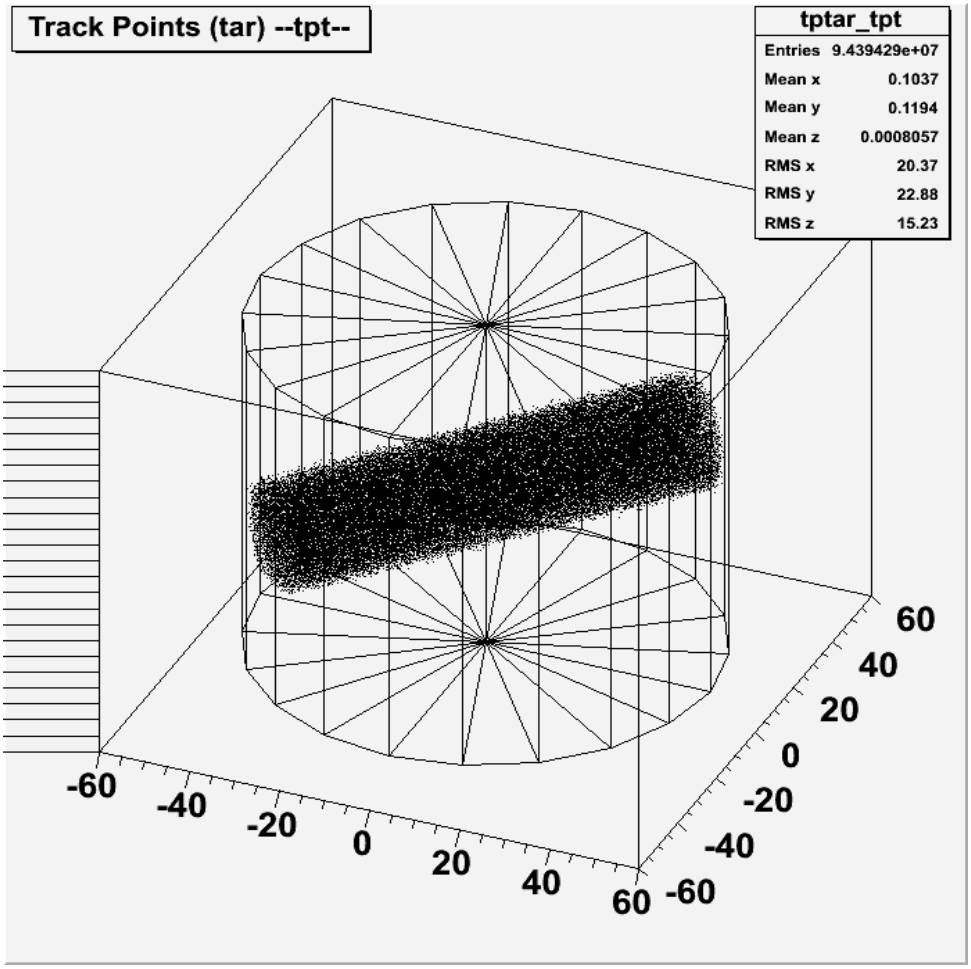

Figure 3: uniform distribution of studied interaction vertices in the cylindrical target

Track length is sampled in *CexmcPhysicsManager::ResampleTrackLengthInTarget()* in *PreUserTrackingAction()*. This method can be called once again for the same particle in *UserSteppingAction()* (for example if the particle visits the target second time).

CexmcStudiedProcess watches to sample the studied interaction in its method *PostStepGetPhysicalInteractionLength()*. It returns difference between sampled for this particle track length in the target and current track length the particle already passed. Geant4 uses this value to know when to simulate a physical process that was registered in *CexmcStudiedProcess*.

6 Charge exchange production model

Class template *CexmcChargeExchangeProductionModel* implements the two hadronic models mentioned in the beginning of this document. It inherits *G4HadronicInteraction* and *CexmcProductionModel*. The only method it provides is *ApplyYourself()*. *ApplyYourself()* uses a *phase space generator* to generate secondaries and add them to *theParticleChange*.

There are two phase space generators that user can choose: original *cernlib* FORTRAN routine *GENBOD()* called from *CexmcGenbod::Generate()* and same routine reimplemented in class *CexmcReimplementedGenbod*. Implementation of GENBOD() in *CexmcReimplementedGenbod* was mostly adopted from CERN ROOT class *TGenPhaseSpace*. The first choice (*CexmcGenbod*) makes the program depend on FORTRAN libraries and *cernlib*. The choice are controlled by macro **CEXMC_USE_GENBOD**; by default **CEXMC_USE_GENBOD** is not defined.

7 Sensitive detectors, hits and digits

Cexmc collects two kinds of hits: energy deposits in detectors and track points of various tracks of interest (beam particle, output particle, decay products of output particle etc.). Related sensitive detectors are created in *lht.gdml* with help of GDML tag *auxiliary* and attributes *EnergyDepositDetector* and *TrackPointsDetector*. To avoid direct correlation between a logical volume name and a sensitive detector name in the program code a notion of *detector role* was introduced: detector role is specified in the attribute *auxvalue* of the tag *auxiliary*. There are 4 kinds of detector roles: *Monitor*, *VetoCounter*, *Calorimeter* and *Target*. Logical volumes may share same roles but one logical volume may not have 2 or more *different* roles.

Energy deposit (*ED*) hits are implemented in class *CexmcSimpleEnergyDeposit* (collects ED in the monitor counter) and its descenders *CexmcEnergyDepositInLeftRightSet* (ED in veto counters) and *CexmcEnergyDepositInCalorimeter* (ED in calorimeters). All classes use event map of type *G4ThitsMap<G4double>*, the main difference between them is how parts of detectors are indexed in the event map. Indices of *G4THitsMap<>* have type *G4int*. *CexmcSimpleEnergyDeposit* collects all events in index 0; *CexmcEnergyDepositInLeftRightSet* collects events in the left detector in index 0 and events in the right detector in index 1<<16 (i.e. an integer value with 16^{th} bit set); *CexmcEnergyDepositInCalorimeter* marks that an event has occurred in the right calorimeter setting 16^{th} bit, number of row occupies bits 8^{th} - 15^{th}, number of column occupies bits 0^{th} - 7^{th}.

Track points (*TP*) hits are implemented in a similar manner in classes *CexmcTrackPoints*, *CexmcTrackPointsInLeftRightSet* and *CexmcTrackPointsInCalorimeter*. The main difference is that ED hits store integral value of all energy deposited in a detector or one of its part, but TP hits store information about multiple tracks (using specific *id* which contains *type* of a track and its *copy number*). Therefore *CexmcTrackPoints* collects events in indices that correspond to the value of the *id*; *CexmcTrackPointsInLeftRightSet* marks that an event has occurred in the right detector setting 24^{th} bit and the *id* occupies bits 0^{th} - 23^{rd}; *CexmcTrackPointsInCalorimeter* marks that an event has occurred in the right detector setting 24^{th} bit, number of row occupies bits 16^{th} - 23^{rd}, number of column occupies bits 8^{th} - 15^{th} and the *id* occupies bits 0^{th} - 7^{th}.

Energy deposit hits are used in method *CexmcEnergyDepositDigitizer::Digitize()* to detect if the setup has triggered. First of all *Digitize()* converts ED data from calorimeters if user has parameterized resolution of calorimeters in init macro with commands */cexmc/detector/addCrystalResolutionRange*. Then *Digitize()* applies the trigger logic.

The trigger logic takes information from the monitor counter, veto counters and calorimeters and tests if the monitor ED is bigger than related monitor threshold (it means that a charged particle —

supposedly π^-, has passed the monitor), ED in each veto counter is smaller than veto counter thresholds (it means that no charged particle has passed the veto counters) and full ED in *inner* crystals of each calorimeter is bigger than calorimeter thresholds (it means that calorimeters has supposedly caught the decay products of output particles, i.e. the 2γ). The energy thresholds are specified by user in init macro. There are several variations of basic trigger logic related mostly to filtering out events when most energy deposit spot is located in *outer* crystals of a calorimeter; all variations of the trigger logic are controlled by user in init macro.

In *CexmcEventAction::EndOfEventAction()* all collected energy deposits data are converted into a new object of type *CexmcEnergyDepositStore* which is then passed to the *reconstructor* methods, can be printed out on screen, used for histograming and saved in event data project files. Class *CexmcEnergyDepositStore* uses *G4Allocator* as far as its objects are created and deleted in every *EndOfEventAction()*.

Track points hits are used in method *CexmcTrackPoitntsDigitizer::Digitize()* to detect if a studied interaction has occurred. In *CexmcEventAction::EndOfEventAction()* track points data are converted into a new object of type *CexmcTrackPointsStore* which is then can be printed on screen, marked by a visualizer (the green dots on ***Fig.2a*** and ***2b***), used for histograming and saved in event data project files. Class *CexmcTrackPointsStore* uses *G4Allocator* as far as its objects are created and deleted in every *EndOfEventAction()*.

8 Physics reconstruction

Energy deposit and track points hits are *true* or *real* Monte Carlo data. One can compare them with the real physics that hides behind the experimental data. In real world measured experimental data does not have to be the exact reflection of the underlying physics: we cannot know exactly energy of the incident beam particles or coordinates of the studied interaction vertices in the target, instead we have to suppose that the beam particle had some average energy and that the studied interaction vertex was exactly in the center of the target. But using a specific *reconstruction model* we can reconstruct physics processes from experimental data obtained from the physical setup. To be sure that calculated acceptances correspond to the experimental data we have to apply same reconstruction model to the *true* Monte Carlo data.

The main target of the experiment is to find differential cross-sections of the studied interactions. The reconstruction model has to retrieve the angle of the studied interaction output particle in the Center of Mass system both for experimental and *true* Monte Carlo data. Knowing angles of output particles let us parameterize experimental and Monte Carlo data by these angles and thus find the cross-sections.

Cexmc finds two kinds of setup acceptances. The first kind is the *real* acceptances: they are parameterized by angles from the *true* Monte Carlo data, the second kind is the *reconstructed* acceptances which are parameterized by reconstructed angles of the output particle.

The reconstruction model is applied only when the setup has triggered and includes following parts:

1. Reconstruction of energies of the incident beam particle and of the output particle's decay products (the 2γ) having entered the calorimeters, vertices of the interaction in the target and of entries of the 2γ in the calorimeters.
2. Reconstruction of the studied interaction and the angle of the output particle in the Center of Mass system.
3. Optionally reconstruction model may include different energy cuts and other parameters that

will increase quality of the calculated data (but possibly make statistics smaller).

The first part of the reconstruction is implemented in class *CexmcReconstructor*. The second part is implemented in class *CexmcChargeExchangeReconstructor*. Both parts have plenty of settings that user can choose to adjust the reconstruction model for his requirements. All these settings are configured in init macro by the */cexmc/reconstructor/* subcommands and may be freely customized in the *replay* run mode.

Reconstructed vertices of the studied interaction and the entries of the output particle's decay products in the calorimeters are visualized with red dots (see ***Fig.2a*** and ***2b***).

9 Persistency

Cexmc uses concept of a *project*. Project files are saved in a directory specified by environment variable **CEXMC_PROJECTS_DIR** and then can be retrieved by Cexmc in *show results* mode or reused in *replay* mode. Name of the project is specified by option *-w*. Project files include files with extensions *.rdb* (run data base), *.fdb* (fast events data base), *.edb* (events data base), *.root* and *.gdml*. File with extension *.root* stores histograms, with extension *.gdml* — geometry description actual for this project, with extension *.rdb* — results of the run, *.fdb* and *.edb* — events data of the run. Files *.rdb*, *.fdb* and *.edb* have binary format and are created using *boost::serialize* library.

Files *.fdb* and *.edb* are complementary, their content depends on values of *eventDataVerboseLevel* and *skipInteractionsWithoutEDT* that can be set in init macro in */cexmc/run/* directory. There are 4 levels of event data verbosity:

1. Do not save event data (*/cexmc/run/eventDataVerboseLevel* is *nosave*).
2. Save minimum event data (*economy mode*). In this case *eventDataVerboseLevel* is *trigger*. All events with triggered interaction *or* the setup trigger are marked in the *.fdb* file; *.edb* file has verbose information only on events with the setup trigger. Data can be used in *replay* mode, but if this mode was used in *replay* run mode then next *replay* run will give wrong acceptances.
3. Save minimum event data with *skipInteractionsWithoutEDT* equal *true*. This mode is effective for the replay run mode and must be used in *long chain runs* (i.e. multiple sequential *replay* runs).
4. Save all event data (*greedy mode*). In this case *eventDataVerboseLevel* is *interaction*. Every event entry available in *.fdb* file has a counterpart in *.edb* file. This mode is useful if user wants to use the *custom filter* for simulated data in future, but size of data will be much larger than in economy modes.

10 Custom filter

The *custom filter* is an optional feature. Compilation of the custom filter is controlled by macro **CEXMC_USE_CUSTOM_FILTER**. It can be used only in the *replay* run mode. The idea of the custom filter is to make it possible to remove specific events from *.fdb* and *.edb* files in a very flexible way. This can be useful for example when the experimental data have some known defect. Imagine that you know that your left calorimeter was misconfigured in the experiment and all data with energy deposits more than a specific value are wrong. With the custom filter you won't have to throw the experimental data or compile another algorithm in the Cexmc. All you need is to remove defective events from the experimental data and create specific *custom filter script* to remove events that meet same dynamic requirements from Monte Carlo events. The custom filter has even more power allowing to investigate changes in geometry of the setup after run: for example user can

select to study an area within the target by removing events when the studied interactions occurred outside this area.

To achieve this dynamic flexibility the custom filter implements a specific arithmetic language using *boost::spirit* library. This language allows arbitrary arithmetic expressions with type safety in mind, and is aware of variables related to events data and several functions like *Sqr()* and *Sqrt()*. There is an example of custom filter script *example.cf* in the code repository.

The custom filter script consists of several expressions that are evaluated sequentially from top to bottom. Expressions start from commands *delete tpt*, *keep tpt*, *delete edt* or *keep edt* appended by an arbitrary logical expression. Evaluation stops as soon as current logical expression is evaluated to *true* and the related command is executed on the current event. *Tpt* (*track points trigger*) events are related to events with studied interaction triggered and file *.fdb* and *edt* (*energy deposit trigger*) events — to events with the setup trigger and file *.edb*. As such using most of *tpt* commands requires that event data were saved with the *greedy* event data verbosity. Only a few *tpt* commands are safe when events data have been saved in an *economy* mode: they are listed in the *example.cf*.

Here is the content of the *example.cf* to feel the thing:

```
# I. EDT examples

# EDT: delete events when sum of absorbed energy in left calorimeter is more
# than specified value
#delete edt if Sum( clEDcol ) > 350 * MeV

# EDT: keep events if sum of absorbed energy in two specified crystals in left
# calorimeter is more than sum of absorbed energy in outer crystals of left
# calorimeter
#keep edt if clEDcol[2,2] + clEDcol[2,3] > Sum( Outer( clEDcol ) )

# EDT: delete events if sum of absorbed energy in inner crystals of right
# calorimeter is more than specified value
#delete edt if Sum( Inner( crEDcol ) ) > 200 * MeV


# II. TPT examples. Can be safely used only for rich event data sets

# TPT: make diagonal cut in monitor counter
delete tpt if bp_mon_posl[1] < bp_mon_posl[2]

# TPT: make target radius smaller
delete tpt if Sqrt(Sqr(op_tgt_posl[1]) + Sqr(op_tgt_posl[2]) + \
                   Sqr(op_tgt_posl[3])) > 2 * cm


# III. TPT examples. Can be safely used for any event data sets

# TPT: delete all events in angular range (0.0000, -1.0000)
#delete tpt if op_cosTh_SCM < 0

# TPT: delete all events with id less than 500
#delete tpt if event < 500

# TPT: delete all events when monitor was not triggered
#delete tpt if ! mon

# TPT: delete all events without TPT (i.e. false triggered events)
#delete tpt if ! tpt
```

11 Histograming

Histograming is an optional feature implemented using **CERN ROOT** libraries. Compilation of the histograming manager is controlled by macro **CEXMC_USE_ROOT**, this macro is set automatically in the makefile if ROOT libraries were found in the system.

Histograms are saved besides other project files in a file with extension *.root*. Available histograms can be listed with command */cexmc/histo/list*. *Live histograms* are available in interactive Qt sessions; they can be shown on screen even if no project is written.

12 User commands

There are many user commands available in */cexmc/* directory. Subdirectories include *geometry/*, *physics/*, *gun/*, *detector/*, *event/*, *run/*, *reconstructor/*, *vis/* and *histo/*. All user commands are well documented.

13 Utilities

There are several utilities in directory *util*. Utilities *mkjob* and *jobprog* are related to running Cexmc in the *gridengine* environment: *mkjob* creates a job and *jobprog* shows all running Cexmc instances and related process information like completeness in percents and current size of written project files. Utility *mkacchist* reads output of *cexmc* launched in the *show results mode* and creates a C script which then can be used in the ROOT environment to create a nice graphical image of the calculated acceptances like this:

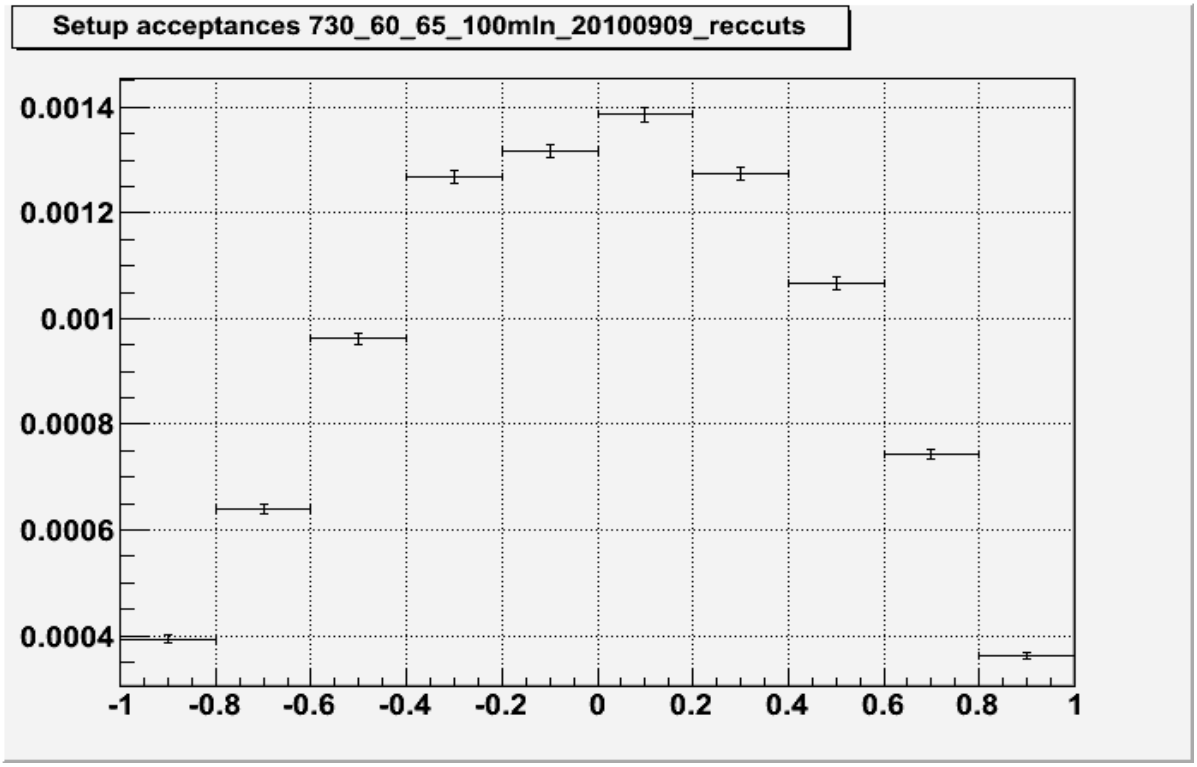

Figure 4: setup acceptances from results of a run