User's Guide for Showering/Calorimetry in MCFAST (version 4.1.0 and above)

Julia Yarba

November 10, 1999

1 Introduction

This documents serves as a user's guide and reference manual for the showering/calorimetry subpackage in MCFAST. This subpackage is completly integrated into MCFast Monte-Carlo pacakge: the user defines tracking systems and calorimeters and absorbers on equal basis, and MCFast takes care of converting particles to shower automatically, through ad-hoc interruption of the tracing processes. In addtion,energy losses in the bulk material of the calorimeters and absorbers are taken into account. As other features of MCFAST, the showering simulation is based on an extensive parametrisation scheme. That is, there is no detailed, particle by particle, description of an electromagnetic or hadronic shower. Instead, the energy densities due to the high energy showering processes within volumes are parametrised, and shower fluctuations are simulated globably rather than locally. Then in calorimeters these energy densities are mapped onto a user-controlled x-y or η-ϕ, or z-ϕ, or r-ϕ grid, and calorimetry hits are generated.

2 Method & algorithm

2.1 Geometry

Unlike particle tracing in MCFAST, the simulation of the shower propagation is intrinsically 3-dimensional.

To keep the program simple, only 3 volume types are recognized: BOX, TUBE and CONE. These shapes must be oriented in such a way that the symmetry axis is oriented along the Z-axis.

It should be pointed out that MCFAST, originally written in FORTRAN77, is undergoing an extensive modernization and migration to C++ programming language. This allowed to upgrade hit generation in calorimeters and thus singificantly improve the speed of the code, while the basic concepts and alogirithms for showering simulation have been kept unchanged.

However, the process of upgrading MCFast has not been completed yet, and thus only TUBE and BOX shaped volumes have been implemented in the C++ release so far (the task extends beyond showering simulation and is mostly related to the tracing part of the package).

These volumes are filled with a user-specified material, where showering occurs (electromagnetic and/or hadronic). Up to 40 such volumes can be defined in an MCFast job. In calorimeters, the segmentation is mapped to variables of physical interest, such as z or radius of particle, or the pseudo-rapidity η and azimutal angle ϕ, or x-y for forward calorimeters. Please note that the z-ϕ segmentation is valid for concentric calorimeters in central region only. The r-ϕ segmentation is valid only for tube-shaped forward and backward calorimeters. Currently, in order to select z-ϕ segmentation for a **central** calorimeter user should specify parameter **type** of EMCAL be 51; to specify r-ϕ segmentation for **forward/backward** calorimeters type should be 52 (see description of geometry definition below). Any attempt to set r-ϕ readout for cone-shaped forward/backward calorimeters will end up in resetting it to η-ϕ by the system automatically. No provision for longitudinal segementation is given, however, the user can implement that functionality by defining multiple calorimters, for a given η-ϕ or x-y range. For a central, solenoidal type detector, these calorimetric are usually concentric, while for a forward type geometry, they will be located at successive Z planes.

To initialize calorimetry/showering a user should define at least one calorimeter or absorber in the input geometry ASCII file in the following format.
include db/emcal.db
make EMCAL name [1] **shape type rmin@zmin rmin@zmax rmax@zmin rmax@zmax**
z_center z_length average_material_name active_material_name [2]
n_coor1_segms n_coor2_segms [3] **a_em b_em a_had b_had e/π_ratio nlayers**
for concentric calorimeters, or
include db/calorbox.db
make CalorBox name shape [4] **xmin xmax ymin ymax**
xmin_gap xmax_gap ymin_gap ymax_gap
z_center z_length average_material_name active_material_name
n_coor1_segms n_coor2_segms [5] **a_em b_em a_had b_had e/π-ratio nlayers**
for box-shaped calorimeters.
To specify absorbers use the following.
include db/absorber.db

[1] It is a character variable, its length can be up to 40 characters

[2] Average and active materials should be different for sampling calorimeters and same for homogenious calorimeters; both average and active materials must be define before they are first used; here it must be specified by **name** of type character

[3] it basicaly means ϕ-η or ϕ-z (if type=51), or ϕ-r (if type=52) segmentation; the first values should be the number of ϕ-segments, and the second one is for the number of η-, or z-, or r-segments

[4] it should be BOX; any combination of low case or capital letters is allowed

[5] this basibaly means the $x - y$ segmentation; the first value means the number of x-segments, and the second one is for the number of y-segments

make Absorber name shape type [6]
rmin@zmin rmin@zmax rmax@zmin rmax@zmax z_center z_length material_name
for concentric absorbers, or for box-shaped abosrbers
include db/absorberbox.db
make AbsorberBox name shape type
xmin xmax ymin ymax z_center z_length material_name

To turn the calorimetry off, a user should delete or comment out everything about calorimeters and absorbers in the input geometry file.

2.2 Showering algorithm(s)

At the initialisation stage, MCFast compute surface boudaries for these volumes. At the tracing stage, once such a surface is reached, the calorimetry subpackage returns an estimate of the mean free path length for the particle. For electrons/positrons and gammas it is simulated according the formula $f(x) = e^{-x/X_0}$, X_0 is radiation length. For hadrons $f(x) = e^{-x/\lambda}$, λ is absorbtion length (the mean free path for inelastic interactions). Based on this information and random number(s), the interaction inside the calorimeter is computed, if need be. Should that primary interaction occur, the trace identity switches from "elementary particle" to "shower", the core of that shower is propagated across the geometry by the MCFast tracing routing. This allows us to propagate a shower from one calorimter to the next. However, there is currently no provision for a magnetic field inside a calorimeter. Unlike particles, the core of the shower propagate in a straight line.

Upstream of this conversion point, the energy loss inside the material is taken into account, assuming that the particle behaves as a minimum ionizing particle (MIP) and the corresponding energy deposition is recorded, e.g., calorimetry hits are created. Showering switches on at the shower conversion point.[7]

Downstream of the conversion point, the fraction of "visible energy" is calculated for sampling calorimeters. After each step, the shower core position and energy/momentum are estimated and the control is given to the hit generation if necessary (see below). The package can optionally simulate energy deposition to the cells surrounding the shower core (Transverse profile "ON").

Note that a shower can spread over multiple volumes both longitudinally and transverselly (see description of "neighbouring" calorimeters below). Thus, even if a shower develops in an absorber, it still may produce hits in nearby calorimetrs (if any).

The electromagnetic shower longitudinal profile is described as

$$dE/dx = E_0 b \frac{(bt)^{a-1} e^{-bt}}{\Gamma(a)}$$

[6] it should be either 1 for central region or 2 for forward/backward region; in case 41 or 42 are specified only energy loss via minimum ionization and multiple scattering will be performed

[7] Defined as the point where the first hadronic or electromagnetic high energy interaction occurs

were

$$t = x/X_0, \qquad b \simeq 0.5, \qquad a = 1 + bt_{max},$$
$$t_{max} = 1.0 \times (ln(y) + C_i), \qquad i = e, \gamma, \qquad (C_\gamma = +0.5, \qquad C_e = -0.5)$$
$$y = E/E_c, \qquad E_c = \frac{800MeV}{Z + 1.2},$$

(Z is atomic number, E_0 is energy of incident particle).

For hadronic shower we use

$$dE/dx = E_0(wb\frac{(bt)^{a-1}e^{-bt}}{\Gamma(a)} + (1-w)d\frac{(du)^{c-1}e^{-du}}{\Gamma(c)})$$

were

$$\begin{aligned} t &= x/X_0 \\ u &= x/\lambda \\ a &= 0.6165 + 0.3183lnE_0 \\ b &= 0.2198 \\ c &= a \\ d &= 0.9099 - 0.0237lnE_0 \\ w &= 0.4634 \end{aligned} \tag{1}$$

When the shower transverse profile is turned on, the energy deposited on each step is distributed around the shower axis according to shower transverse profile (which becomes wider as shower moves deeper into a calorimeter).

The transverse profile for both electromagnetic and hadronic showers is described as

$$\frac{dE}{dR} = e^{(-R/A_1)} + A_2e^{(-R/A_3)}$$

In this formula, parameters A_1, A_2, A_3 depend of the material properties and on the depth the current shower depth. Note that the dependencies are different for electromagnetic and hadronic showers. The distribution of energy occurs in a plane perpendicular to the direction of motion of the shower core. Showers may have cross particular calorimeter boundaries, both longitudinally and transverserly. In this case energy deposition is also computed in the corresponding cells of those neighboring calorimeters.

The process stops if either :

- the particle (shower) leaves the calorimeter volume
- 95% (or more) of incident energy is deposited; in this case there is one more step, and the remaining energy is distributed according the transverse profile at (N+1)X_0

It is important to note that if a showering has started, the longitudinal path length of the shower core inside the calorimeter is recorded. If a shower core leaves the calorimeter volume still carrying some energy, this information is passed along to the simulation procedure for the next calorimeter. Thus, a calorimeter accepts incident showers as well as MIPs.

The deposited energy is smeared during the process of showering simulation according to the energy resolution formula

$$\frac{\sigma_E}{E} = \frac{a}{\sqrt{E}} \oplus b$$

where a and b are **input** parameters for the code.

For a sampling calorimeter, a visible fraction of energy is calculated (depends on the densities and proportion by volume of material from which a calorimeter is built).

2.3 Accounting for shower spread over multiple volumes

As mentioned above, showers may spread over multiple volumes both longitudinally and transvereselly. Longitudinally, it's basically a transport of the shower core from one volume to another. Transverse expansion over multiple volumes happens as the lateral profile becomes wider and a portion of deposited energy ends up outside of a given volume and inside a another volume nearby (if any), producing there hits (if the nearby volume is a calorimeter).

In the latest C++ release, a list of neighbouring calorimeters is created for each volume at the initialiuuzation stage. Note that only calorimeters are allowed in the list of neighbours while any volume, either a calorimeter or an absorber is allowed to have neighbours.

A calorimeter C is considered to be a neighbour N of a volume if either their z-sides overlap at least partially by radii, or inner surface of C overlaps at least partially in z with outer surface of N, or outer surface of C overlaps at least partially in z with inner surface of N. Any tracking devices between volumes are concidered "transperent" for showers.

If a portion of deposited energy is found outside a given volume, a search loop over all neighbouring calorimeters is performed in order to find out whether the portion of energy ends up in one of them. The loop breaks as soon as the right calorimeter is found in order to optimize the speed of the code.

In the FORTRAN77 release, a search loop is performed over the whole list of volumes in the system; the loop breaks once the right calorimetr is found (if any).

2.4 Comparison MCFAST vs GEANT

GEANT (with the same detector geometries and the same event input as for MCFAST) was used in order to compare how well data obtained with the detailed (e.g. GEANT) Monte Carlo matches the parametric simulation (MCFast).

We have made comparison MCFAST vs GEANT with a single particle incident a block of iron. A shower may start and/or develop in the first block and two more blocks of iron. All three blocks are of different thinknesses and η-ϕ segmentations.

In GEANT, standard energy cuts are used, i.e. 1 MeV for γs and electrons, 10 MeV for hadrons and muons (by energy cut they mean the energy at which the tracing of a particle of a given type stops). As hadronic shower generator we used GHEISHA which is the default option for GEANT.

The test is made for 10 GeV and 100 GeV γ and π.

The speed up factor of MCFAST over GEANT is found to be about 50 and 21 for 10 GeV γ and π, respectively, and about 475 and 200 for 100 GeV γ and π, respectively.

The shower conversion point generation is in a good agreement and the average shower transvesre profile generation is in a very reasonable agreement for both EM and HAD showers, comparing MCFAST vs GEANT.

The e/π ratio (which is an input parameter for MCFAST) may be tuned to correspond GEANT results.

Here it should be said that the speed of GEANT package depends pretty much on energy cuts selection. Also, hadronic shower generator choice affects the GEANT speed. For example, quick check indicatites that FLUKA package is 2 times slower that GHEISHA.

Also, those who want to compare MCFAST vs GEANT should keep in mind that GHEISHA generates e/pi ratio that depends on the particle/shower energy. FLUKA package does not produce this effect. It is not clear for the moment whether and how MCFAST should reflect this effect. The decision will be taken in a nearest future after more detailed comparison of MCFAST simulation results vs test beam data.

The shower conversion point and average transverse profile do not depend on GEANT hadronic shower generation choice, but the transverse profile may be slightly affected by the choice of GEANT energy cuts.

At last, we would like to stress that we do NOT intend to make any comparison, other than mentioned, of different GEANT hadronic shower generators.

2.5 Hit generation package

In calorimeters, energy depositions in cells are recorded as a shower propagates.

However, with the ongoing upgrate and migration of MCFAST to C++, hit generation is calorimeters go on different schemes in FORTRAN77 and C++ releases and thus application interface methods are also different.

It should be pointed out that **with the scheme implemented within the C++ release hit generation in calorimeters is significantly faster**. Thus the C++ release is strongly recommended, especially for those who are interested in calorimeters with fine segmentation.

2.5.1 Hit generation package for C++ release

This subpackage has been written completelly in C++ and takes full advantage of dynamic memory allocation.

A "Hit" refers to a total energy deposition in a cell(base class CalorSignal) and information on contributing tracks/particles, i.e. track number referring to the HEP event list along with the energy deposition via minimum ionization, electromagnetic, and hadronic showering processes caused by the particle/shower; the information on contributing particle is organized as an STL vector container (class CalorSignalMC derived from class CalorSignal). The information on hits is stored in a two-dimensional array of pointers to "Hit" objects, with the total number of such pointers being equal to the number of cells specified by a user for each calorimeter. The advantage of such approach is that as particle/shower propagates and shower develops, indeces of affected cells are calculated and hits be accessed directly in order to update information which in its turn allows to avoid multiple blowing and compressing of hit lists (though this approach is not the most economical one in terms of memory allocation, i.e. some of the hits do not contain any meaningful information). In each calorimeter, such array of pointers to hits is allocated dynamically once a particle/shower touches the calorimeter for the first time (if no particle/shower ever touches a given calorimeter, no array of hits is allocated in it). The selection of the two-dimensional array of pointers vs two-dimensional array of objects is related to the desire to create the right memory layout for users that may want to disregard the information on contributing particles/showers at the clusterization stage and deal only with energy deposition in cells.

2.5.2 Hit generation package for FORTRAN77 release

Unlike most of the original MCFAST written in FORTRAN77, the calorimetric hit generation subpackage was rewritten in C to take advantage of dynamic memory allocation unavailable in FORTRAN.

Energy deposition in calorimetric cells are organized a list of "hits". A "Hit" refers to a set of variables describing the cell coordinates and the type of energy (MIP vs E.M., vs. Hadronic). Once a particle/shower leaves a given calorimeter, hits in it are collected with respect to the segmentation of a certain granularity specific for every calorimeter, so that several hits produced in the same cell during showering collapse into one. At that stage, the track STDHEP number is associated to this collection of hits. NOTE : if during the process a showers parts touches a neighbouring calorimeters, hit collection occurs there, too. At the end of each event, hits pointing to the same cell produced by different particles are looked over in order find out segments in each calorimeter in which several particles may have produced energy deposition (this process is called hits consolidation). At this stage, a pointer to each track/particle that has contributed to a given hit/cell is also saved, along with the percentage of total energy caused by the particle. After this all the information is given to a user for future analysis.

3 Reference Section

The description of the calorimetry package methods and routines is given in the next section. The code(s) can be found in $MCFSRC/shower/src and $MCFSRC/showering/src and $MCFSRC/calorimetry/src areas.

The language for geometry definition structures is FORTRAN77.

3.1 Description of the User Data cards

3.1.1 control_shower

This is set of keywords in the MCFAST command file that allow a user to set some parameters which control shower generation.

- **shw_trans**
  Default value is **ON**. In order to further speed up the program and artficially reduce the number of hits generated, it is possible to create hits only the exact path of the shower-core. Only cells crossing that path get energy deposition. To set this option, set **shw_trans OFF**.

- **e_thresh**
  The default value is 0.05. With the current showering algorithm we stop tracing a shower if the remaining energy of the shower is 5% or less of the incident energy. But if a user wants to stop tracing of a shower earlier or later, he/she can do it by setting threshold as desired.

- **e_min_vis**
  The default value is 0.002 GeV. This is a minimum energy that can be recorded in the calorimeter cell (currently is set once for all calorimeters).

- **max_hit_step**
  The default value is 25. This is maximum number of hits with status SHOWERING (see below) that can be produced over each step as shower propagates. Correlated with **e_vis_min** (nhits = min((edep/e_vis_min),max_hit_step)); max_hit_step=0 is equivalent to shw_trans OFF. It is up to a user to change this parameter in order to speed up the code but the precision of shower transverse profile generation may be affected.

- **hit_generation**
  The default value is **ON**. This is a switch that may be turned **OFF** if a user does not want to generate hits in calorimeters. If it is OFF, **shw_trans** also turns OFF.

3.2 Description of the MCFAST calorimetry data structures

3.2.1 Geometry definition: emcal.inc & emcal_struct.inc

The include files can be found in $MCFINC/geom area.

integer type - 1= central, 2= forward
integer seg_type - 0 = η-ϕ, 1 = x-y, 2=z-ϕ, 3=r-ϕ segmentation
character*4 shape - can be either BOX, TUBE, or CONE
real rmin(2) - inner radius at zmin and zmax respectevely (for CONEs and TUBEs)
real rmax(2) - outer radius at zmin and zmax respectevely (for CONEs and TUBEs)
real xlimit(2) - left and right side edge x-coordinate (for BOXes)
real ylimit(2) - left and right side edge y-coordinate (for BOXes)
real xlimit_gap(2) - left and right side edge x-coordinate of inner gap (for BOXes)
real ylimit_gap(2) - left and right side edge y-coordinate of inner gap (for BOXes)
real zmin - left side edge z-coordinate
real zmax - right side edge z-coordinate
real eta_min - lower limit of η-coverage
real eta_max - upper limit of η-coverage
integer ncoor1 - number of x- or ϕ-segments
integer ncoor2 - number of y- or η-, or z-, or r-segments
integer material - pointer on the "average" material which fills the calorimeter
integer active_material - pointer on the "active" material (for sampling calorimeters)
integer nlayers - number of layers in a sampling calorimeter (default is 1) [8]
real siga_em - stochastic term in energy resolution formula for EM showers
real sigb_em - constant term in energy resolution formula for EM showers
real sigc_em - noise term in energy resolution formula for EM showers
real siga_had - stochastic term in energy resolution formula for HAD showers
real sigb_had - constant term in energy resolution formula for HAD showers
real sigc_had - noise term in energy resolution formula for HAD showers
real zerosup - threshold on visible energy in calorimeter
real c_coor1_size - cell size in x- or ϕ-units
real c_coor2_size - cell size in y- or η-, or z-, or r-units
character*40 name - device name

NOTE : The confused nomenclature "EMCAL" has historical roots. Hadronic showers we added later, the names of data structure did not changed for sake of expediency.

[8] this has been implemented for GEANT based code which is under development in parallel with MCFAST; the MCFAST itself does not use this information

3.2.2 Basic calorimetry hit structure for C++ release : CalorSignal.hh, CalorSignalMC.hh, CalorInfoTrack.hh

The header files can be found in $MCFSRC/calorimetry/src area.
class CalorSignal
{
......
public:
double getE() const - returns total energy deposition
virtual void clear() - zeros out energy deposition
......
protected:
double fEnergy - total energy deposition
......
}
class CalorSignalMC : public CalorSignal
{
......
public:
virtual void clear() - zeros out energy deposition and erases info on contributing particles/showers
const vector<CalorInfoTrack>& getInfoTracks() const - returns a constant referrence to a vector of InfoTrack objects (see below)
double getEMIP() const - returns energy deposition via minimum ionization
double getEEM() const - returns energy deposition via electromagnetic showering
double getEHAD() const - returns energy deposition via hadronic showering
double getEMIP(const int) const - returns energy deposition via minimum ionization by given particle
double getEEM(const int) const - returns energy deposition via electromagnetic showering by given particle
double getEHAD(const int) const - returns energy deposition via hadronic showering by given particle
......
private:
vector<InfoTrack> fInfoTracks - vector of InfoTrack objects
......
}
where
struct CalorInfoTrack
{
int trackID - track number referring to the HEP event list

double EMIP - energy via minimum ionization the track causes
double EEM - energy via electromagnetic showering the track causes
double EHAD - energy via hadronic showering the track causes
......
}

3.2.3 Basic calorimetry hit structure for FORTRAN release : cal_hit.h

The include file can be found in $MCFSRC/shower/src area.

int icr1 - x or ϕ cell number for the hit
int icr2 - y or η cell number for the hit
float e_mip - amount of energy deposited through ionization
float e_em - amount of energy deposited through EM processes
float e_had - the amount of energy deposited through HAD processes
int status - word defining the hit status, see below
int n_tracks - number of tracks contributing to this hit
cal_hit_tracks *info_track - pointer to set of track vs calorimetry hit,
where
cal_hit_tracks is a structure containing :
int tr_num - track number referring to the HEP event list
float percent_e_tot - percentage of the total hit energy this track contributes

The word **status** may be either :
EMPTY (or 0) - memory allocated, but no meaningfull information
NOISE (or 1) - noise component; no track information available
SHOWERING (or 2) - hit created as shower propagates; track information is not available
SINGLE_TRACK (or 3) - energy deposition is due to a single track; typicaly created when showering is completed for a given particle
INTEGRATED (or 4) - energy deposition may include contribution from more than 1 track; typicaly obtained at the end of each event.

3.3 Application Interface Routines and Methods

3.3.1 Application methods for C++ release

The header file is $MCFSRC/showering/src/CalorInfo.hh

The following methods can be invoked in a C++ version of **usr_analysis_** routine or any other user C++ routine (since user routines are still called from fortran code, please do not forget to put the _ at the end).

static CalorInfo* getCalorInfo()

- instanciates interface to calorimetry information (once per job !)

int numberOfCalorimeters() const

- returns number of calorimetrs in system

const vector<const DenseMedium*>& listOfCalorimeters() const

- returns vector of pointers to objects where minimum ionizing and showering processes happen and depostited energy is registered (i.e. calorimeters)

const int numberOfNeighbours(const DenseMedium* iCal) const

- returns number of neighbouring calorimeters for calorimeter iCal

Example of usage :

```
..........
#include "CalorInfo.hh"
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
int nNeighbours ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters.end() ;
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
    nNeighbours = info→numberOfNeighbours( *iter ) ;
}
.........
```

vector<DenseMedium*>& listOfNeighbours(const DenseMedium* iCal) const

- returns a vector of pointers to objects where energy deposition through minimum ionization and showering processes is recorded, located around calorimeter iCal

int numberOfHits(const DenseMedium* iCal) const

- return number of hits in calorimeter iCal

Example of usage :

....

```
#include "CalorInfo.hh"
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
int nNeigh ;
int nHits, nHitsInNeigh ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters.end() ;
vector<DenseMedium*>::iterator iterNeigh, iterNeighEnd ;
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
    nHits = info→numberOfHits( *iter ) ;
    if ( info→listOfNeighbours( *iter ).empty() ) continue ;
    iterNeighEnd = info→listOfNeighbours( *iter ).end() ;
    for ( iterNeigh=info→listOfNeighbours( *iter ).begin() ;
    iterNeigh!=iterNeighEnd; iterNeigh++ )
{
    nHitsInNeigh = info→numberOfHits( *iterNeigh ) ;
}
}
.........
```

CalorSignalMC* pGridOfCells(const DenseMedium* iCal) const**

- returns a 2-dimensional array of pointers to hits (objects of type CalorSignalMC; see description below) in calorimeter iCal; some of them are real hits (non-zero energy deposition), others are empty hits (zero energy deposition).

Example of usage :

```
..........
#include "CalorInfo.hh"
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
CalorSignalMC** pGrid ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters().end() ;
for ( iter=info→listOfCalorimeters().begin(); iter!=iterEnd; iter++ )
{
    pGrid = info→pGridOfCells( *iter ) ;
}
```

const char* segmentationType(const DenseMedium* iCal) const

- returns segmenation type of calorimeter iCal, either "Eta-Phi", or "Z-Phi", or "R-Phi" or "X-Y";
the system does NOT maintain other segmenation types

NOTE : z-ϕ and r-ϕ segmentation type is specific to c++ subpackage of mcfast, version 4.1.2 and above.

const int* numberOfCells(const DenseMedium* iCal) const

- returns number of cells, either in η-ϕ, or z-ϕ, or r-ϕ, or x-y units, depends on segmentation type, for calorimeter iCal

Example of usage :

```
..........
#include "CalorInfo.hh"
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
int nCells[2] ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters.end() ;
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
    nCells[0] = info→numberOfCells( *iter )[0] ;
    nCells[1] = info→numberOfCells( *iter )[1] ;
}
.........
```

const double* cellSize(const DenseMedium* iCal) const

- returns cell size either in η-ϕ, or z-ϕ, or r-ϕ or x-y units, depends on segmentaion type, for calorimeter iCal

Example of usage :

```
..........
#include "CalorInfo.hh"
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
double cellSize[2] ;
vector<const DenseMedium*>::const_iterator
```

```
    iter, iterEnd = info→listOfCalorimeters.end() ;
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
    const double* cell = info→cellSize( *iter ) ;
    cellSize[0] = *cell ;
    cellSize[1] = *(cell+1) ;
}
.........
```

const double stochasticTermEM(const DenseMedium* iCal) const

- returns stochastic term in the electromagnetic resolution of calorimeter iCal

const double constantTermEM(const DenseMedium* iCal) const

- returns constant term in the electromagnetic resolution of calorimeter iCal

const double stochasticTermHAD(const DenseMedium* iCal) const

- returns stochastic term in the hadronic resolution of calorimeter iCal

const double constantTermHAD(const DenseMedium* iCal) const

- returns constant term in the hadronic resolution of calorimeter iCal

const double ratioEMtoHAD(const DenseMedium* iCal) const

- returns e/π ratio for calorimeter iCal

const char* name(const DenseMedium* iCal) const

- returns name of calorimeter iCal as specified by user in geometry ASCII file

const double limits(const DenseMedium* iCal) const

-returns limits either in η-ϕ, or z-ϕ, or r-ϕ, or x-y units, depends on segmentation type, for calorimeter iCal. For example, if segmentation type is "Eta-Phi" limits[0] is η_{min}, limits[1] is η_{max}, limits[2] is ϕ_{min}, and limits[3] is ϕ_{max}

const double etaMin(const DenseMedium* iCal) const

- returns η_{min} for calorimeter iCal, if segmentation type is "Eta-Phi"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[0]** (see above).

const double etaMax(const DenseMedium* iCal) const

- returns η_{max} for calorimeter iCal, if segmentation type is "Eta-Phi"; otherwise retuns FLT_MAX. This method duplicates **limits(iCal)[1]** (see above).

const double phiMin(const DenseMedium* iCal) const

- returns ϕ_{min} for calorimeter iCal, if segmentation type is "Eta-Phi", or "Z-Phi", or "R-Phi"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[2]** (see above).

const double etaMax(const DenseMedium* iCal) const

- returns η_{max} for calorimeter iCal, if segmentation type is "Eta-Phi", or "Z-Phi", or "R-Phi"; otherwise retuns FLT_MAX. This method duplicates **limits(iCal)[3]** (see above).

const double rmin(const DenseMedium* iCal) const

- returns rmin for calorimeter iCal if segmentation type is "R-Phi"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[0]** (see above).

const double rmax(const DenseMedium* iCal) const

- returns rmax for calorimeter iCal if segmentation type is "R-Phi"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[1]** (see above).

const double zmin(const DenseMedium* iCal) const

- returns zmin for calorimeter iCal.

const double zmax(const DenseMedium* iCal) const

- returns zmax for calorimeter iCal.

const double xmin(const DenseMedium* iCal) const

- returns xmin for calorimeter iCal if segmentation type is "X-Y"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[0]** (see above).

const double xmax(const DenseMedium* iCal) const

- returns xmax for calorimeter iCal if segmentation type is "X-Y"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[1]** (see above).

const double ymin(const DenseMedium* iCal) const

- returns zmin for calorimeter iCal if segmentation type is "X-Y"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[2]** (see above).

const double ymax(const DenseMedium* iCal) const

- returns ymax for calorimeter iCal if segmentation type is "X-Y"; otherwise returns FLT_MAX. This method duplicates **limits(iCal)[3]** (see above).

There are also two methods that may be useful for the need of clusterization

vector<CalorCellAddress> listOfHits(const DenseMedium* iCal) const

- fills up and returns vector of addresses (2-dimensional, either η-ϕ or x-y) of cells with non-zero energy deposition

vector<CalorCellAddress>
listOfHitsBySortedEnergy(const DenseMedium* iCal) const

- fills up and returns vector of addresses (2-dimensional, either η-ϕ or x-y) of cells with non-zero energy deposition so that address of cell with maximum energy deposition occupies the first position in the vector and cell with minimum energy deposition is at the end of vector.

Here, structure CalorCellAddress is defined in CalorCellAddress.hh file and is represented by public data member int address[2], where address[0] is either η-, or x-, or z-, or $r-$-index of cell and address[1] is either ϕ- or y-index of cell. The header file can be found in $MCFSRC/calorimetry/src area.

However, user should be careful with using these two methods and keep number of calls to a minimum since each time a method is called a vector is created which is inefficient.

Example of recommended usage :

```
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
int nCells[2] ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters.end() ;
vector<CalorCellAddress> HitList ;
CalorSignalMC*** pGrid ;
double eHit, eTotal ;
int iHit, nHits ;
int address[2] ;
```

```
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
HitList = info→listOfHits( *iter ) ;
if ( HitList.empty() ) continue ;
eTotal = 0 ;
nHits = HitList.size() ;
pGrid = info→pGridOfCells( *iter ) ;
for ( iHit=0; iHit<nHits; iHit++ )
{
address[0] = HitList[iHit].address[0] ;
address[1] = HitList[iHit].address[1] ;
eHit = pGrid[address[0]][address[1]]→getE() ;
eTotal += eHit ;
}
}
.........
```

Example of **NOT** recommended usage :

```
..........
CalorInfo* info = CalorInfo::getCalorInfo() ;
int nCells[2] ;
vector<const DenseMedium*>::const_iterator
    iter, iterEnd = info→listOfCalorimeters.end() ;
CalorSignalMC*** pGrid ;
double eHit, eTotal ;
int iHit, nHits ;
int address[2] ;
for ( iter=info→listOfCalorimeters().begin(), iter!=iterEnd; iter++ )
{
if ( info→listOfHits( *iter ).empty() ) continue ;
eTotal = 0 ;
nHits = info→listOfHits( *iter ).size() ;
for ( iHit=0; iHit<nHits; iHit++ )
{
address[0] = info→listOfHits( *iter )[iHit].address[0] ;
address[1] = info→listOfHits( *iter )[iHit].address[1] ;
eHit = pGrid[address[0]][address[1]]→getE() ;
eTotal += eHit ;
}
}
.........
```

Note : User may iterate instead of addressing elements of HitList by index.

Bedides, similar methods are supplied to address calorimeters and their neighbours (if any) by index (as in array) instead of iterator. Methods to get information on a given calorimeter by supplying its index in the vector have the same names as described above. Methods that have "ForNeighbour" in their names are meant for getting information on neighbours iNeigh of a given calorimeter iCal, where iCal and iNeigh are integer indeces.

3.3.2 Application routines for FORTRAN77 release

CALOR_INIT

This routine initialize calorimetry hits if calorimeters are defined for the MCFAST job. In principle, this routine should not be called by the user since MCFAST takes care of the initialisation of public data structure. However, it is documented in case the user wishes - at his/her own risk - to reinitialize the event for some reason (same comment applies to the next 5 routines).

NOTE : Call to this routine turns off automatically if user selects integrated tracing (i.e. C++ release) by using **trace_integrated TRUE** in the command file.

Calling arguments : none

Language : FORTRAN77.

The routine is currently called at the stage of MCFAST initialization, by the MAIN_FORTRAN routine, after the geometry is read in.

INIT_ALL_CAL_HITS(NCAL)

Initialisation of the heap space to store hit information.

Calling arguments :

- Input NCAL is the number of calorimeters defined for the job. Type - INTEGER.

Language : C.

The routine is currently called by CALOR_INIT routine.

NOTE : actualy, CALOR_INIT duplicates INIT_ALL_CAL_HITS; the reason of implementing CALOR_INIT is not to put the whole calorimetry geometry data structure in MAIN_FORTRAN.

EMCAL_CLEAR_HITS

This routine loops over all calorimeters (if any) defined for the MCFAST job and clears out hits is each calorimeter on the event-by-event basis.

Calling arguments : none.

Language : FORTRAN77.

The routine is currently called by MCP_ANALYZE routine.

NOTE : Call to this routine turns off automatically if user selects integrated tracing (i.e. C++ release) by using **trace_integrated TRUE** in the command file.

INIT_CAL_HITS(ICAL)

This routine destroys all previous hits in a given calorimeter and preallocates calorimetry hits for the current event.

Calling arguments :

- Input ICAL is the calorimeter number to clear out hits in. Type - INTEGER.

Language : C.

The routine is called by EMCAL_CLEAR_HITS.

CONSOLIDATE_CAL_HITS(ICAL)

This routine collects all the hits produced by one or more particle(s)/shower(s) that may end up in the same cell, and builds the track information subarray, at the end of each event.

Calling arguments :

- Input ICAl is the number of calorimeter to handle hits in. Type - INTEGER.

Language : C.

This routine is called by CALOR_CONSOLIDATE.

CALOR_CONSOLIDATE

This routine loops over all calorimeters specified for the MCFAST job and does the hit collection in each of them, as the event precessing is completed.

Calling arguments : none.

Language : FORTRAN77.

This routine is called by MCP_ANALYZE.

NOTE1 : CALOR_CONSOLIDATE duplicates in some sense CONSOLIDATE_CAL_HITS; the reason of creation of CALOR_CONSOLIDATE is not to include the whole calorimetery geometry data structure in MCP_ANALYZE. NOTE2 : Call to this routine turns off automatically if user selects integrated tracing (i.e. C++ release) by using **trace_integrated TRUE** in the command file.

The following functions and routines may be called by a user at any stage of analysis, for example, from USR_ANALYSIS.

NUM_CAL_HIT(ICAL)

This function returns the number of hits in a given calorimeter.

Calling arguments :

- Input ICAL is the calorimeter to check the number of hits in. Type - INTEGER.

Language : C.
NOTE1 : As the function is written in C, the indexing starts from 0, not from 1.
NOTE2 : If called from a FORTRAN code, should look as NUM_CAL_HIT(%VAL(ICAL)).

INFO_CAL_HIT(ICAL,IHIT,ICR1,ICR2,EMIP,EEM,EHAD,NTRACK)

This routines provides information about a particular hit.
Calling arguments :

- Input ICAL is the calorimeter number the hit to look at belongs to. Type - INTEGER.
- Input IHIT is the number of hit to get information about. Type - INTEGER.
- Output ICR1 is x- or ϕ-cell number the hit belongs to. Type - INTEGER.
- Output ICR2 is y- or η-cell number the hit belongs to. Type - INTEGER.
- Output EMIP, EEM, EHAD are fraction of energy in a given hit produced due to minimum ionizing, electromagnetic showers, or hadronic showers respectevely. Type - REAL.
- Output NTRACK is the number of tracks that have contributed to the hit. Type - INTEGER.

Language : C.
NOTE : ICR1 and ICR2 are set to -1 if invalid hit.
NOTE : Since the routine is written in C, it is necessary to remember that indexing of calorimeters and hits starts from 0, not from 1.
NOTE : If called from FORTRAN code, should look as
INFO_CAL_HIT(%VAL(ICAL),%VAL(IHIT),ICR1,ICR2,EMIP,EEM,EHAD,NTRACK)

NUM_TRACKS_CAL_HIT(ICAL,ICR1,ICR2)

This function returns number of tracks that have contributed into a given cell (ICR1,ICR2) in a given calorimeter ICAL. Returns 0 if "empty" cell.
Calling arguments :

- Input ICAL is the calorimeter number. Type - INTEGER.
- Input ICR1 and ICR2 are indeces (address) of a given cell. Type - INTEGER.

Language : C.
NOTE : If called from FORTRAN routine, indexing should start from 0 and %VAL should precede the input arguments.

INFO_TRACK_CAL_HIT(ICAL,ICR1,ICR2,ITRACK,HEP,PERCENT_E_TOTAL)

This routine provides information on a given track ITRACK that have contributed into a given cell (ICR1,ICR2) in a given calorimeter ICAL.
Calling arguments:

- Input ICAL is the calorimeter number. Type - INTEGER.
- Input ICR1 and ICR2 are indeces (address) of a given cell. Type - INTEGER.
- INPUT ITRACK is the contributing track index (ranges from 0 to NTRACKS-1). Type - INTEGER.
- Output HEP is the track index in the HEP list. Type - INTEGER.
- Output PERCENT_E_TOTAL is the portion of total energy deposited in the (ICR1,ICR2) cell by the particle HEP (or by shower initiated by the particle HEP).

Language : C.
NOTE : If called from FORTRAN routine, indexing should start from 0 and %VAL should precede the input arguments while output arguments should be passed without %VAL.

E_MIP_CAL_HIT(ICAL,ICR1,ICR2)

E_EM_CAL_HIT(ICAL,ICR1,ICR2)

E_HAD_CAL_HIT(ICAL,ICR1,ICR2)

The above routines return energy deposition via minimum ionization, electromagnetic showers, and hadronic showers respectevely into a given cell (ICR1,ICR2) in a given calorimeter ICAL.
Calling arguments :

- Input ICAL is the calorimeter number. Type - INTEGER.
- Input ICR1 and ICR2 are indeces (address) of a given cell. Type - INTEGER.

Language : C.
NOTE : If called from FORTRAN routine, indexing should start from 0, and %VAL should precede the input arguments.

ENERGY_CAL_HIT(ICAL,IHIT)

This function returns the total energy of a given hit in a given calorimeter (or a sum of energy deposited via minimum ionizing, electromagnetic, and hadronic processes).

Calling arguments :

- Input ICAL is the calorimeter number a given hit belongs to. Type - INTEGER.
- Input IHIT is the hit number to get information about. Type - INTEGER.

Language : C.

NOTE : If called from a FORTRAN routine, should look as ENERGY_CAL_HIT(%VAL(ICAL),%VAL(IHIT)) with indexing on ICAL and IHIT starting from 0, not from 1.

SUM_ENERGY_CAL_HIT(ICAL,ICR1,ICR2)

This function returns the total energy deposited via minimum ionizing, electromagnetic, and hadronic processes in a given cell for a given calorimeter.

Calling arguments :

- Input ICAL is the calorimeter number a given cell belongs to. Type - INTEGER.
- Input ICR1 and ICR2 are indeces of x(or ϕ)-y(or η) cell to find energy deposition in. Type - INTEGER.

Language : C.

NOTE1 : If called from FORTRAN routine, should look as SUM_ENERGY_CAL_HIT(%VAL(ICAL),%VAL(ICR1),%VAL(ICR2)) with indexing starting from 0, not from 1.

NOTE2 : Variable names **ICR1** and **ICR2** are specific to version 2.6 and above; in version 2.5.1 they appear as **IPHI** and **IETA**.

NOTE3: Please be advised that in the version 2.5.1 **IETA** appears first in the list of arguments, and **IPHI** follows it - the "reverse" order of arguments ! Please pay attention when you make a call to the routine and check which version you are using !

SUM_E_TOT_CAL_HITS(ICAL,TRNUM)

This function returns the total energy deposited by a given track/shower in a given calorimeter. If TRNUM = 0, all energy in a given calorimeter is summed up.

Calling arguments :

- Input ICAL is the calorimeter number. Type - INTEGER.

- Input TRNUM is the track number in the HEP list. Type - INTEGER.

Language : C.
NOTE : If called from FORTRAN routine, should look as SUM_E_TOT_CAL_HITS(%VAL(ICAL),%VAL(TRNUM)) with indexing of ICAL starting from 0, not from 1; TRNUM should be actual track index in the HEP list, as in FORTRAN.

PRINT_ALL_CAL_HITS(ICAL)

This routine prints full information about all hits in a given calorimeters.
Calling arguments :

- Input ICAL is the calorimeters number. Type - INTEGER.

Language : C.
NOTE : If called from FORTRAN routine, should look as PRINT_ALL_CAL_HITS(%VAL(ICAL)) with indexing starting from 0, not from 1.

GET_CETA_CPHI(ICAL,IETA,IPHI,PHI,ETA)

This routine calculates coordinates of the given η-ϕ cell in η-ϕ units.
Calling arguments :

- Input ICAL is the calorimeter number a given cell belongs to. Type - INTEGER.
- Input IETA, IPHI are indeces of the η-ϕ cell. Type - INTEGER.
- Output ETA, PHI are coordinates of the cell center in η-ϕ units. Type - REAL.

Language : FORTRAN.
NOTE : User should remember that in our current scheme IETA corresponds coordinate #2 (just like Y).

CALOR_ENERGY(ICAL,ICR1,ICR2)

This function returns total energy deposition (due to minimum ionization, electromagnetic, and hadronic processes) in a given cell for a given calorimeter.
Calling arguments :

- Input ICAL is the calorimeter number the given cell belongs to. Type - INTEGER.
- Input ICR1, ICR2 are indeces of the x-(or ϕ)-y-(or η) cell. Type - INTEGER.

Language : FORTRAN.