ECAL Calibration with Data at High Energy
Code Manual

E. Carrera, Y. Gershtein
Florida State University

The calibration program takes a root file that has a tree with all the necessary variables. This root file is made separately by another piece of code which finds the two hardest superclusters in E_T and performs a Monte Carlo matching.

The code (Calib.cpp) has to be compiled aginst the ROOT libraries using the simple command:

```
g++ `$ROOTSYS/bin/root-config --cflags --glibs` -lMinuit -o Calib Calib.cpp
```

Two main data containers are declared in the *main()* function of the code. *allData* lodges, as it name implies, the information for all available data. *mcData*, on the other hand, only stores the information of those events that have at least one electron that can be classified as *well-known*, i.e. has energy between $700 - 1000$ GeV. Both of these containers allow differentiation between the *tag* status and the *probe* status of the electrons, which will have to be decided according to the physics involved in each event.

These two containers, together with the root file that contains the tree with the appropiate information, are passed to the function *GetData()*. Once these two containers have been filled by this function, they are passed to the function *GetMassBand* which performs the complete calibration, graphing, etc.

1 Function GetData()

This function fills the two main containers declared in the *main()* function. It first applies a set of global cuts which basically consist of requiring a back-to-backness cut of $\Delta\phi > cosPhiCut$ between electrons, a Monte Carlo matching threshold of $\Delta R < 0.2$, and the suppression of those events that fall in the detector cracks. It also limits the energy range for the electrons to be between 700 and 2000 GeV. Events passing all this set of cuts fill up the *allData* container. Here, no differentiation is being made yet between the *tag* and *probe* status and it is given randomly.

An event will be declared *well-known*, and will be stored in the *mcData* container if, at least, one of the electrons has an energy between $700 - 1000$ GeV, i.e if it has a *tag* electron. *GetData()* performs this check and also offers the possibility of restricting the region in rapidity.

If the two electrons qualify to be a tag electron, the one with lower energy is taken as tag. If only one of the electrons qualify to be a *tag*, the code makes sure it stores it in the *mcData* container with that status and also updates the order in which the data is stored in the *allData* container to reflect any changes in status.

2 Function GetMassBand()

This function is called by the *main()* function of the program and its basic objective is to create an invariant mass distribution plot with an uncertainty band due to the uncertainty in the determination of the correction function. It does so by taking both main containers, the *allData* container, where all the information is stored, and the *mcData* container, where the *well-known* events are stored. It also takes the number of pseudo-experiments that are going to be performed in order to determine the uncertainty band.

The necessary histogram variables are created along with some containers to store the information from each pseudo-experiment. Then, this function goes into a loop determined by the number of experiments chosen.

For each experiment a set of containers are defined. First, a copy of the *allData* and *mcData* containers are made. Also, containers *calibData* and *corrFmap* are created. The first one will store all the so-far calibrated events, and the other one will hold the informations about the η-dependent correction functions for each energy bin.

This last two containers, along with the copies from the main two containers are passed to the *CalibrateMe()* function which will be encharged of calibrating all the events available and of filling up the right data conatiners.

Once this calibrated information is available, the *GetMassHist()* function is called to get the invariant mass distribution histogram for that particular experiment. For each experiment, also a *MBmap* container is filled with the mass bin number and the corresponding number of events for that bin. A profile histogram is filled simultaneously. A vector (*MBvect*) of *MBmap*s is updated for each experiment with the corresponding *MBmap*.

Using the information stored in this last vector, a one dimensional histogram is made for the distribution of the number of events for each mass bin. That distribution is fitted to a Gaussian law and the one sigma upper and lower limits together with the central value are stored in a *MBLmap* which is later used to draw the final mass distribution plot with the uncertainty band.

Note that, for the first experiment performed, the Monte Carlo and the uncorrected invariant mass distributions are also gotten and that they are plot together with the corrected one.

3 Function CalibrateMe()

This function was wrote to accept uncalibrated raw data and to deliver calibrated data together with a container with the correction function information. The binning in energy (and in eta) is controlled by this function. This function works in two basic steps. First it calibrates the *well-known* data conatiner through the *CalibMC()* function, and then it calibrates the rest of the events by finding the correction function, for each energy bin, through the *FitInBins()* function and applying those corrections to calibrate the left-over uncalibrated data through the *CalibAlaMC* function.

This function accepts a *smear* switch which will turn on or off the smearing in the parameters of the correction function.

3.1 Function CalibMC()

CalibMC() calibrate those events that have at least one *well known* electron. If both electrons have this characteristic their E_T and E are set to the Monte Carlo truth values. However, if only one of the electrons serves as the *well-known* electron, the E_T for the *probe* electron is calculated through E_T balancing. Then the energy for the *probe* electron is recalculated using the new E_T value and considering a very small angular error.

Finally, the event is stored in the *calibData* container.

3.2 Function CalibAlaMC()

This function first looks for events that have not been calibrated, it then tries to find an electron that is suitable to start the calibration. It basically does so by checking the energy of the electrons and determining wheather or not it can find a correction function for one of them in the correction function container (*coefMap*). If both electrons serve this purpose, both are corrected. However, the one with less energy gets assigned as the *tag* electron.

If only one of them can be used to start the calibration, that one is labeled as *tag*. The calibration happens in a similar fashion as for the *CalibMC()* function, i.e., through E_T balancing. Note that, after storing the information in the calibrated data container, proper care has been taken to update the main data container to reflect the right labeling for the *tag* and *probe* electrons.

3.3 Function FitInBins()

The main purpose of this function is to fill out a container with all the information for the η-dependent correction function for each bin of energy. In particular, the *CFmap* is updated with the information that is gotten after the correction function parameters for the current energy bin have been gotten by the *FitMyEbin()* function.

3.4 Function FitMyEbin()

This function is the heart of the program. Its primary goal is to return the parameters and errors for the correction function for the current energy bin. It basically checks the container that stores the calibrated data to see if it finds an event that can be used to perform a fit in the current energy bin. Then it separates the events according to the type of tag they have, and for each type it fills a 2D histogram with the correction factor calculated as $F = E^{true}_{probe}/E_{probe}$, where E^{true}_{probe} is the calibrated and E_{probe} the uncalibrated value for the energy of the *probe* electron.

Each 2D histogram is then fitted to a Gaussian law for each rapidity bin using the auxiliary function *GetGausHist()*. The result is a one-disemsional histogram which stores the mean and the error on the mean (from the Gaussian fit) for each rapidity bin. Then, a vector *fitVect* is filled with the points and errors from the 1D histograms. This last task is performed by the function *FillFitVect()*.

Once the *fitVect* is gotten, it is used to perform a TMinuit fit through the function *TMFit()*. This function, as it is detailed later, is encharged of performing a fit to a parabolic correction function taking proper care of the *tag* electron types. The *TMFit()* basically return the parameters and errors for the correction function. An optional *GraphMe()* function is called to visually check the results of the fit.

Now that the different points for each type of *tag* for each η-bin have been gotten, an average of those points is peformed using the function *AveragePoints()*, and a *finalVect* container is filled with those averaged points. The function *TMFit()* is called again to perform the final fit which gives the final parameters and errors of the correction function. Once again, an optional *GraphMe()* routine can be executed to visually check the results.

4 Function GetMassHist()

This is an overloaded very simple function, whose primary goal is to deliver a 1D histogram with the invariant mass distribution. In order to calculate the mass values, it uses the function *Massing()*.

5 Function GetGausHist()

This bridge function receives a 2D histogram *_hin* and gives as output a 1D histogram *_hout* with the points comming from a Gaussian fit of the 2D histogram. For this purpose, this function relies on the *GausFitTH2InXbins()* function.

5.1 Function GausFitTH2InXbins()

This is a function that takes a 2D histogram and fit, to a Gaussian, the Y-axis data in bins of the X-axis, and returns arrays with the values of Y, Yerror, X, and Xerror.

The method depends a lot on how to determine the region in which the gaussian fit is going to be performed. For that reason, a couple of switches, which control (1) wheather to take the range based on the maximum of the distribution or not, and (2) to perform a simple (not gaussian) determination of the output values, have been introduced. By default, the fit range is calculated from the mean of the distribution a distance *r_width* (and *l_width*) times the RMS of the distribution. It also accepts a parameter that controls the minimum number of events that have to be present in order to make the fit. This proved to be useful in helping the convergence of the fits.

Additionaly, one can print the fitting results to visually inspect them. The storage directory is hardcoded so it has to be modified.

6 Function FillFitVect()

This function is called by the *FitMyEbin()* routine and it simply fills out a vector with the values comming from a 1D histogram. Note that the function differentiates the identity of each point that it stores depending on which class of *tag* electron was used. The output vector is then used to feed a TMinuit fit.

7 Function TMFit()

This is a function that controls the performance of the fit procedure in TMinuit. It is encharged of creating a TMinuit object and activating the proper parameters that are going to be used in the fit. The bool switch *refit* is specially important because it will activate the extra parameters used in the first fit (see Ecal Calibration note). The actual TMinuit iteration is carried on by calling the function *iterationTM()* which will use the *fcn* function to perform the fit and it delivers the results in the proper arrays.

7.1 Function iterationTM()

This function performs the actual TMinuit fit using the function *fcn*. Choosing the right *fcn* function is particular important. If the *fcn* function is set to be *zfcn*, the vector *fitVect* will be used to feed the minimization. Besides, choosing this function will invoke the usage of the extra paramenters (due to the correct propagations of the errors for each correction function).

If, instead, the function *refcn* is chosen, the vector *finalVect* will be used to feed the fitting iteration, and no extra parameters will be invoked.

8 Function GraphMe()

This a function that can be invoked to check, visually, the results of the fitting procedures. The directories where the plots are stored are hardcoded, so they need to be changed accordingly.