A Fast Monte Carlo Simulation of the DØ Luminosity Monitor

Joseph Tuggle,[4], Sung Hwan Ahn,[3] Michael Begel,[4]
John Hauptman,[2] Jae Won Park,[3] Richard Partridge[1]

[1] *Brown University, Providence, Rhode Island*
[2] *Iowa State University, Ames, Iowa*
[3] *Korea University, Seoul, Korea*
[4] *University of Rochester, Rochester, New York*

May 27, 2003

Abstract

We describe a program that has been written to simulate the Run II DØ Luminosity Monitor parameters. It is intended to be used as a first-order approximation of the effects of any changes to the Luminosity Monitor.

1 Introduction

Although DØGSTAR (DØ GEANT Simulation of the Total Apparatus Response) [1] provides an excellent, in-depth simulation of the complete DØ experiment, it is not always practical for studies involving only a single detector, especially one so small as the Luminosity Monitor [2, 3, 4]. In this note, we describe a simple, fast program called `lm_fastMC` that simulates the Luminosity Monitor to give information on the detector's acceptance, efficiency, and ability to find interaction vertices. Due to its simplicity, it is orders of magnitude faster than DØGSTAR.

2 Detector

The DØ Luminosity Monitor consists of two sides, which are 371 mm in diameter and cover a pseudorapidity range of $2.7 \leq \eta \leq 4.4$. They sit on the North and South endcap calorimeters at ± 140 cm in z. Each side is composed of 24 scintillating wedges arranged in a circle around the beam pipe. The wedges each cover a 15^o range in azimuthal angle ϕ. Attached to each scintillating wedge is a photomultiplier tube, which picks up a signal and emits photoelectrons. Collected data is divided into Luminosity Block Numbers (LBNs), each of which covers about 60 seconds of data taking.

The interaction region of DØ is in a 2 Tesla solenoid-generated magnetic field. Since the Luminosity detectors are located near the ends of this solenoid, the magnetic field is only 1 Tesla for them.

The Luminosity Monitor is a time of flight detector. It also reports information on how much charge passed through it. Since it detects time of flight, we can use that to determine the location in z of an interaction vertex. We also describe how the detector can be used to find the average (x, y) beam position during one LBN.

3 Program Flow

The Fast Monte Carlo program has three major components: initialization, event generation, and reconstruction. This section describes the flow of abstract data in the program. For information on the actual C++ coding, see Section 6.

3.1 Initialization

To start the program, use the following command:

`lm_fastMC < input file >`

where `<input file>` is the name of the file that contains the user-defined parameters (Sec. 4). The program will then read in all the parameters that were set in the input file to check their validity. Any invalid parameters will be reported. Next, the program initializes the detector geometry and all internal variables.

As described in Section 2, each side (North and South) of the Luminosity Monitor has 24 scintillation wedges. The program arranges them one next to the other in a circular shape around the beam pipe (Fig. 1). The wedges' z position and radial distance from the center of the beam pipe can be modified. The angle $\phi_0 = 0$ corresponds to the middle of the wedges labeled `NE05` and `SE05` in Fig. 1. Note that for the program (internally), the numbering starts at `00` (instead of `01`). Therefore, in the program, $\phi_0 = 0$ corresponds to the middle of the wedges labelled `NE04` and `SE04`.

3.2 Event Generation

We use Pythia [5] to generate low–p_T, hard QCD interactions, and accept only long-lived, charged particles. Pythia supplies us with the momenta of the particles; we create the interaction vertices.

A vertex is first generated in z using a gaussian random number generator. The z width of the beam $\sigma_{z\text{Beam}}$ is 25 cm. Next, the x and y components of the vertex are generated and smeared with a gaussian random number of width

$$\sigma_{xy\text{Beam}} = 0.0027 + 10.0 \cdot \left(\frac{z_{vertex}}{40.0}\right)^2 .$$

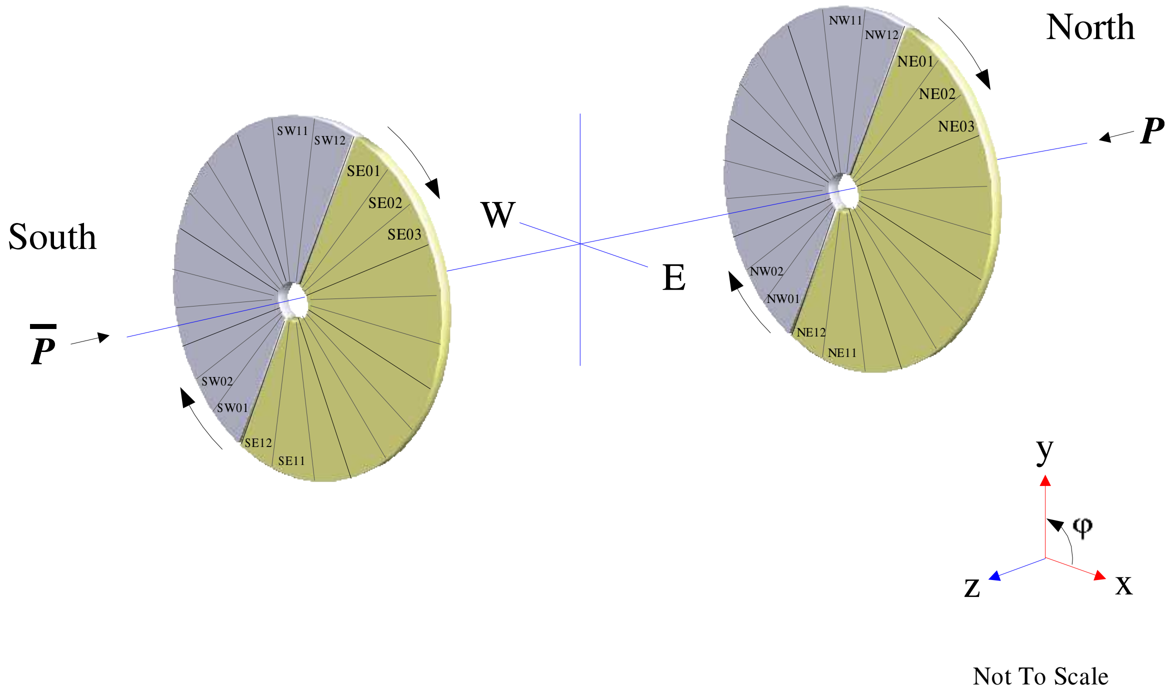

Figure 1: The arrangement and numbering of the luminosity wedges.

`lm_fastMC` then extrapolates the straight-line flight paths of the particles based on their momentum. If, when a particle reaches the z position of a detector (± 140 cm), it falls within the area of the detector, then the particle's time of flight is calculated and smeared with a gaussian of width $\sigma_t = 0.296$ ns. The time of flight reported by a wedge is the time of flight of the first particle to reach it.

Halo is simulated by picking a random wedge and assigning it a time of flight of $-140/30$ ns $=$ -4.67 ns, smeared by a gaussian of width σ_t. This would be the effect on the Luminosity Monitor of a particle passing through one side of the detector, then the other, instead of passing outward from the interaction region.

3.3 z Reconstruction

The program has a simple trigger that "fires" whenever there is at least one hit on both sides of the detector. This causes it to do a z reconstruction of up to three vertices for the event.

3.3.1 FPGA-Implemented Algorithm

Placeholder for the FPGA z reconstruction algorithm.

3.3.2 Least Squares Algorithm

The Least Squares z reconstruction algorithm first tries to fit all the time hits in the wedges to a single vertex. This fit is a least-squares solution as described in DØ Note #3968 [6]. We fit z and time according to the following equations:

$$z = \frac{c}{2}[\frac{\xi_N}{w_N} - \frac{\xi_S}{w_S}],$$

and

$$t = \frac{1}{2}[\frac{\xi_N}{w_N} + \frac{\xi_S}{w_S}].$$

In these equations, ξ and w are the weighted deviation from the time at $z = 0$ and the weight, respectively. The subscripts n and s refer to the North and South sides of the detector. ξ is calculated by

$$\xi = \sum_i \frac{t_i - L/c}{\sigma_i^2},$$

and w by

$$w = \sum_i \frac{1}{\sigma_i^2}.$$

The index i represents a wedge index.

After z and t are set, we calculate the confidence level of the fit based on its χ^2 and degrees of freedom. If the confidence level is below 1%, then we assume that more than one vertex exists, and the fitted z and t are simply the averages of the real vertices. Therefore, we split the vertex into two new vertices located at ± 10 cm and ± 0.25 ns on either side of the original vertex. The correct time-z association is based on minimizing χ^2. We loop over

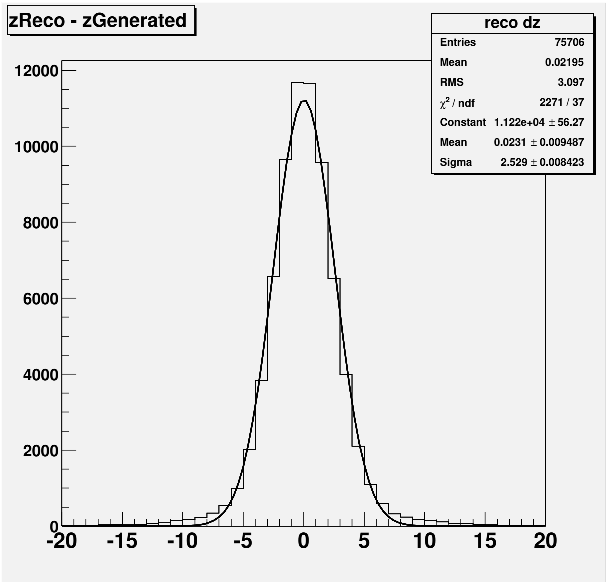

Figure 2: The distribution $\Delta z \equiv z_{\text{Reconstructed}} - z_{\text{Generated}}$ for 75,000 single-vertex events. It has a Gaussian width of 2.5 cm.

all possible vertex-hit combinations, and assign hits to the vertex that gives the combination a lower χ^2. Once the new vertices have hits assigned to them, their z and t are re-fit using the same algorithm as the one-vertex fit for each of them. We calculate a confidence level for the two-vertex system, and if this is now below 10%, we split the vertex with the higher χ^2. The reconstruction routine stops after reconstructing three vertices, or if the confidence level of a given fit is high enough.

We find that in single-vertex events, this algorithm is 95% accurate in finding a vertex within 9 cm of the original generated vertex. The difference between generated and reconstructed z position is shown in Figure 2. For two-vertex events, the algorithm's accuracy drops to just under 50% for finding both vertices within 9 cm. In a little over 30%of the events, one vertex is found to within 9 cm, and in the remaining events a three-vertex fit was incorrectly attempted. For true three-vertex events, the accuracy drops to under 10%.

3.4 (x, y) Reconstruction

If the beam is off-center, then the particles coming out of beam interactions, which are symmetric about the beam, will non-uniformly populate the Luminosity Monitor wedges. The overall effect is analogous to two circles, one offset from the center of the other. The resulting hit distribution will follow a cosine function in azimuthal angle ϕ, as seen in Fig. 3.

We can calculate the (x, y) beam position by using the parameters of the cosine function:

$$\frac{dN}{d\phi_k} = A + B\cos(\phi_k - \phi_0),$$

where k is the index of the k^{th} wedge. The fitted parameter ϕ_0 gives us the direction of the most hits, and is therefore the direction in which the beam is offset. A measures the average

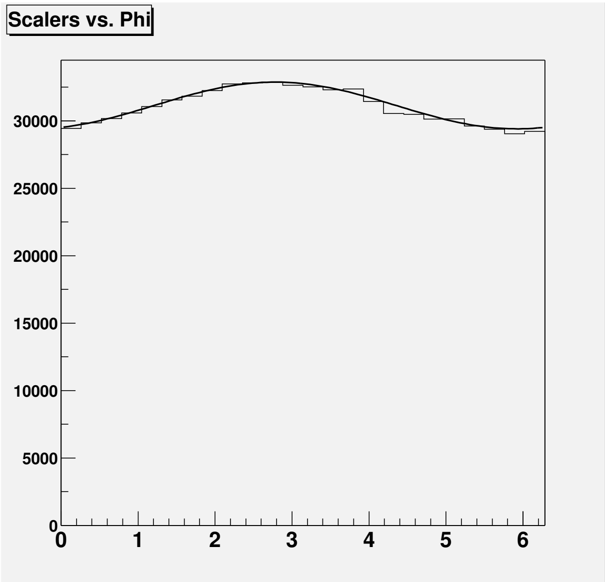

Figure 3: The hit distribution in Luminosity Monitor wedges caused by an off-center beam. This figure represents data from one side of the detector in one LBN, which produced 75,000 low-p_T events. The beam was offset in x and y by two centimeters.

number of hits, and B is the amplitude of modulation. The ratio B/A will give us a good measurement of the radial deviation of the beam, R_{beam}, in the form:

$$R_{\text{beam}} \approx \rho \cdot \left[\frac{B}{A}\right].$$

We currently take ρ to be 4.05 cm, however this number has not been carefully calibrated and may indeed introduce a bias, which is shown in the upper-right plot of Figure 4. Nevertheless, once we have R_{beam} it is a trivial matter to calculate x and y:

$$x_{\text{beam}} = R_{\text{beam}} \cdot \cos \phi_0,$$

and

$$y_{\text{beam}} = R_{\text{beam}} \cdot \sin \phi_0.$$

The accuracy of these fits depends greatly on the number of events used. For a good fit, at least 75,000 events (as a bare minimum) are needed. Figure 4 shows the results of (x, y) beam reconstruction for 20 LBNs, and 100,000 low-p_T QCD single interaction events per LBN.

4 The Input File

The `lm_fastMC` program accepts the name of an input file as its only command line parameter. This input file currently can contain 15 parameters, and the program will probably soon be modified to accept more. Parameters are **case-sensitive**. One invokes a parameter by typing its name, then the value (after whitespace). The required parameters are:

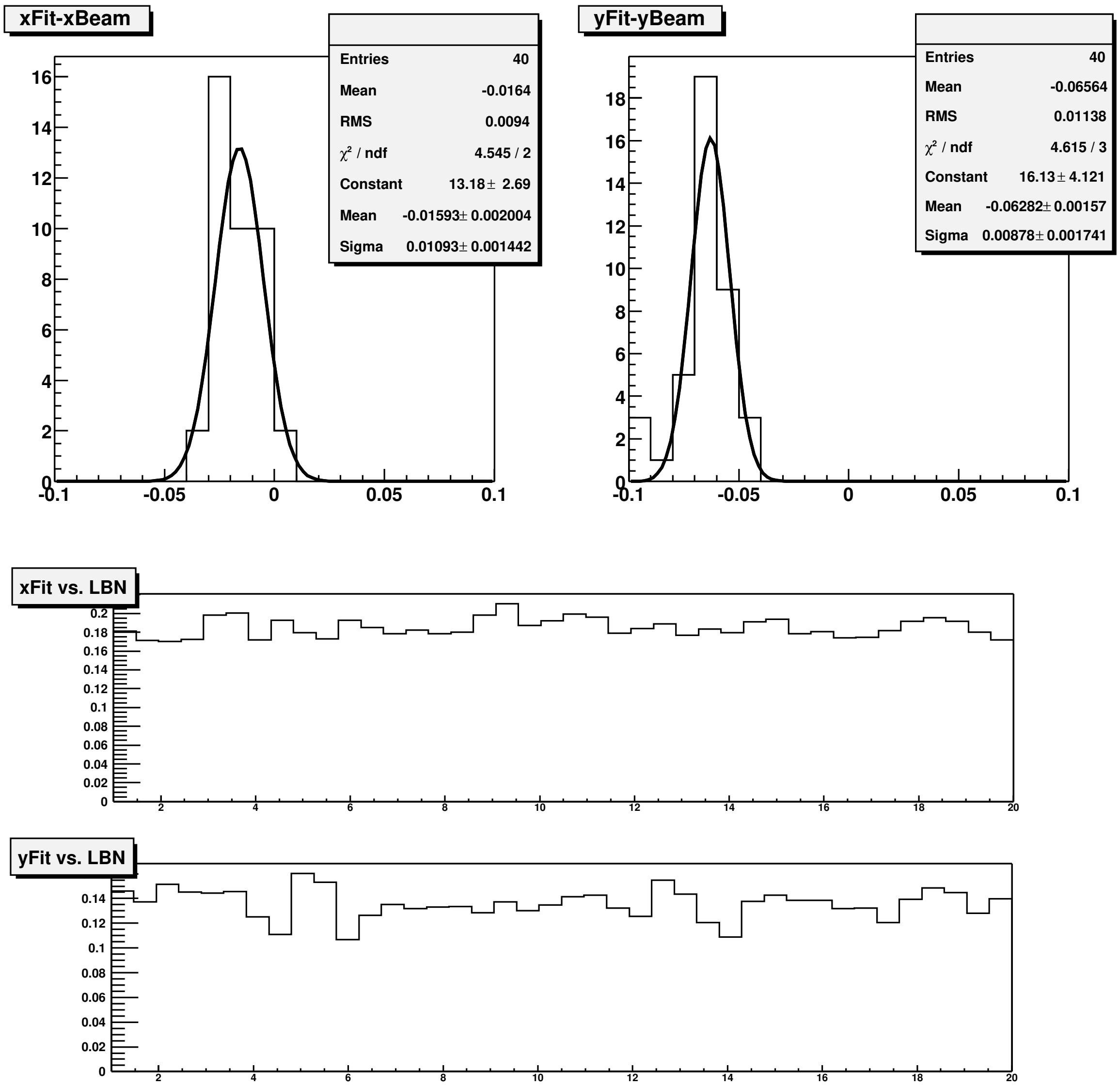

Figure 4: The fitted (x, y) mean beam position compared to the Pythia beam position (upper two figures); the x and y fitted beam position as a function of LBN for 100,000 QCD inelastic collisions per LBN. The Pythia beam was centered at $(x, y) = (0.2, 0.2)$ cm. There is obviously a bias towards fitting too low in y. The most likely source of this is the ρ parameter, which is used after the cosine function has been determined.

`PythiaProc`, `LBN`, `Events`, and `Seed`. One can use the `#` character to comment out whole lines. All of the parameters are described in detail below.

4.1 PythiaProc

This is a specialized parameter that is followed by the number of Pythia sub-processes to run, and then the ID numbers of the sub-processes themselves. These can be found in the Pythia manual. At least one sub-process is required.

4.2 LBN

This is simply the number of Luminosity Block Numbers to run. It is a required parameter.

4.3 Events

This is the number of events per LBN to generate. It is a required parameter.

4.4 Seed

This is a seed for the random number generator. It must be a negative number from -1 to -2,147,483,647 (inclusive). It is a required parameter.

4.5 LeastInt, MostInt

These are, respectively, the least and most number of Pythia interactions to generate per event. They must be between 1 and 3. If a range is specified, `lm_fastMC` will generate equal numbers of multiple interactions (*e.g.*, if one specifies 1 to 3 interactions per event, and 9000 events, then `lm_fastMC` generates 3000 of each type of event). Obviously, `LeastInt` should not be greater than `MostInt`. The program will filter out any initializations of that sort. By default, they are left at 1.

4.6 Separation

This is only useful if more than one Pythia interaction is being run per event. It specifies a minimum separation in z for multiple interactions. By default it is left at zero.

4.7 XBeam, YBeam

These set the average (x, y) location of the beam. They are each zero by default.

4.8 RateLum, PHalo, PBHalo

These three parameters relate to halo effects. `RateLum` is the rate, in kilohertz, of event generation. `PHalo` and `PBHalo` are the rates, in kilohertz, of proton and antiproton halo generation. If `RateLum` is zero, but `PHalo` and `PBHalo` are set, then they are changed to zero as well. Setting PHalo or PBHalo higher than RateLum is no different from setting them equal to RateLum. By default all three are zero.

4.9 NorthEff, SouthEff

At the time of this writing, these parameters have not been implemented, but they are placeholders for the efficiency of each side of the detector. It may be useful instead to track the efficiency of individual wedges. These parameters can be set, but they have no effect on the program.

4.10 Debug

This is a number from 0 to 3, a higher number indicating more verbose output. A debug level of 0 gives no extra output; level one gives information on when a function is being entered or exited; level two gives information on some actual values calculated in the program; and level three gives a full report on all important internal variables. Debugging is turned off after 10 events have been generated. By default, the debug level is zero.

5 Output

There is not yet an agreed-upon standard for what output lm_fastMC produces. In fact, there are two versions of some of the source files. The difference is in the number of histograms written to disk. The program uses ROOT [7] to create and fit histograms. These are written out to a ROOT file called fastMCOutput.root. It is relatively easy to output more histograms as they are needed.

Besides the histograms, lm_fastMC displays the following on standard output: debugging information, Pythia initialization and statistics, real *vs.* fake vertices for each LBN, and (x, y) reconstruction (North and South) for each LBN.

6 Programming

This section describes in detail the class structure and public methods of the C++ objects that go into the lm_fastMC program. There are four important classes: LMWedge, LMVertex, LMEvent, and LMParser. There is also a small base class called LMObject. The reconstruction routines are functions called LMReco_z and LMReco_xy. They are not contained in classes, but they are programmed as individual modules.

6.1 LMWedge Class

This class contains all the parameters that we wish to track for a Luminosity wedge . At the very least, these are pulse height and time. The idea is to deliver our actual (or simulated) electronics output to the wedge, and retrieve pulse height and time. The existing public methods are very simple: mainly "get" and "set" routines. One can set the charge and time information (from the electronics), pulse height cutoff, the pedastals, time uncertainty (σ_t), radial distance from the origin, z position, and scaler (number of hits). One can retrieve all of these values as well, except instead of electronics output, LMWedge gives the actual pulse height and time. We can expand the functionality of this class to track wedge geometry and positioning information.

6.2 LMVertex Class

This class is written with an eye towards vertex reconstruction. The only values that the user may set are the initial z position and time of an interaction vertex. There are a number of public methods used in vertex reconstruction. They are listed here.

6.2.1 AddWedge Method

There are two versions of this method. The first simply assigns a wedge to the vertex for the purpose of getting the initial single-vertex fit. No fit statistics (*i.e.*, χ^2 and degrees of freedom) are calculated in this version of the method. The second version acceps a wedge, and also the χ value for that particular wedge-vertex combination. This will cause LMVertex to track that information. For most purposes, the latter version of the method should be used.

6.2.2 Split Method

This is the method used when a single- or double-vertex fit is not good enough. One passes a pointer for the new vertex being created to this method, and it takes care of splitting. It creates two new vertices, at ±10 cm on either side of the original vertex, and sets the times (arbitrarily) to ±0.25 ns of the original time. Assigning the correct z-time combination is left to the calling function. The Split method takes care of unassigning any wedges that were associated with the original vertex.

6.2.3 DoFit Method

This is the method that performs the weighted deviation technique described in Sec. 3.3. It depends on having wedges from both sides of the detector assigned to the vertex.

6.3 LMEvent Class

This class is responsible for generating events and recording any hits in the Luminosity Monitor. Its constructor requires the IDs of the Pythia processes to run, the memory location of the detector, and the random number seed. It uses these for initialization of the event generation engine. There are two trivial methods: GetTrigger and PyStat. GetTrigger just returns whether both sides of the detector were hit in the event, and PyStat works exactly like the `PYSTAT` function in the Pythia manual.

6.3.1 GenerateInteractions Method

This is the heart of the LMEvent class. One gives this method the number of interactions desired, the (x, y) position of the beam, any desired z separation for multiple interactions, and a place to put the generated z components of the vertices (for use in determining how well the reconstruction algorithm runs). The method will then generate events using Pythia and record any hits in the detector.

6.3.2 GenerateHalo Method

If one wishes to generate halo, give this method the rate of event generation, as well as the rates of proton and antiproton halo (respectively). It will record any halo hits in the detector.

6.4 LMParser Class

This class is the interface between the `lm_fastMC` program and the user. The constructor of this class requires the command line parameters as they are in the main function: *i.e.*, `char **argv`. The constructor then parses the input file, and places any user-defined values in easy-to-access variables which can be retrieved by GetVariableName methods. If one wishes to add extra parameters to be read in through the input file, this is the class that needs to be modified.

6.5 LMObject Class

LMObject is the base class for LMEvent and LMParser. It currently only tracks debugging information, but if we desire any other information to be tracked, this is the place to put it. It is possible that no other information will be required, in which case this class may be phased out.

6.6 LMReco_z Function

This function requires the memory location of the detector and the location of an array of three verticies to fill. It returns the number of vertices that it found. LMReco_z runs as described in Sec. 3.3. It should be noted that the original Fortran code for this function used a routine to clear out halo effects. It was decided that this halo-clearing function should not go into the final C++ code until a better way to simulate halo is developed.

6.7 LMReco_xy Function

This function requires the memory location of one side of the detector (North or South), and a two-dimensional array into which it puts the fitted (x, y) position and the errors. The function fitting algorithm described in Sec. 3.4 is accomplished by linking to `ROOT` library functions. This function writes the `ROOT` objects it uses out to disk (*i.e.*, a histogram of the wedges' scalers and the function used to fit the histogram).

7 Conclusion

The `lm_fastMC` program does produce output that is satisfactory for a first-order approximation of the Luminosity Monitor. There are, however, areas which can be improved. It would not affect speed very much to take wedge geometry and positioning more into account. This would be extremely beneficial to the simulation, since in reality the sides of the detector are not symmetric.

Another improvement would be to calculate a better trajectory for the generated particles. Since they are charged particles in a magnetic field, they do not follow a straight-line path, as the program calculates. This would be a minor correction, but it could have a noticeable effect on detector acceptance.

The last major improvement would be to correct the ρ value used in the (x, y) reconstruction. This is a simple matter of calibration.

Once we use the fast Monte Carlo program to simulate major effects based on changes to the Luminosity Monitor, we can fine tune our settings using DØGSTAR. The `lm_fastMC` program will point us in the right direction.

References

[1] Y. Fisyak and J. Womersley, *D0gstar D0 GEANT Simulation of the Total Apparatus Response*, DØ Note 3191.

[2] C.-C. Miao and R. Partridge, *Study of the Run II Luminosity Monitor Counter Design*, DØ Note 3319.

[3] A. Lo, C.-C. Miao, and R. Partridge, *Luminosity Monitor Technical Design Report*, DØ Note 3320.

[4] C.-C. Miao, *The DØ Run II Luminosity Monitor*, Nucl. Phys. Proc. Suppl. **78**, 342 (1999), DØ Note 3573.

[5] T. Sjöstrand, Computer Physics Commun. 82 (1994) 74.

[6] S. H. Ahn *et al.*, *Reconstructing Interaction Vertices in X, Y, and Z using the DØ Luminosity System*, DØ Note 3968.

[7] René Brun *et al.*, *The ROOT User's Guide*, 2002.