LHCb-INT-2019-013
May 24, 2019

Development and deployment of a Delphes based simulation in the LHCb simulation framework Gauss

A. Davis[1], B. Siddi[2].

[1] *University of Manchester, Manchester, UK*
[2] *INFN Sezione di Ferrara, Ferrara, Italy*

Abstract

Faster alternatives to a full, GEANT4-based simulation are being pursued within the LHCb experiment. In this context, the integration of the Delphes toolkit in the LHCb simulation framework is intended to provide a fully parameterized option. This document presents the current status of the Delphes toolkit in Gauss, the LHCb simulation framework. In this integration, the particle transport performed by GEANT4 and subsequent mimicking of detector response and reconstruction has been replaced with a parametric response of the various detector elements. The implementation required significant changes to Delphes itself to constrain the particle transport inside the detector acceptance and to match the LHCb dipole magnetic field. The configuration of various parameterizations of resolution and efficiency is also tuned to provide a fully functional LHCb simulation. The output of the resulting fast simulation is formatted to be used directly in the LHCb physics analysis framework DaVinci.

1 Introduction

In LHCb, the interactions of particles with the detector elements are simulated in detail with the GEANT4 toolkit. This kind of simulation requires huge computing resources; therefore the generation of large Monte Carlo samples is usually slow and CPU-intensive. However, there are cases such as feasibility studies, detector design, and evaluation of systematic uncertainties in the final phases of analyses, where a high level of complexity is not required; a simplified approach based on the parameterization of the detector response is sufficient. The DELPHES framework [1] is designed to implement a parameterized simulation in an arbitrary High Energy Physics experiment. Starting from common event generator outputs, it performs a fast and realistic simulation of a general purpose collider detector. The particle energies are computed by smearing the initial visible particles momenta according to the detector resolution in a highly-customizable way. As a result most particles and detector effects can be reconstructed and reproduced. DELPHES was originally developed for general-purpose detectors as CMS and ATLAS.

A generalized parametric Monte Carlo tool was not available up to now in the LHCb experiment and the DELPHES toolkit has been chosen to provide this kind of functionality. The main steps to make DELPHES available in the LHCb software environment is to integrate it in the LHCb simulation framework, GAUSS [2], and bypassing the particle transport performed by GEANT4, in order to provide all the physical quantities and LHCb variables necessary for physics analysis. Moreover, a careful implementation of the DELPHES plugin allows parameterization of only parts of the simulation sequence, while keeping fully-detailed simulation for the rest. For example, an efficient and realistic hybrid simulation could be made by fully simulating charged particle tracking, and by a fast parameterization of calorimetry and particle identification. A correct LHCb detector parameterization is fundamental to obtain a correct physical output using DELPHES; this is achieved by looking at the full detector simulation as well as real data control samples, and extrapolating the parameters to be used. The DELPHES implementation in LHCb provides a faster way to generate high-statistics samples, but would also be extremely helpful in designing possible future detectors. Such studies are already underway for use in the LHCb Upgrade-II calorimeter, specifically in design aspects concerning detector granularity, timing resolution and layout.

This document presents the interface of DELPHES within the GAUSS and the subsequent processing sequence, following in the order of the DELPHES sequence itself. We present in the following sections the general implementation, followed by the description of the particle transport inside the LHCb acceptance, then the parameterization of resolution and efficiency for charged particle tracking; the efficiency and misidentification of particle identification, and the calorimeter. The fast simulation output has been formatted in order to be directly used in the LHCb physics analysis framework DAVINCI. We also present in this document the timing of the sequence and a few examples of the physics performance.

2 Delphes in LHCb

2.1 Implementation of Delphes Within Gauss

As Delphes is a stand alone package, the calling of it within Gauss must be done similarly to other packages. Therefore we try to adopt the same strategy of integration as Geant4 [3]. To this end, the Delphes github repository, found at`https://github.com/delphes/delphes` is mirrored within the LHCb Gitlab framework at `https://gitlab.cern.ch/lhcb/delphes-srcs`. This mirroring allows for individual patches necessary for the LHCb implementation to be applied on top of any outwardly maintained Delphes branch. For the purposes of development, we have been using the Delphes 3.4.1 branch, as it is still the latest release. At `cmake` configuration time, the path to a local installation of Delphes is searched for; if any local configuration is not found, then at build and compile time, the Delphes project is cloned from the `delphes-srcs` repository and the specific changes, discussed in Section 3.[1]

2.2 Output for physics analysis

In the context of the LHCb simulation framework, Delphes will substitute the detailed simulation performed by Geant4, and will give as output only high level reconstructed objects. In Figure 1 a schematic view of the chain from generation to analysis in LHCb is given. Offline analyses in LHCb are performed using the DaVinci software package [**?**]. The DaVinci application allows for reconstructed particles to be combined and kinematic quantities, such as invariant masses of decayed particles and their distance of flight. It takes as input reconstructed basic objects and it is possible to create new particles which are composed of said basic object and to perform candidate selection. Since the hits in the detector are not simulated, the reconstruction cannot be run; therefore the output from Delphes must provide directly objects expected by the analysis tool DaVinci. The minimal output that can be given to the analysis software is the `LHCb::ProtoParticle`[2]. This is a high level reconstructed object containing information for particles, such as:

- Links with tracks;
- At least one particle ID information;
- Link to calorimeter objects;
- Link to particle ID hypothesis objects.
- The relevant covariance matrices
- The track χ^2/ndf
- The ghost probability
- PID response objects

[1]The possibility of having a local installation of Delphes is accomplished by setting the `CMAKE_PREFIX_PATH` to point to the local installation of Delphes. NB: Compiling with the same environment as the rest of Gauss is strongly recommended to avoid compatibility/linking issues.

[2]We note that this is for the Run I and II event model. Should the event model change in the future, this structure will then need to change.

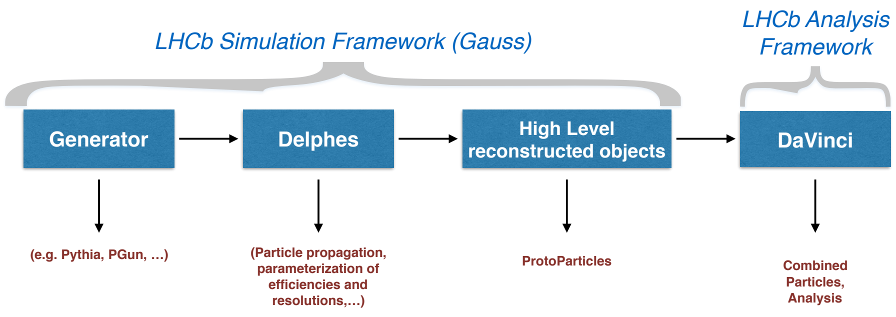
LHCb Simulation Framework (Gauss)
LHCb Analysis Framework
Generator
Delphes
High Level reconstructed objects
DaVinci
(e.g. Pythia, PGun, ...)
(Particle propagation, parameterization of efficiencies and resolutions,...)
ProtoParticles
Combined Particles, Analysis

Figure 1: Schematic view of the chain from generation to analysis in LHCb.

The standard DELPHES workflow concerning the module configuration is schematized in Fig.2.

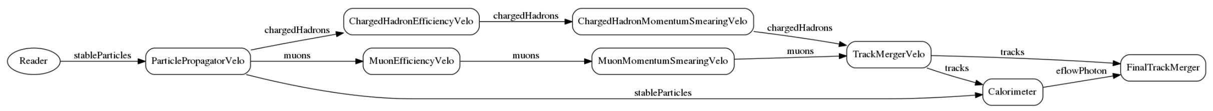

Figure 2: Schematic view of the modules configuration.

In the following sections a detailed description of the customized modules for LHCb will be presented.

3 Custom LHCb modules

In this section, a detailed description of the customized modules for LHCb are presented.

3.1 LHCb Particle Propagator

The default DELPHES particle propagator module has been written for a solenoidal magnetic field embedded in a cylindrical acceptance. It was therefore necessary to write new code to add functionality to match the LHCb acceptance. The propagation is made with a simple transport into a dipole field assuming a mean value for the shift of the transverse momentum component of the particles, known as the $p_x - kick$ or single bend point approximation; this is similar to what is done in the pattern recognition for transport between the VELO and T stations for long tracks, or vice-versa. Nevertheless, the point in which the $p_x - kick$ is applied has been parameterized as function of the inverse of the momentum of each particle, $Z_{\text{MagnetCenter}} \propto \left(\frac{1}{p}\right)^2$, as shown in Fig.3; The main principle of the particle propagator is to perform an acceptance check at various z of the LHCb detector. It proceeds in the following way:

1. Check if the final state particle produced is in the LHCb Detector "box" of allowed angles, e.g., $12 < |t_x| < 300$ and $12 < |t_y| < 250$;

2. an effective z coordinate of the magnet center "felt" by the charged particle is parameterized with a second order equation as function of $1/p$. This value can be given by a second order function or a fixed coordinate taken as external input;

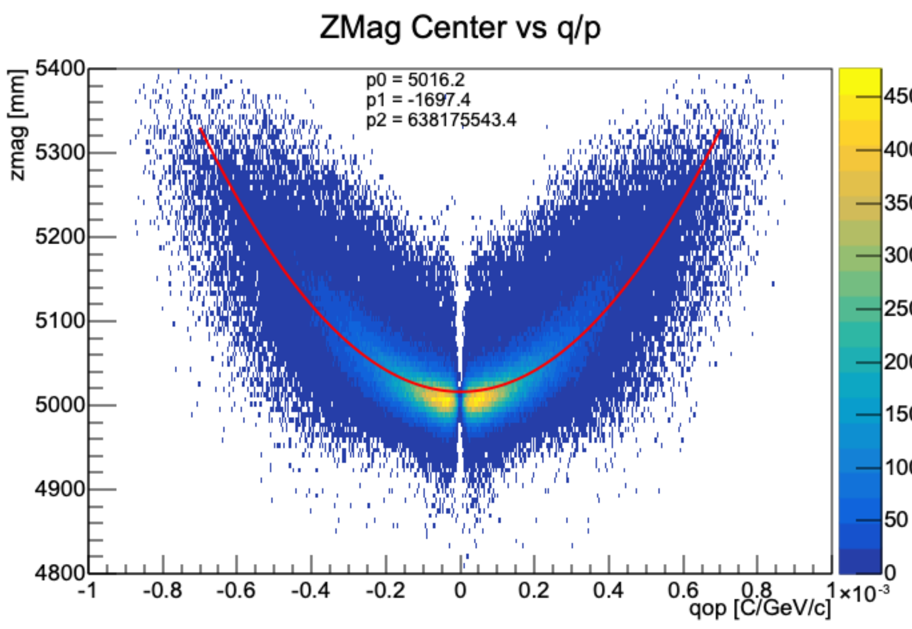

Figure 3: Magnet center parameterization.

3. particles are propagated until the corresponding magnet center;

4. A momentum kick in the transverse direction, i.e. p_x, is given to charged particle. The mean value has been computed comparing the p_x component of the momentum at two z, one before the magnet, i.e. at the end of the VELO, and one after, i.e. at the calorimeter entrance. The result is shown in Fig.4;

5. after the kick, another check is done to see if the particle is still in the acceptance of the LHCb detector at the end of the T stations.

The acceptance is then divided into three regions:

- In acceptance before the magnet and not after;
- In acceptance after the magnet and not before;
- In acceptance along the whole detector.

Particles are then saved in corresponding containers such as:

```
fOutputArray =
    ExportArray(GetString("OutputArray", "stableParticles"));
fOutputArrayUpstream =
    ExportArray(GetString("OutputArrayUpstream","upstreamParticles"));
fOutputArrayDownstream =
    ExportArray(GetString("OutputArrayDownstream","downstreamParticles"));

fChargedHadronOutputArray =
    ExportArray(GetString("ChargedHadronOutputArray", "chargedHadrons"));
fChargedHadronOutputUpstreamArray =
    ExportArray(GetString("ChargedHadronOutputUpstreamArray", "upstreamchargedHadrons"));
fChargedHadronOutputDownstreamArray =
    ExportArray(GetString("ChargedHadronOutputDownstreamArray", "downstreamchargedHadrons"));

fElectronOutputArray =
    ExportArray(GetString("ElectronOutputArray", "electrons"));
fElectronOutputUpstreamArray =
    ExportArray(GetString("ElectronOutputUpstreamArray", "upstreamelectrons"));
fElectronOutputDownstreamArray =
    ExportArray(GetString("ElectronOutputDownstreamArray", "downstreamelectrons"));
```

```
fMuonOutputArray =
    ExportArray(GetString("MuonOutputArray", "muons"));
fMuonOutputUpstreamArray =
    ExportArray(GetString("MuonOutputUpstreamArray", "upstreammuons"));
fMuonOutputDownstreamArray =
    ExportArray(GetString("MuonOutputDownstreamArray", "downstreammuons"));

fNeutralOutputArray =
    ExportArray(GetString("NeutralOutputArray","neutrals"));
fNeutralOutputUpstreamArray =
    ExportArray(GetString("NeutralOutputUpstreamArray","upstreamneutrals"));
fNeutralOutputDownstreamArray =
    ExportArray(GetString("NeutralOutputDownstreamArray","downstreamneutrals"));

fPhotonOutputArray =
    ExportArray(GetString("PhotonOutputArray","photons"));
fPhotonOutputUpstreamArray =
    ExportArray(GetString("PhotonOutputUpstreamArray","upstreamphotons"));
fPhotonOutputDownstreamArray =
    ExportArray(GetString("PhotonOutputDownstreamArray","downstreamphotons"));
```

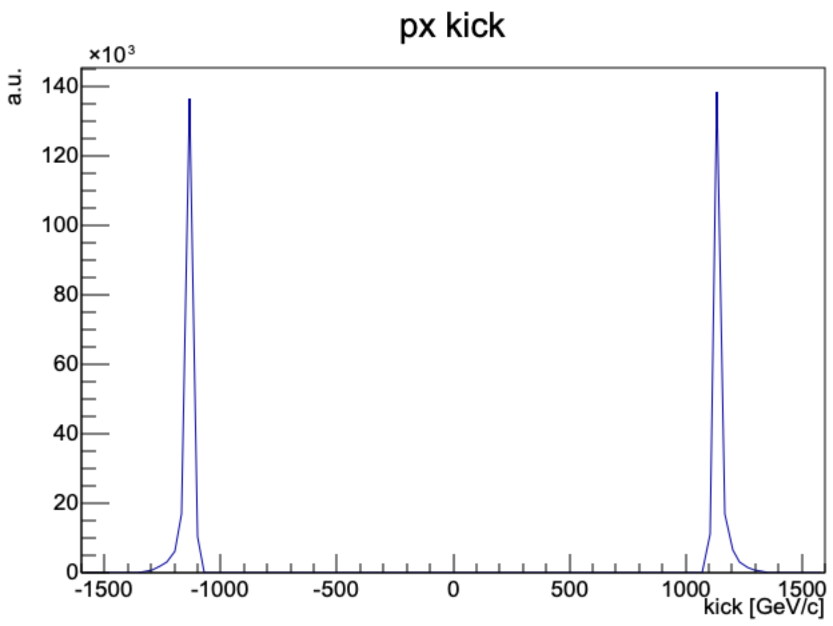

Figure 4: p_x kick distribution for charged positive and negative particles.

Correlations with the track slopes were searched for using full Monte Carlo simulation, but were found to be negligible. Should this no longer be the case, as may very well be with either mis-alignments, change of detector, or otherwise, the change in parameterization is easily set as a formula in the Delphes configuration card.

3.2 Efficiency

Tracking resolutions and efficiencies are obtained from the LHCb detailed simulation. To include these in Delphes, two dedicated Delphes modules have been developed which take as input histograms providing the parameterization for the LHCb detector. Efficiencies are taken as reconstructed tracks in the tracker acceptance. The resolution is defined as the root mean square of differences between reconstructed Monte Carlo tracks and the Monte Carlo truth. The kinematic variables adopted to fill both histograms are the track slopes $t_x = p_x/p_z$, $t_y = p_y/p_z$ and the inverse of the particle momentum $1./p$. Figure 5 shows the comparison between Delphes and detailed simulation output for the t_x, t_y and $1/p$ variables used for the parameterization. A specific parameterizations for each data taking period and detector conditions will be provided as full simulation becomes available. The procedure is described in Sec.3.4.

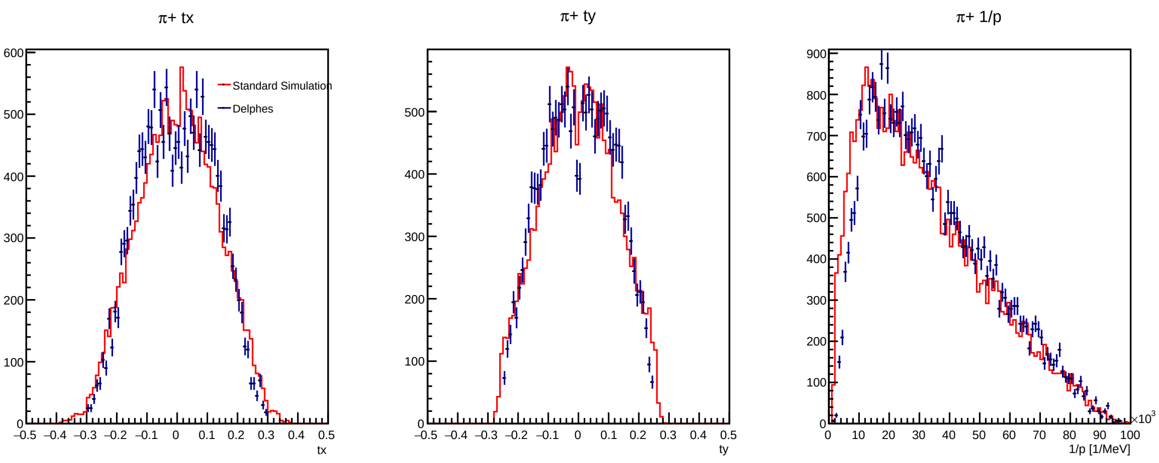

Figure 5: Track slopes $t_x = p_x/p_z$, $t_y = p_y/p_z$ and $1/p$ comparison between standard simulation and Delphes.

The original efficiency module has been rewritten in order to take as input two 3D histograms. The efficiency is obtained in the following way:

1. Input of two TH3D histograms (Reconstructed/Reconstructible) binned in t_x , t_y and $1/p$ (same z and acceptance cuts as in particle propagator);
2. X, Y, Z projections of Numerator/Denominator histograms;
3. Compute histogram ratio of each projection (hx_eff, hy_eff, hz_eff)

Then at the execution time the efficiency is applied doing:

1. Take candidate Lorentz momentum vector;
2. Compute t_x , t_y and $1/p$;
3. Search t_x, t_y and $1/p$ associated bin in hx_eff, hy_eff, hz_eff;
4. Take the efficiency in each bin;
5. Generate a uniform random number [0,1];
6. If random number $>$ the *or* of each efficiency $\rightarrow$ skip the event.

3.3 Resolution Smearing

As for the new efficiency module, the resolution smearing module has been written to take as input a 6D histogram binned as follow: Monte Carlo truth distributions of t_x, t_y, and p, differences between the reconstructed and the Monte Carlo truth of the same variables in the last 3 axis. At execution time the smearing is applied in this way:

1. Take candidate four-momentum vector as a TLorentzVector;

2. Compute t_x and t_y;
3. Find corresponding t_x, t_y and p bin in `THnSparse` histo;
4. Set `THnSparse` corresponding range in t_x, t_y and p;
5. Retrieve (True-Rec) `THnSparse` projections;
6. Compute RMS for each projection;
7. Generate Gaussian distributed random number with $\mu = var$, $\sigma = RMS$;
8. Assign new value to t_x, t_y and p.

3.4 Efficiencies and resolutions histograms

A custom python script is used to construct the efficiency and resolution histograms, respectively two `TH3D` for efficiency (reconstructed and reconstructible) and a 6D `THnSparse` for resolution, and to save them into a `ROOT` file. It takes as input a `TTree`, described in Sec.3.5, containing several track quantities computed at a given z coordinate necessary to fill the histograms. The workflow is the following:

1. reduce the trees with the cuts: same acceptance cut on t_x, t_y at the origin of the particle;
2. loop over the trees;
3. fill `THnSparse` and `TH3D`:
    - 6D `THnSparse` binned in true t_x, t_y, p and true-rec t_x, t_y, p;
    - two `TH3D` binned in t_x, t_y, p both reconstructed and reconstructible
4. write to file.

3.5 Custom Brunel algorithm

The `TTree` is created by the algorithm `TrackEffRes.{cpp,h}` embedded in Tr/-TrackChecker tools of the Rec package. (see branch `WIP_addTrackEffRes_forDelphes` in LHCb/Rec repository). The algorithm proceeds as follows:

1. loops over the `LHCb::MCParticles` container;
2. for each `LHCb::MCParticle` several quantities (e.g. p_x, p_y, $mass$) are saved at different z coordinate (this is done thanks to a track extrapolator tool) into a `TTree`
3. a check if the `LHCb::MCParticle` correspond to a reconstructible and reconstruced track the same quantities are saved into other two `TTrees`
4. the trees are saved into a `ROOT` file used as input to other algorithms

3.6 Implementation of the LHCb calorimeter in Delphes

As with the magnetic field, Delphes itself was designed to treat the calorimeter in bins of η and ϕ, exploiting the cylindrical symmetry of a general purpose detector. As this is clearly not suited to the LHCb geometry, as in the case of the particle propagation module, we opt instead to use this as a base to build our own calorimeter module which instead uses Cartesian coordinates. This was first implemented by Zehua Xu in the `CalorimeterLHCb` module, which has been simplified and included in the `delphes-src` repository. The great strengths of this module are two-fold: First, as the LHCb calorimeter is made of repeating cuboid towers, we can use one simple *tower* as a building block, and execute three simple loops within the definition of the card itself to create the entire LHCb calorimeter. Second, the module is completely replaceable without hurting the overall framework of Delphes or Gauss, meaning that if a better model of shower formation is possible, it is easily implemented.

For each photon that is propagated to the calorimeter face, the energy is smeared by the resolution of the LHCb calorimeter measured in test-beam data, $\frac{\sigma(E)}{E} = \alpha/\sqrt{E} \bigoplus \beta$, with $\alpha = 10\%, \beta = 1\%$ [4]. The energy smearing was only limited to one tower per hit, then the subsequent smearings between cells are done after the return of the tower itself from Delphes to Gauss in the `DelphesCaloProto` algorithm. In the current implementation, only photons are considered, with the possibility of extension to neutral hadrons foreseen with adequate tuning.

3.7 Python Configuration

In the following sections, we describe how we fulfill the criteria stated in Section 2.2. It is easiest to understand by walking through the logic of the Delphes sequence within Gauss, which is defined by the configuration of the python options. This is given by (and listed in `Sim/LbDelphes/options/LbDelphes.py`):

- `DelphesAlg`: Algorithm which takes as input HepMC particles, passes the collection to Delphes, which then performs the particle propagation, smearing and efficiency, then returns the result; this result is then packaged and output as `LHCb::MCParticles`

- `DelphesHist`: Histogramming for monitoring after `DelphesAlg`. This tool is historical, and has been deprecated in favor of `DelphesTuple`.

- `DelphesProto` converts the charged `LHCb::MCParticles` into `LHCb::ProtoParticles`

- `DelphesRecoSummary`, which adds to the TES the `LHCb::RecSummary` information needed for calculation of RICH and Muon response. In its simplest version, it draws a random number of tracks from a histogram defined via the Configurables and stores it into a newly created `LHCb::RecSummary` object; in the future it will extended to randomize the number of reconstructed tracks (`nTracks`) as a function of the number of charged particles at generator level;

- `DelphesParticleId`, which computes the detector response of the RICH and Muon detectors and loads it in the `LHCb::RichPID` and `LHCb::MuonPID` objects; the

Configurable of `DelphesParticleId` allows to define the paths in the filesystem to the TensorFlow models modelling the Rich detector differential log likelihoods, the efficiency of the `isMuon` criterion and the differential log-likelihoods obtained from the Muon system; the description of the neural network models and the interface between Gaudi and discussed in a dedicated note [5];

- `ChargedProtoParticleAddRichInfo`, which loads the RICH log-likelihoods from the `LHCb::RichPID` objects to the `LHCb::ProtoParticle` as an `additionalInfo` flag
- `ChargedProtoParticleAddMuonInfo`, which loads the `isMuon` and Muon log-likelihoods from the `LHCb::MuonPID` objects to the `LHCb::ProtoParticle` as an `additionalInfo`
- `ChargedProtoCombineDLLsAlg`, which combines the Rich, the Calorimeter (if available) and the Muon log-likelihoods into the Combined DLLs, commonly used at analysis level to define particle identification criteria; as the algorithms `ChargedProtoParticleAddRichInfo` and `ChargedProtoParticleAddMuonInfo`, `ChargedProtoCombineDLLsAlg` is not specific to Delphes, but is developed and maintained for the reconstruction and selection frameworks (Brunel and DaVinci), and listed here to reproduce the behaviour of the reconstruction step which parametric simulation allows to skip
- `DelphesCaloProto`: This algorithm is responsible for converting the neutral MC particle into usable calorimeter deposits, followed by the production of neutral protoparticles for use in DaVinci
- `DelphesTuple`: Algorithm which creates an NTuple of the results for use in either tuning or MCParticle level studies
- `BooleInit`: provide the ODIN information for Delphes. This will be deprecated as the algorithm will be moved to the LHCb package and will be accessible across projects.
- `PgPrimaryVertex`: Provide a PV as with ParticleGun simulations. This package was moved from the Brunel package to the LHCb package for use across projects.

This sequence takes all HepMC particles and transforms them to the particles used in final analysis level simulation.

In the following sections, we describe first the implementation of the Delphes configuration, then describe each step that the generated particles see. This follows the logic of the Delphes sequence, and provides the most natural description of why each algorithm is where it is.

3.7.1 Automatic Generation of Delphes Configuration Card File

Delphes is configured via a tcl file called a *card*. This card file defines first the sequence of execution, then the individual configurations of each of the steps. This includes the particle propagation, resolution and efficiency smearing, as well as the definition of the calorimeter geometry to be used. Instead of forcing all LHCb users to learn tcl, we factor the card definition into a python configurable option, housed in `Sim/LbDelphes/python/LbDelphes/LbDelphesCardTemplates` called `DelphesCard`. This file allows one to first configure directly a simple card file for use in all manners of simulation, and (2) allow for custom modification for expert studies. The default configuration looks like:

```
from LbDelphes.LbDelphesCardTemplates import DelphesCard
c = DelphesCard.DelphesCard(name = 'delphes_card',
                            year = '2012',
                            magPolarity='up',
                            efficiencies=True,
                            resolutions=True
                            )
#configure the card here for expert users
#c.modules['ECAL'].custom_xy_binning(...)
c.FinalizeCard()
```

This is accomplished using python template replacement on the template files included in the same directory. Three templates are used, namely `module_base.tcl`, which is the general base input for any DELPHES module, `module_merger.tcl`, which is special, as it takes multiple inputs and concatenates them into one output, and `module_lhcb_ecal.tcl`, which is necessary to form the ECAL response. We keep track of the order that the modules are added within the sequence, to ensure that the python configuration creates a self-consistent configuration card. We have also left the possibility of extending the card maker to include options for the future, for instance tracker-only requests would immediately eliminate the need for the calorimeter configuration.

This framework is then completely customizable for expert users, as a new DELPHES algorithm can be added to the externally compiled DELPHES build, then configured by the same python options given by the card creator.

4 DelphesAlg

With the specific DELPHES modules declared, we now discuss the remaining parts of the sequence. The first is `DelphesAlg`, which is the only algorithm which ever interacts with DELPHES. The algorithm starts by taking HEPMC events which are written to the transient event store (TES) and filling them into the input arrays necessary for DELPHES. At this point, the MC primary vertex is found one of three ways[3]:

1. One searches for the beam particles associated to the event, and takes their decay vertices, if they exist.
2. The signal process vertex stored in HepMC is used directly
3. Take the production/end vertex of the particle with barcode 1. This particle is special.

If no PV is found, an error is thrown.

[3]With thanks to Gloria Corti

Once all arrays are filled, the Delphes object is asked to process the task configured by the tcl card. This uses all the information presented in Sections 2.1-3.6. Once this is completed, we import all results from the final output location of Delphes back into Gauss. From here, all `LHCb::MCParticles` and `LHCb::MCVertices` are written to the TES.

One specific note for `DelphesAlg`, the variable `M2` in the Delphes Candidate class is constant both before and after the interaction with the `DelphesFactory`, hence the particle propagation. This provides a good variable to attach the key of a particle to, which is necessary to follow the particle before and after propagation. This is also used as the key in the keyed container of an object.

5 DelphesProto

`DelphesProto` is the algorithm dedicated to build the minimal output for charged particles, to be given as input to DaVinci analysis framework. This consists in `LHCb::ProtoParticles`, `LHCb::Tracks` with at least one `LHCb::State` and basic PID information that will be filled after with dedicated algorithms.

`DelphesProto` algorithm take as input `LHCb::MCParticles` and `LHCb::MCVertices` created in the previous step and fill the corresponding quantities into the higher level objects. Covariance error matrix associated to the track is also parameterized. This has been done with a lookup table in which each matrix element has been averaged as function of $1./p$. A very similar procedure is used to fill other track properties such as ghost probability, fit likelihood, track χ^2 and number of degrees of freedom. The track fit information and the covariance matrix are filled with lookup table generated by a standalone script described in Sec.5.1. This last step is done retrieving from the table the row corresponding to a particular $1/p$ range of the particle, then for each entry of the matrix the mean value and the error is used to smear the value to be given to the track. All objects related to tracking are filled and stored in the corresponding TES containers the standard LHCb analysis tools can be used as they are proving the feasibility of the generation to analysis chain. Figure 6 show the invariant mass of two reconstructed pions through Delphes, obtained with standard LHCb analysis framework. Once all objects related to tracking are filled (covariance matrices, χ^2/ndf), the standard LHCb analysis tools can be used: this proves the feasibility of the generation to analysis chain. Figure 6 show the invariant mass of two reconstructed pions through Delphes, obtained with standard LHCb analysis framework. To perform a direct comparison with the standard simulation the correct parameterization has to be performed and validated. Figure 7 presents a comparison of the reconstructed invariant mass of the decay $D_s^+ \to \phi\pi^+$ with $\phi \to K^+K^-$ as obtained from the 2016 calibration samples and from Delphes. The plot was obtained using Bender and DaVinci to process the simulated and real data samples, respectively. While the simulation slightly underestimates the error on this mass resolution, the level of agreement is already impressive.

5.1 Covariance and track information lookup tables

Two custom python scripts has been written to create a `.dat` file containing the mean value and the error of each element that goes inside the lookup table as function of $1/p$.

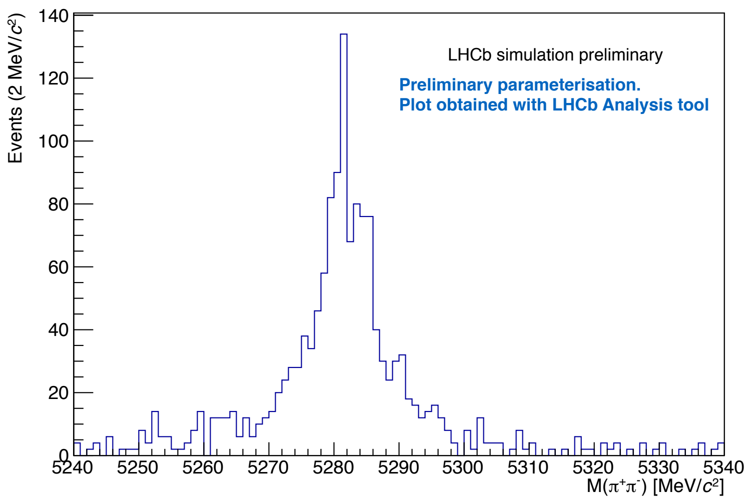

Figure 6: Invariant mass of $\pi^+\pi^-$ combination of the decay $B^0 \to \pi^+\pi^-$, obtained using final reconstructed objects from the DELPHES sequence in GAUSS to LHCb analysis framework DAVINCI.

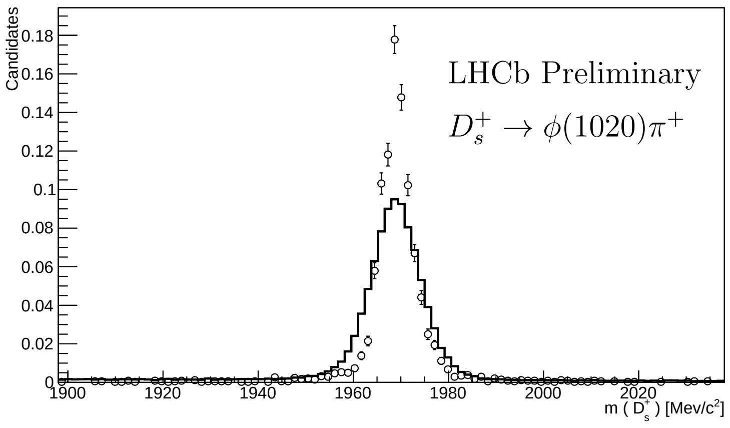

Figure 7: Comparison between the mass resolution of the decay $D_s^+ \to \phi(\to K^+K^-)\pi^+$ as obtained from the 2016 calibration samples (solid line) and with the DELPHES smearing (markers with error bars). To reduce the kinematic differences between the two samples, only D_s^+ candidates with p_{T} between 3 and 4 GeV/c and η between 2.5 and 3.0 have been used in the fit.

They take as input a `ROOT` file with a `TTree` containing several track quantities computed at a given z coordinate necessary to fill the table.

The structure of the covariance lookup table is the following:

$$\text{min } 1/p \mid \text{max } 1/p \mid cov_{0,0} \mid \sigma(cov_{0,0}) \mid cov_{0,1} \mid \sigma(cov_{0,1}) \mid \ldots \mid cov_{n,n} \mid \sigma(cov_{n,n})$$

The track parameter lookup table structure instead is:

$$\text{min } 1/p \mid \text{max } 1/p \mid \chi^2 \mid \sigma(\chi^2) \mid nDOF \mid \sigma(nDOF) \mid \mathcal{L} \mid \sigma(\mathcal{L}) \mid$$

In order to compute the entries, the script proceeds in this way:

1. Create and fill a `TProfile` for each element;
2. For each bin get the mean and the error;
3. Save each entry into a .dat file

The scripts are extendable to have the possibility to include other track parameters.

6 DelphesCaloProto

The output of Delphes requires more work to make the proper calorimeter object. There are three major steps in the calorimeter protoparticle maker:

1. Generate all the showers for the photon individually, ignoring overlap in cells
2. For each generated shower, sum the energy of overlapping cells to form the total calorimeter response
3. Now that all contributing cells have the correct energy, compute the covariance matrix and the barycenter of the cell, as calculated in the LHCb framework

This defines a maximum of 3 loops over all particles. Should the clustering and overlap be changed to be performed in Delphes itself, the number of loops over particles would be reduced to one, but with a lack of clarity of overlap in cells.

6.1 Smearing of energy to neighboring cells

The smearing of energy to neighboring cells is accomplished by Monte Carlo integration. We assume a simple model that the Molière radius R_M well describes the calorimeter shower response. Under this assumption, concentric circles of $1, 2, 3.5R_M$ enclose 90, 95 and 99% of the energy of the shower. Therefore, we draw the circles of these Molière radii around the x, y position of the particle at the calorimeter face. This smearing is applied irrespective of whether or not the shower is confined to one region of the calorimeter. At this point, the Delphes calorimeter module has already smeared the position of the photon hit with respect to the true MC hit. This can be disabled, but has shown good agreement with full simulation for Upgrade II ECAL studies. It is also important to note here that the showering does not take into account the angle of the photon with the calorimeter face. To do this, one should take into account the depth of the calorimeter, then draw two ellipses with the center of each ellipse being the particle trajectory and the other parameters defined by the intersection of the cylinders of said Molìere radii with the planes of the calorimeter. This naive approximation completely ignores all structure of the calorimeter itself, but can be tuned.

We then include all cells where the energy has been smeared, a cluster, into the next step.

6.2 Overlap of clusters

For this part, we rely on the fact that energy is a scalar quantity, and can therefore be summed. We then simply take all clusters from the previous step to check for overlapping towers. If the towers overlap, both clusters reset the energy of that tower to be the sum of the energies of each individual tower. This process is done until all clusters have been processed.

6.3 Formation of covariance matrices and protoparticles

Once all clusters have been formed, one can form all the necessary ingredients for the neutral protoparticles. This relies on a bit of a complicated interaction with the TES at this point, but when moving to `Gaudi::Functional`, it should become much simpler. The first step is to assign all the `LHCb::CaloCellIDs` to their corresponding `LHCb::CaloDigit` with the correct energy associated. These are then put into the TES and accessed from there.

Once the digits are formed, the barycenter and covariance matrix of each cluster is calculated. The barycenter is calculated in the usual way, as the energy weighted mean of the cluster, and the covariance is calculated as

$$\sigma_{ij} = \frac{1}{N} \sum_{clusters}^{N} (\vec{x}_i - \bar{\vec{x}}_i)(\vec{x}_j - \bar{\vec{x}}_j) \tag{1}$$

where $\vec{x} = [x, y, E]^T$, the relevant variables for the calorimeter covariance matrix. We leave open the possibility of tuning these variables for possible detector response, but explicitly do not include any response due to the electronics, as in the development of this module, the study of the PhaseII upgrade ECAL was envisioned and the electronics to be used was unknown.

With the relevant covariance matrices and barycenters in place, as well as the contributing clusters and seed, we form the relevant `LHCb::CaloPosition`, `LHCb::CaloCluster` and `LHCb::CaloHypo` which finalize the `LHCb::ProtoParticle` for the neutral particle in the calorimeter.

7 Timing measurements

To estimate the performance increase gain by adopting DELPHES, we run event simulations with the standard GAUSS implementation for 2012, the implementation only using the generation phase, and GAUSS with DELPHES for 200 events. As the performance for timing can vary wildly with the type of decay produced, we choose a collection of decays files, listed in Table 1. This sample is chosen to try to cover many standard use cases. This includes minimum bias events, as well as decays of both beauty and charm. To ensure that the timing is not biased as a function of load on the lxplus node used, provide both the event loop and GaussGen algorithm timings, which allow for normalization to non-changing algorithms. We report also the individual sequence timing for minimum bias events in Table 2.

We see that on the whole, the timing of the event loop itself is $\mathcal{O}(10\%)$ of the full simulation time. This factor is *only* on the Gauss + Boole itself, and does not include

Table 1: Timing in ms of the standard DELPHES sequence in comparison to GAUSS with and without simulation applied. All tests were performed on lxplus with the default options for each configuration. All have been run using PYTHIA 8. The values reported as the mean clock time from the TimingAuditor.

Decay File	DELPHES Sequence	GaussGen	Event Loop	GAUSS generation only, MainEventSeq	GaussGen	Event Loop	GAUSS with Simulation MainEventSeq	Gauss Gen	Event Loop
Minimum Bias (30000000)	5211.268	0.193	5578.220	103.685	0.148	163.958	33205.383	1.838	33743.742
$B^0 \to \pi^+\pi^-$ (11102013)	4687.377	0.276	45952.051	4687.377	0.276	45952.051	37975.605	1.312	71846.844
$D^{*+} \to D^0\pi_s^+$, $D^0 \to K^-\pi^+\pi^-\pi^+$ (27265000)	4839.892	0.248	6551.347	1322.671	0.172	1408.644	58232.062	0.296	60298.934
$B^+ \to K^{*+}\mu^+\mu^-$ (11114001)	4660.982	0.258	20253.385	16744.717	0.181	16829.184	56409.137	0.331	73254.391
$B_s^0 \to \phi\gamma$ (13102201)	4439.395	0.210	25153.633	20118.775	0.174	20195.631	52773.910	0.396	73449.562

Table 2: Individual contributions to the DELPHES sequence for min bias events generated with PYTHIA 8, as reported by the timing monitor. We also list the Gauss sequence and GaussGen for direct comparison with Table 1.

Step	Mean User Time (ms)	Mean Clock Time	min	max	sigma
InitDelphes	0.250	0.285	0.095	1.6	0.18
DelphesAlg	4111.200	4111.615	0.569	16470.6	3491.46
DelphesHist	3.800	3.784	0.026	11.6	2.81
DelphesProto	0.800	1.020	0.041	3.1	0.57
DelphesRecoSummary	0.000	0.021	0.010	0.3	0.02
DelphesParticleId	532.950	533.180	0.023	17297.0	1280.64
ChargedProtoParticleAddRich	0.150	0.093	0.008	1.9	0.13
ChargedProtoParticleAddMuon	0.050	0.038	0.007	0.9	0.06
ChargedProtoCombineDLLsAlg	0.000	0.050	0.005	0.5	0.04
DelphesCaloProto	546.500	546.458	0.040	3248.9	513.52
DelphesTuple	13.650	14.138	0.053	47.7	11.90
BooleInit	0.300	0.306	0.109	2.0	0.22
PGPrimaryVertex	0.100	0.025	0.017	0.3	0.02

the running of BRUNEL or any other possible steps. Even more striking, comparing $D^0 \to K^-\pi^+\pi^-\pi^+$ for full simulation, the speedup is a factor of 12 for the total event loop. This can be attributed almost entirely to the lack of particles generated in interactions.

8 Conclusion

We have presented the incorporation and deployment of a DELPHES based simulation in LHCb. Algorithms for the full simulation of events taking a fraction of the time of traditional simulation have been implemented and have been shown to produce full physics output on the timescale of the traditional simulation phase alone. Future improvements include the final tunings of the primary vertex smearing for all years, which can then be used as input at configuration time, and as well individual tunings for the calorimeter and PID for each year of datataking. Future collaboration with and feedback from physics working groups is paramount to have the final parameterizations for physics productions provided.

The presented framework here is only valid for signal particles generated from a decay file or any other particle produced by Pythia. This does not adequatedly model background distributions. For this use, either a hybrid generation of signal only using Delphes combined statistically with a background sample or a different solution is warranted. We note this to be completely clear in the application of this tool.

Acknowledgements

We thank Marco Cattaneo, Marco Clemencic, Gloria Corti and Domink Müller for their technichal help in all parts of this project. We thank Agnieszka Dziurda and Lucia Grillo for the help with the development of the tracking parameterizations. We also thank Concezio Bozzi for general support and guidance.

References

[1] DELPHES 3, J. de Favereau *et al.*, *DELPHES 3, A modular framework for fast simulation of a generic collider experiment*, JHEP **02** (2014) 057, `arXiv:1307.6346`.

[2] M. Clemencic *et al.*, *The LHCb simulation application, Gauss: Design, evolution and experience*, J. Phys. Conf. Ser. **331** (2011) 032023.

[3] Geant4 collaboration, J. Allison *et al.*, *Geant4 developments and applications*, IEEE Trans. Nucl. Sci. **53** (2006) 270.

[4] LHCb collaboration, A. A. Alves Jr. *et al.*, *The LHCb detector at the LHC*, JINST **3** (2008) S08005.

[5] L. Anderlini, D. Derkach, N. Kazeev, and A. Maevskiy, *Artificial Neural Networks for the Ultra-Fast Simulation of the LHCb experiment and its upgrade*, Tech. Rep. LHCb-INT-2019-014. CERN-LHCb-INT-2019-014, CERN, Geneva, May, 2019.