Simulation d’un calorimètre

Projet informatique de Line Delagrange, Marie Hartmann et Périne Miriot

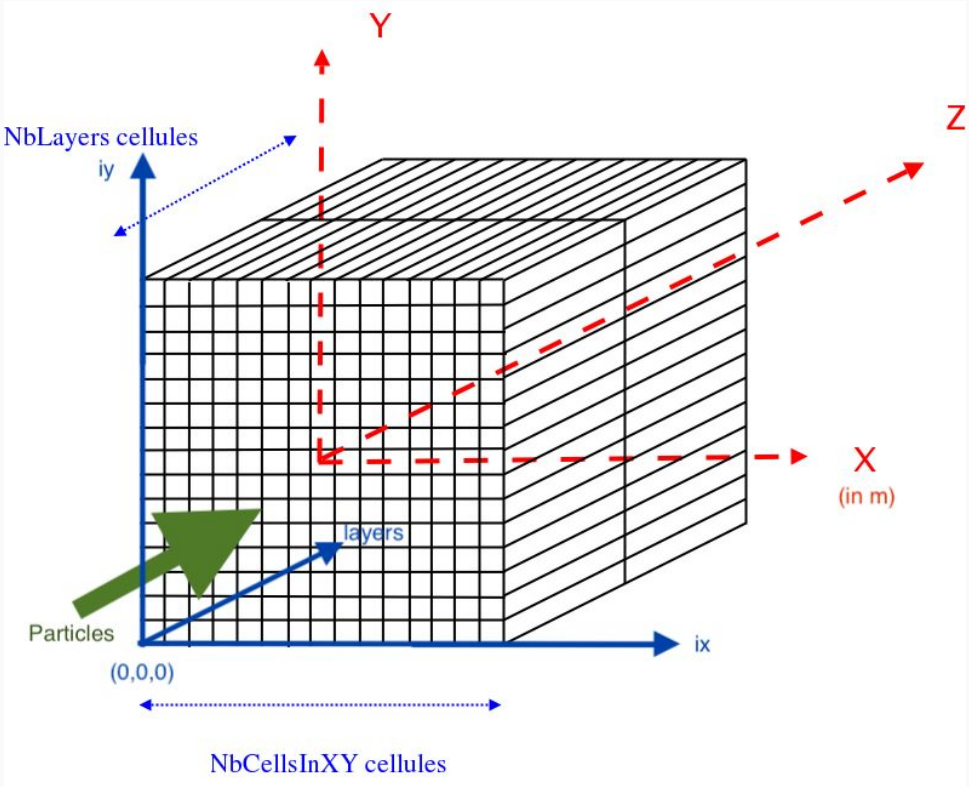
14 mars 2022 - 18 mars 2022

Introduction

- **Objectif** : Simulation simplifiée de particules traversant un calorimètre, reconstruction du point d'impact et de l' énergie déposée, identification de la particule

- **Calorimètre** : parallélépipède divisé en cellules, chacunes lues individuellement

- Langage informatique : C++/ROOT



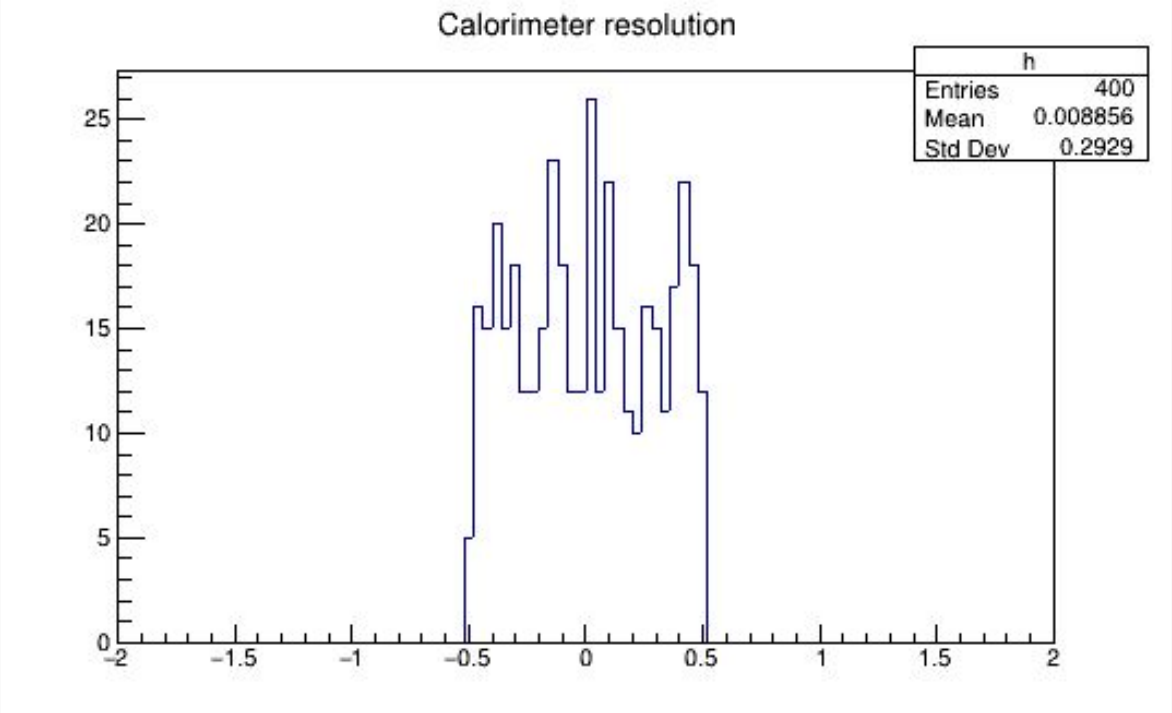
Plan

1) Quelques exercices
2) Simulation du calorimètre
3) Simulation de la gerbe électromagnétique
4) Reconstruction de la position de la particule incidente dans le calorimètre
5) Simulation d’une gerbe hadronique

Quelques exercices

Résolution du calorimètre

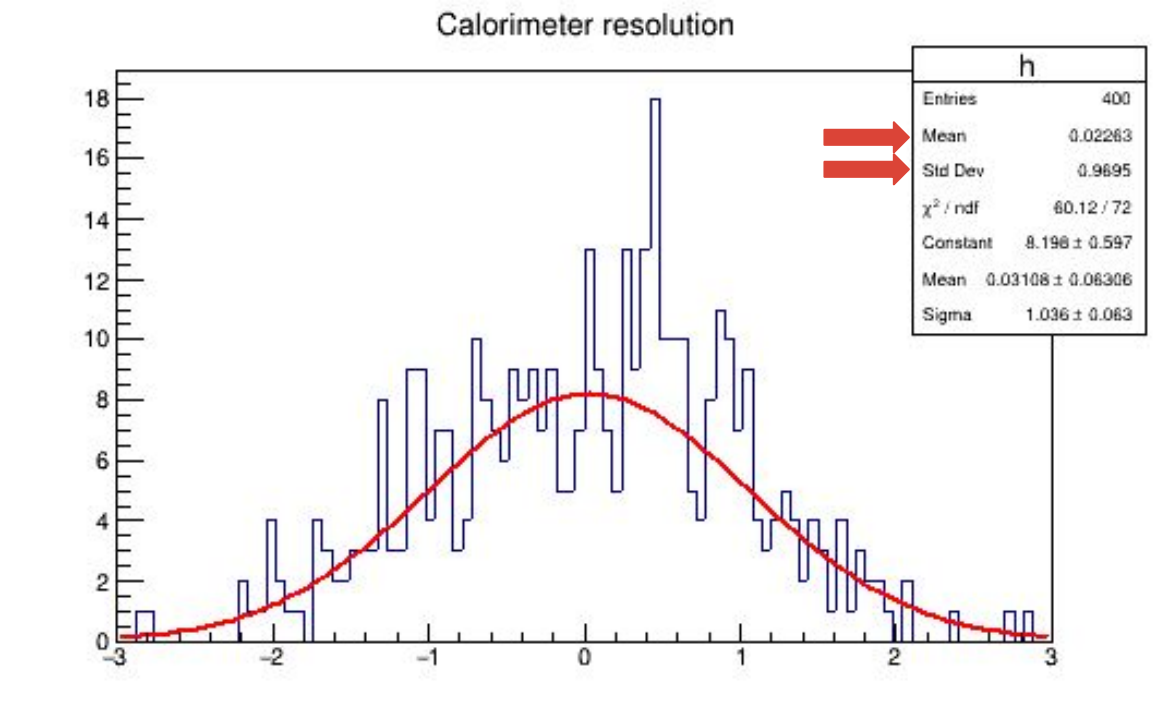
(1) On plot e_reco - e_true (énergie reconstruite dans le calorimètre - énergie de la particule) pour obtenir la résolution du calorimètre. On obtient bien une distribution uniforme entre -0.5 et +0.5 GeV, comme ce qui est parametré dans reconstruct.cxx



Quelques exercices

Résolution du calorimètre

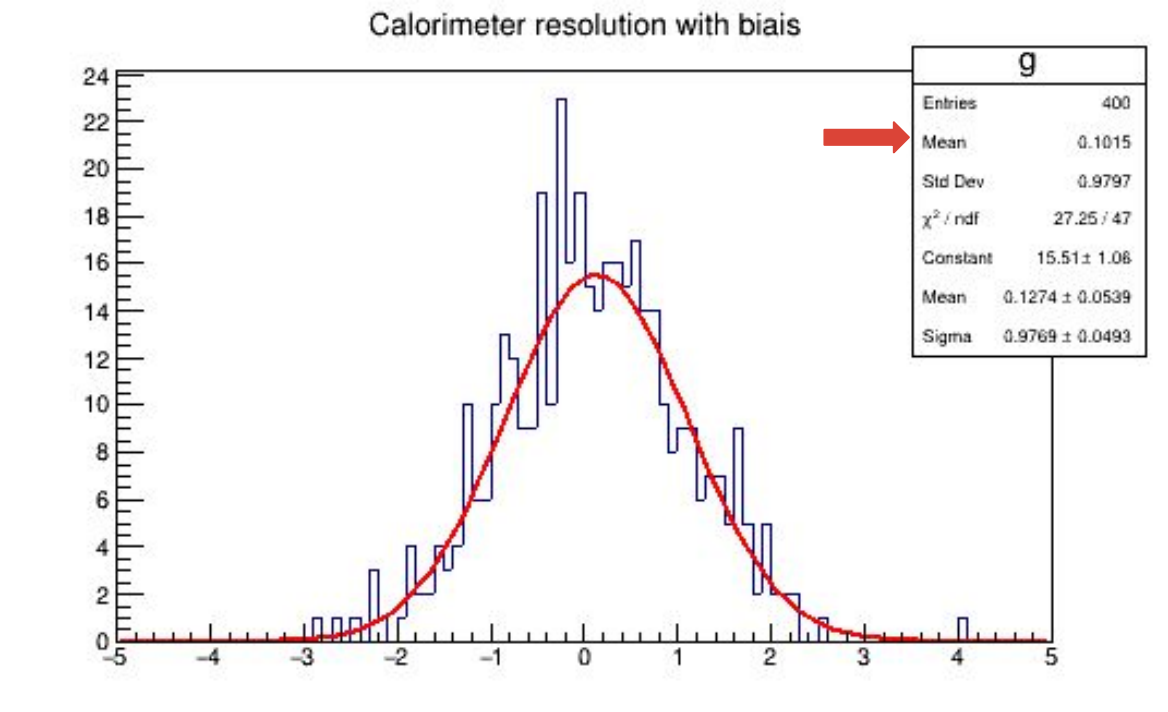
(2) On change la résolution du calorimètre dans reconstruct.cxx en une gaussienne de largeur 1GeV, et on applique un fit pour retrouver une gaussienne centrée en 0 de largeur 1



Quelques exercices

Résolution du calorimètre

(3) On introduit un biais de 0.1GeV dans la résolution et on vérifie qu'on le retrouve lors du fit par une gaussienne (on s'attend à ce que la gaussienne soit centrée en 0.1 GeV)



Quelques exercices

Génération d'une gaussienne par un algorithme Monte Carlo :

On utilise un algorithme de type Monte Carlo pour générer la gaussienne dans un histogram h, et mesurer l'intégrale sous la courbe après 3sigma et avant -3sigma.
On utilise gRandom pour générer une distribution gaussienne pour comparaison.

Un extrait du code est présenté ci-contre :

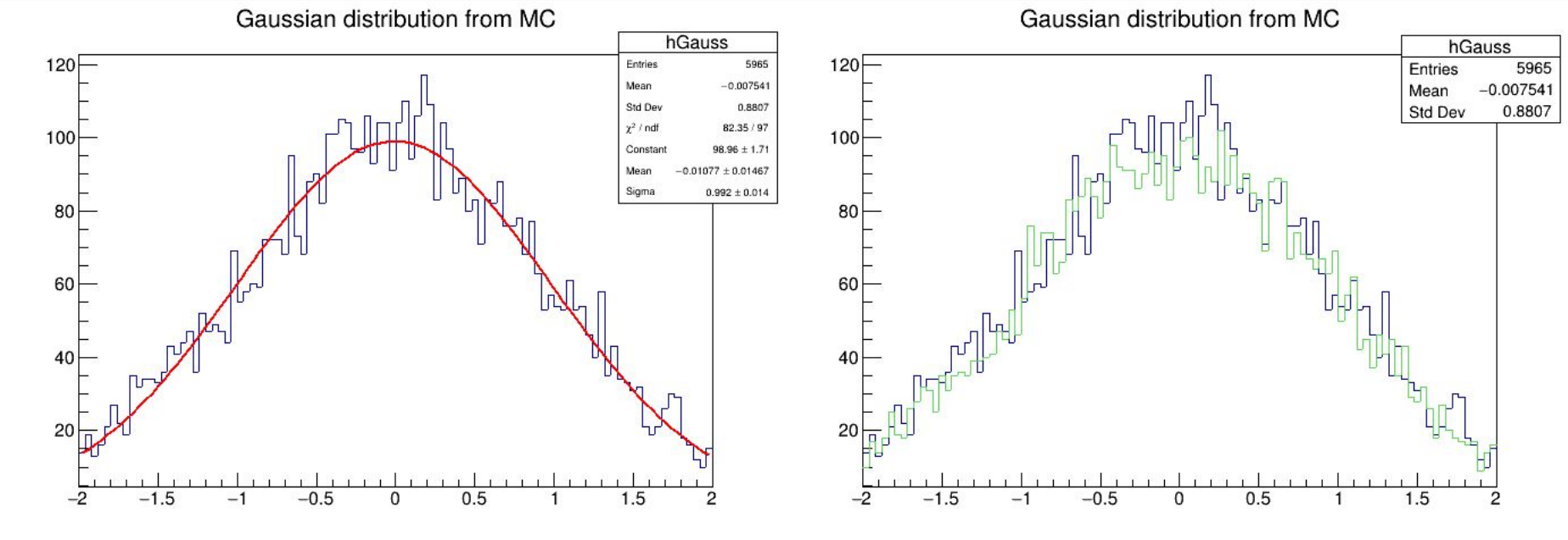
```
// Draw 10000 numbers according the gaussian law
// We use gRandom->Uniform(MAX)  to draw numbers
// uniformly  between  0. and MAX
for (int i=0; i<10000; i++) {
    y = gRandom->Uniform(1);
    x = gRandom->Uniform(4) -2;
    if (y < ROOT::Math::gaussian_pdf(x,sigma,x_0)/F) {
        // we  fill here the histogram h
        h->Fill(x);

        // and we fill the histogram g
        // inside the if to get the same number of events
        x_1 = gRandom->Gaus(x_0,sigma);
        g->Fill(x_1);

        // Computation of the Area
        // for 3 sigma
        counter = counter +1; // for the normalisation
        if ( abs(x) > 3*sigma) {
            area = area + 1.;
        }
    }
}
```

Quelques exercices

Génération d'une gaussienne par un algorithme Monte Carlo :



Distribution de la gaussienne obtenue par Monte Carlo (bleu), et par gRandom (vert), et fit de la gaussienne (rouge)

Simulation du calorimètre

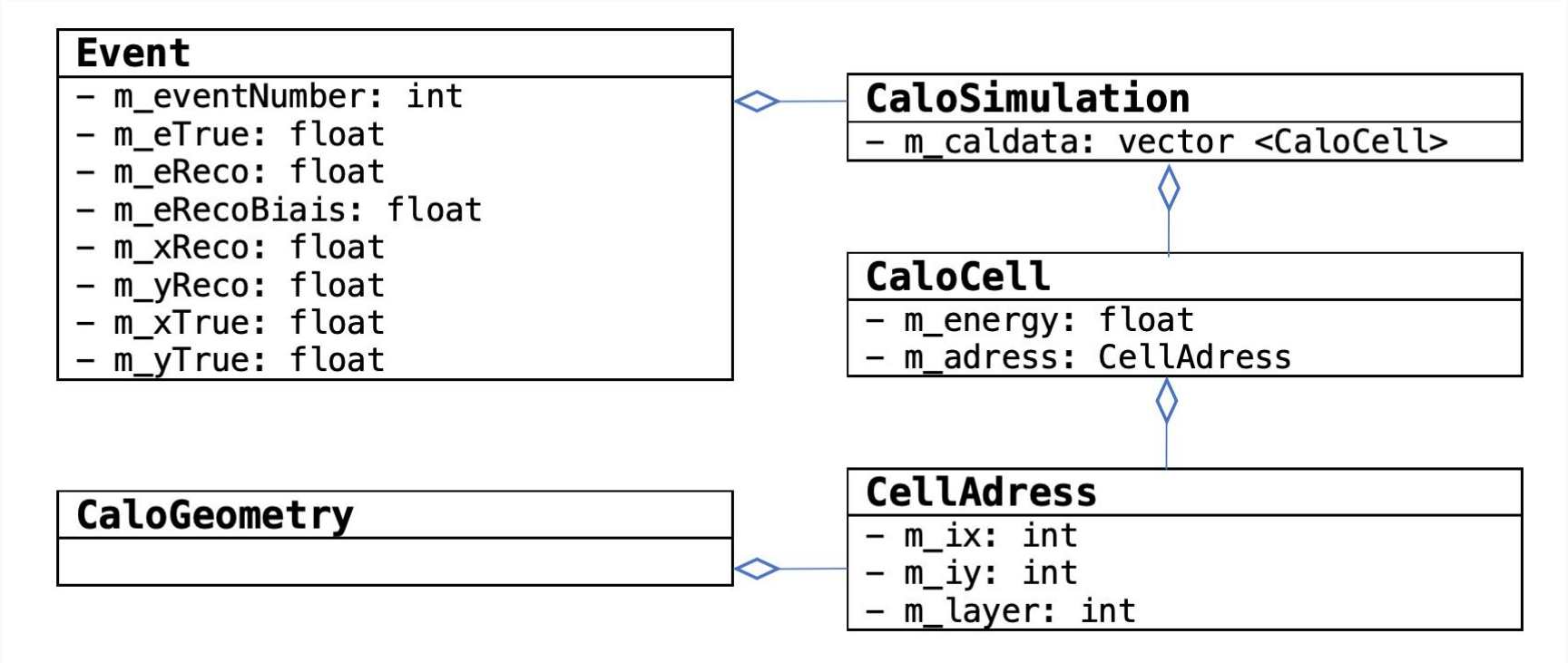


Diagramme de classes ULM avec les noms des classes et leurs attributs (une version de ce diagramme avec les méthodes en plus est disponible sur GitHub)

Simulation du calorimètre

CellAddress

Un objet CellAddress contient les coordonnées en 3D (en terme d'indices) d'une cellule. La classe contient une fonction CellAddress::isValid() pour vérifier que l'adresse est contenue dans les dimensions du calorimètre.

```
// Is the address valid.
bool CellAddress::IsValid() const  {
    return  (m_ix <= NbCellsInXY && 0 <= m_ix &&
       m_iy <= NbCellsInXY && 0 <= m_iy &&
       m_layer <= NbLayers && 0 <= m_layer );
}
```

CaloCell

Un objet CaloCell contient les informations relatives à une cellule : son adresse (sous la forme CellAddress) et son énergie.

Simulation du calorimètre

CaloGeometry::isInside

Fonction qui donne l'adresse de la cellule (sous la forme d'une CellAddress) qui contient un point dont les coordonnées sont données en argument (en m).

CaloGeometry::xCentre et yCentre

Donnent la position en mètre du centre de la cellule en argument

```
// Return false if the (x,y,z) point is not located in the calorimeter
// volume otherwise return true and fill the CellAddress variables with the
// address of the cell that contains this point.
bool CaloGeometry::IsInside(double xyz[3], CellAddress& cellAddress){
    double x = xyz[0], y = xyz[1], z = xyz[2];

    if ( x <= XYMax && XYMin <= x &&
         y <= XYMax && XYMin <= y &&
         z <= ZMax && ZMin <= z )
       {
          // We take the origin of the indices of the cell on the bottom left
          cellAddress = CellAddress(int( (x-XYMin) / XYSize  ),
                                    int( (y-XYMin) / XYSize ),
                                    int( z / ZSize ));
          return true;
       }
    return false;
}

// Give the position of the cell center.
double CaloGeometry::xCentre(const CellAddress& cellAddress){
    int ix = cellAddress.ix();
    return (ix+0.5)*XYSize+XYMin;
}
double CaloGeometry::yCentre(const CellAddress& cellAddress){
    int iy = cellAddress.iy();
    return (iy+0.5)*XYSize+XYMin;
}
```

Simulation du calorimètre

CaloSimulation::CalorimeterData

Crée le calorimètre comme un vecteur de cellules (chacune sous la forme CaloCell). On choisit d'utiliser un vecteur plutôt qu'une map, et on agence les boucles de sorte que les relations d'ordres sur les coordonnées du calo donnent calodata[i] < calodata[i+1].

CaloSimulation::ClearCalorimeter

Vide le calorimètre

```
void CaloSimulation::CalorimeterData() {
    for (int iz=0; iz < NbLayers; iz++){
        for (int iy=0; iy <NbCellsInXY; iy++){
            for (int ix=0; ix <NbCellsInXY; ix++) {
                CellAddress CellAd = CellAddress(ix,iy,iz);
                CaloCell CalCell = CaloCell(CellAd, 0.);
                m_caldata.push_back(CalCell);
            }
        }
    }
}

void CaloSimulation::ClearCalorimeter(){
    for (int iz=0; iz < NbLayers; iz++){
        for (int iy=0; iy < NbCellsInXY; iy++){
            for (int ix=0; ix < NbCellsInXY; ix++) {
                int index = caldataIndex(ix,iy,iz);
                m_caldata[index].setEnergy(0.);
            }
        }
    }
}
```

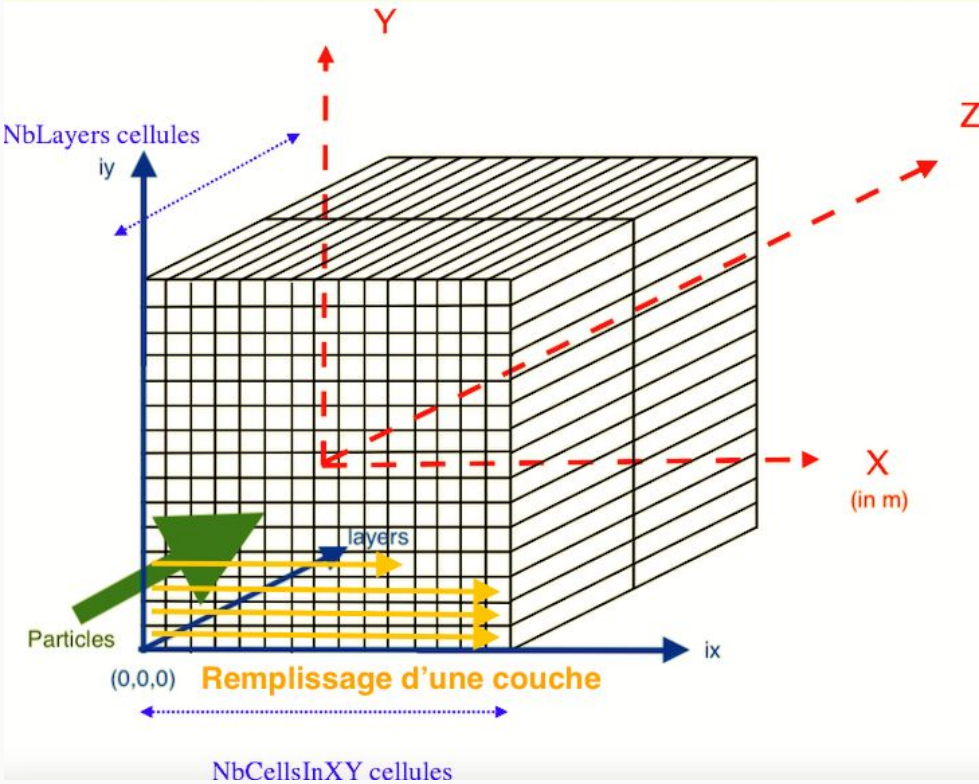
Simulation du calorimètre

CaloSimulation::caldataindex

Cette fonction permet de retrouver l'indice du vecteur correspondant aux coordonnées de la cellule :

$$i = iz \times NbCellsinXY^2 + iy \times NbCellsinXY + ix$$

(on remplit d'abord en x puis en y puis en z)



Simulation de la gerbe électromagnétique

CaloSimulation::SimulateShower(float x, float y, float energy)

Nous avons tout d'abord créé une fonction dans la classe CaloSimulation qui va, pour un point d'impact donné (x,y) et pour une énergie de la particule incidente donnée, générer la gerbe dans le calorimètre.

Les paramètres utilisés pour la simulation de cette gerbe sont données dans l'énoncé (a, b, longueur de radiation, rayon de Molière et sigma de la gaussienne).

Pour simuler la gerbe, nous allons, pour chaque couche, calculer l'énergie déposée par la particule dans cette couche. L'énergie déposée dans chaque cellule de cette couche sera alors extraite par une gaussienne selon les deux dimensions transverses (ici x et y).

Simulation de la gerbe électromagnétique

CaloSimulation::SimulateShower(float x, float y, float energy)

Pour chaque couche, on note *ParticleEnergy* l'énergie de la particule qui arrive. Elle déposera alors une énergie *EnergyDeposited* qui vaut l'intégrale sur la couche de dE/dt.
Nous avons gardé la variable t pour l'intégration, où t=z/X0, et nous avons défini les bornes d'intégration pour cette variable t :

```
TF1 *F = new TF1("F", " [1] * pow(([1]*x),([0]-1)) * exp(-[1]*x) / tgamma([0]) ");
EnergyDeposited = F->Integral(iz*ZSize/X0, (iz+1)*ZSize/X0, 1e-12)*energy;
```

Remarque : Nous avons fait ce calcul à l'intérieur des boucles sur toutes les cellules sur le plan transverse, cela n'est pas nécessaire. On aurait pu le faire avant ces deux boucles for et ainsi éviter de calculer plusieurs fois la même valeur.

Simulation de la gerbe électromagnétique

CaloSimulation::SimulateShower(float x, float y, float energy)

Ensuite, pour chaque cellule de la couche, nous avons calculé l'énergie déposée dans cette cellule en multipliant l'énergie déposée sur l'ensemble de la couche par deux gaussiennes (pour les deux dimensions x et y), intégrées sur la cellule en question.

```
double DeltaGX = GX->Integral(ix*XYSize+XYMin, (ix+1)*XYSize+XYMin, 1e-12);
double DeltaGY = GY->Integral(iy*XYSize+XYMin, (iy+1)*XYSize+XYMin, 1e-12);
float NewEnergy = EnergyDeposited * DeltaGX * DeltaGY;
```

On rempli finalement la cellule du calorimètre avec cette énergie.

Remarque : on ajoute une condition pour l'arrêt du développement de la gerbe, dans le cas où la particule n'a plus suffisamment d'énergie.

Simulation de la gerbe électromagnétique

MainEvent.cxx

L'étape suivante est de simuler la gerbe dans notre calorimètre, en générant un point d'impact aléatoirement sur la première couche du calorimètre, et pour une particule d'énergie 50GeV.

```
// draw randomly the impact point in the calorimeter
//xTrue = gRandom->Uniform(XYMax - XYMin) + XYMin;
//yTrue = gRandom->Uniform(XYMax - XYMin) + XYMin;

// draw randomly the impact point in a particular cell
xTrue = gRandom -> Uniform(0.1) - 0.1;
yTrue = gRandom -> Uniform(0.1) - 0.1;

// simulation
simulate(event, Calorimeter, xTrue, yTrue);
ana_simu(event, Calorimeter);
```

Nous appelons pour ça la fonction **simulate**.

Simulation de la gerbe électromagnétique

Simulate.cxx

La fonction **simulate** va utiliser **SimulateShower** sur notre calorimètre *Calorimeter* défini dans **MainEvent.cxx** et le remplir avec les énergies déposées dans chaque cellule.

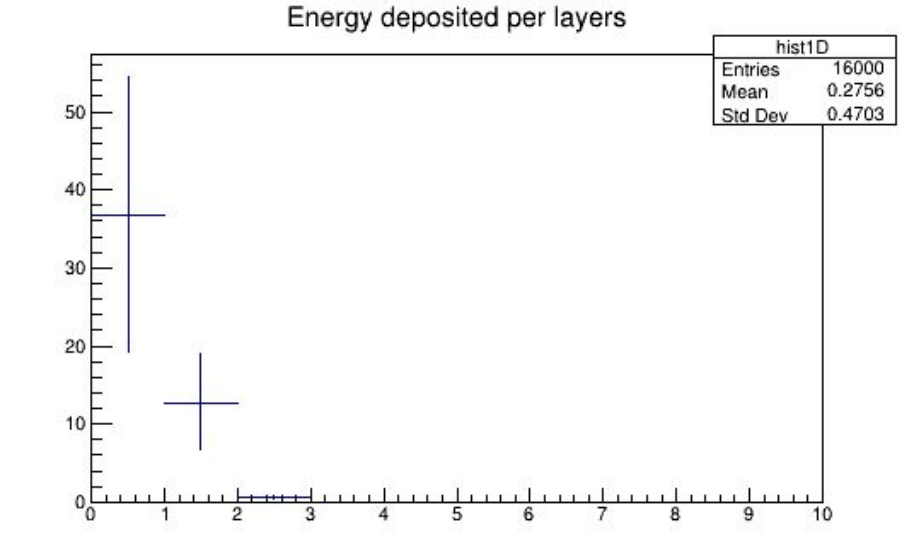
```
void simulate(Event& event, CaloSimulation& Calorimeter, float XImpact, float YImpact)
{
    // simulate the event
    event.seteTrue(50.); // fixed true energy
    float EnergyTrue = event.eTrue();
    float E0 = 1;
    float sigma = 0.1*sqrt(EnergyTrue/E0);
    float eTrueFluc = gRandom->Gaus(event.eTrue(), sigma);
    event.seteTrue(eTrueFluc);

    //simulate the energy deposit (pure electromagnetic)
    Calorimeter.SimulateShower(XImpact, YImpact, event.eTrue());
```

Simulation de la gerbe électromagnétique

Visualisation de la gerbe et test de la simulation

Pour tester nos scripts, nous avons tout d'abord tracé l'énergie déposée dans chaque couche. Nous obtenons le graphique suivant :



Caractéristiques :

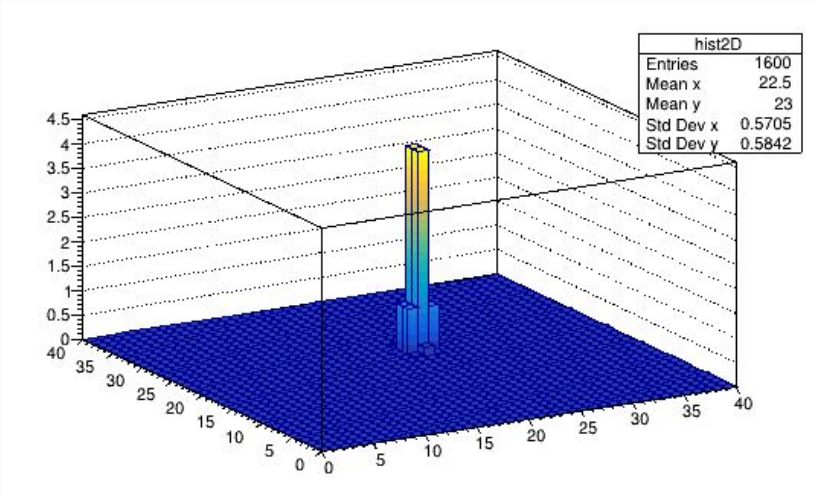
- 10 couches
- énergie incidente = 50 GeV
- 1 événement

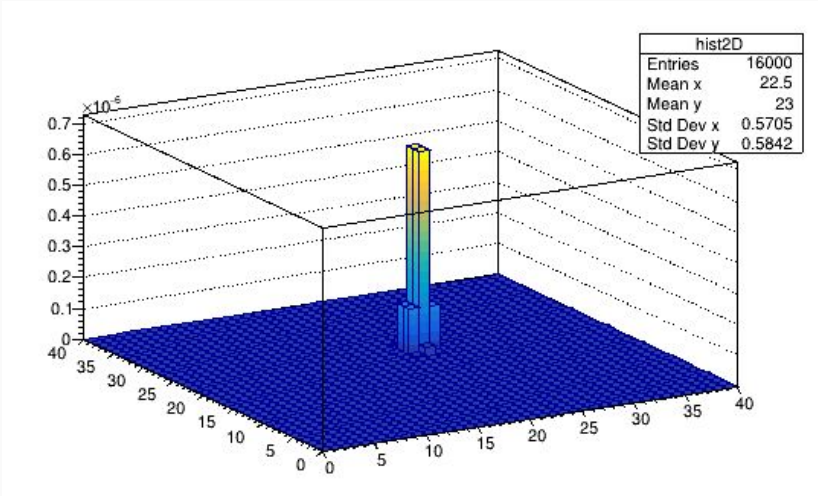
Nous voyons bien que l'énergie déposée diminue à chaque nouvelle couche, et que lorsque la particule atteint la 4ème couche elle n'a plus d' énergie
(notre condition fonctionne donc bien).

Simulation de la gerbe électromagnétique

Visualisation de la gerbe et test de la simulation

Nous avons ensuite tracé le dépôt d'énergie sur le plan transverse, pour différentes couches, toujours avec une énergie incidente de 50 GeV et pour 10 couches. Voici les résultats pour, respectivement, la couche n°1 (à gauche) et n°5 (à droite) :





Simulation de la gerbe électromagnétique

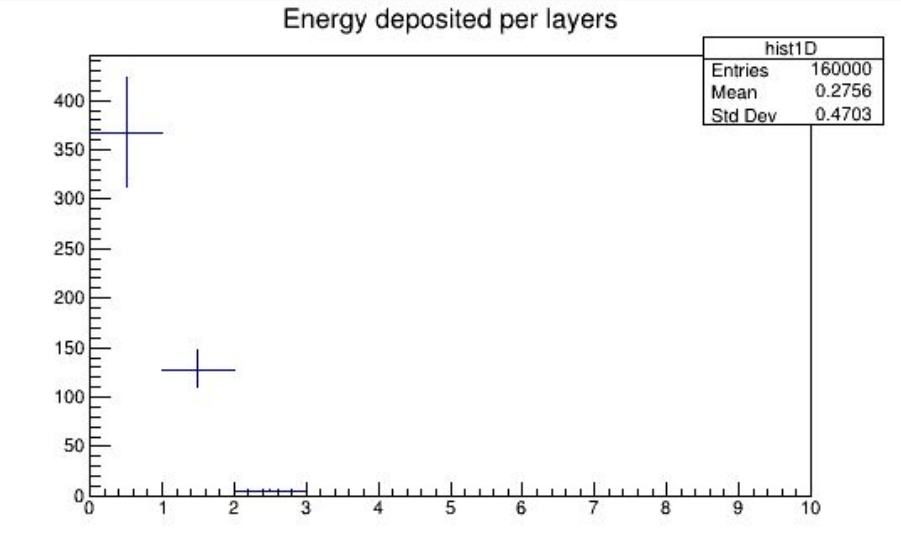
Visualisation de la gerbe et test de la simulation

Nous remarquons bien que la répartition est centrée autour du point d'impact et qu'une fois la couche 5 atteinte, nous n'avons pratiquement plus d'énergie (ordre de grandeur 1e-6). Cela est cohérent avec le tracé des énergie couche par couche slide 20.

Simulation de la gerbe électromagnétique

Visualisation de la gerbe et test de la simulation

Nous avons finalement tracé ces mêmes graphiques pour cette fois-ci 10 événements générés.



Caractéristiques :

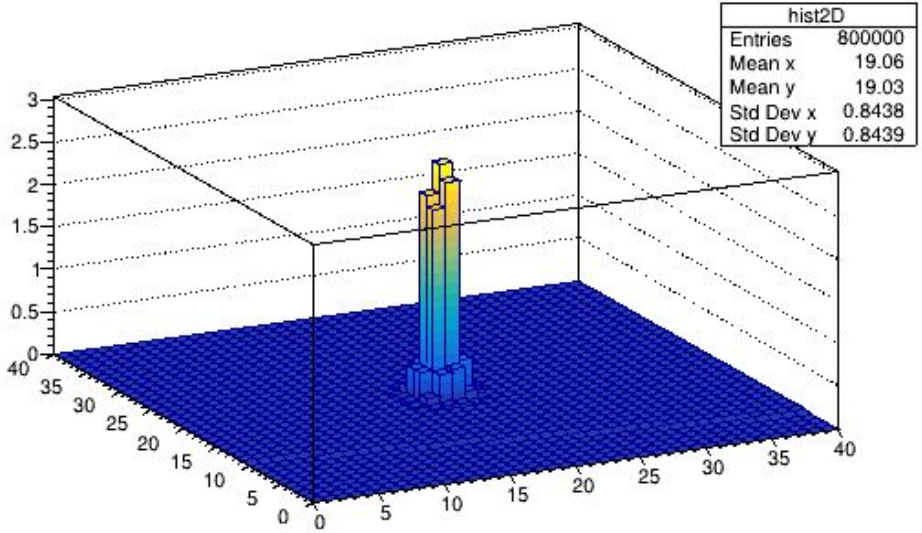
- 10 couches
- énergie incidente = 50 GeV
- 10 événements

Nous voyons bien que l'énergie dans la première couche est d'environ 370, ce qui est en accord avec nos 10 événements de 50 GeV (10*50 GeV = 500 GeV)

Simulation de la gerbe électromagnétique

Visualisation de la gerbe et test de la simulation

Enfin, un dernier graphique que nous avons regardé est l'histogramme 2D pour 500 événements associé à la couche n°1 :



Caractéristiques :

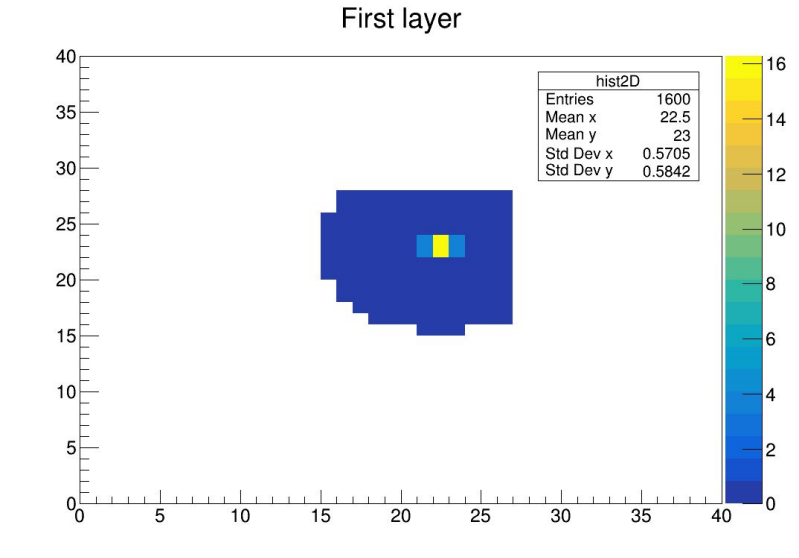
- 10 couches
- énergie incidente = 50 GeV
- 500 événements

Nous voyons que le nombre d'entrées a bien augmenté en conséquence.

Simulation de la gerbe électromagnétique

Visualisation de la gerbe et test de la simulation

Une dernière remarque concernant le tracé de l'histogramme 2D, nous pouvons également le représenter comme suit :



Caractéristiques :

- 10 couches
- énergie incidente = 50 GeV
- 1 événement

Nous voyons ici un problème de symétrie, qui disparaît en passant en échelle log.

Cette représentation nous sera utile pour des comparaisons ultérieures avec la gerbe hadronique.

Simulation de la gerbe électromagnétique

Réponse du calorimètre : fluctuations de l'énergie incidente

Pour prendre en compte les fluctuations possibles du nombre de particules émises pendant la gerbe dans le calorimètre, nous n'utilisons pas directement une énergie de 50 GeV pour tous les événements, mais nous allons la faire fluctuer.

```
void simulate(Event& event, CaloSimulation& Calorimeter, float XImpact, float YImpact)
{
    // simulate the event
    event.seteTrue(50.); // fixed true energy
    float EnergyTrue = event.eTrue();
    float E0 = 1;
    float sigma = 0.1*sqrt(EnergyTrue/E0);
    float eTrueFluc = gRandom->Gaus(event.eTrue(), sigma);
    event.seteTrue(eTrueFluc);

    //simulate the energy deposit (pure electromagnetic)
    Calorimeter.SimulateShower(XImpact, YImpact, event.eTrue());
```

L'énergie utilisée dans **simulate** est alors *eTrueFluc*, calculée à partir de eTrue = 50GeV en suivant une gaussienne de sigma donné dans l'énoncé.

Reconstruction

Objectif

Lors d'une vraie prise de donnée, le calorimètre va mesurer l'énergie déposée dans chacune de ses cellules, et ces énergies seront les seules données accessibles. Il sera alors intéressant de savoir reconstruire, à partir de cette répartition en énergie, les coordonnées de la particule incidente sur le calorimètre.

Reconstruction

Objectif

Pour ce faire, nous avons décidé de calculer la position du barycentre en énergie en x et en y.
Dans chaque couche, nous avons sommé la contribution de chaque position en énergie :

```
CenterX = CenterX + ((CaloElements[i].address().ix()+0.5)*XYSize+XYMin) * EnergyCell;
CenterY = CenterY + ((CaloElements[i].address().iy()+0.5)*XYSize+XYMin) * EnergyCell;
```

Remarque : dans le calcul de CenterX et de CenterY, nous avons récupéré à partir de la coordonnée de la cellule (un entier) la position physique de son centre, qui est le point que l'on considère.

Reconstruction

Objectif

Puis nous avons divisé par l'énergie totale (calculée en sommant l'énergie de toutes les cellules du calorimètre), afin d'obtenir la position reconstruite du point d'impact :

```
xReco = CenterX / EnergyTot;
yReco = CenterY / EnergyTot;
```

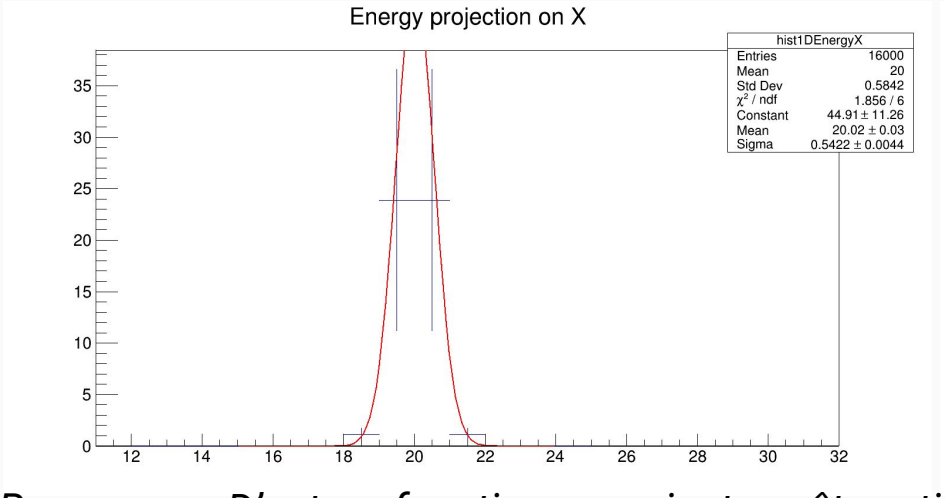
Ainsi, nous avons reconstruit la position du point d'impact de la particule.

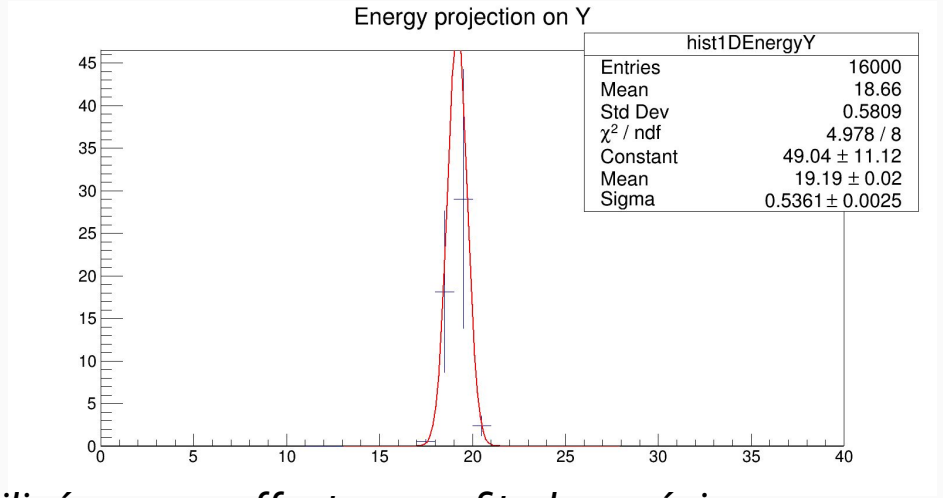
Nous avons également reconstruit l'énergie, en sommant les énergies de chaque cellule du calorimètre.

Reconstruction

Résolutions

Il est alors possible d'obtenir la résolution en énergie et en position en traçant l'énergie en fonction de x et de y. Nous obtenons, avec un fit gaussien :



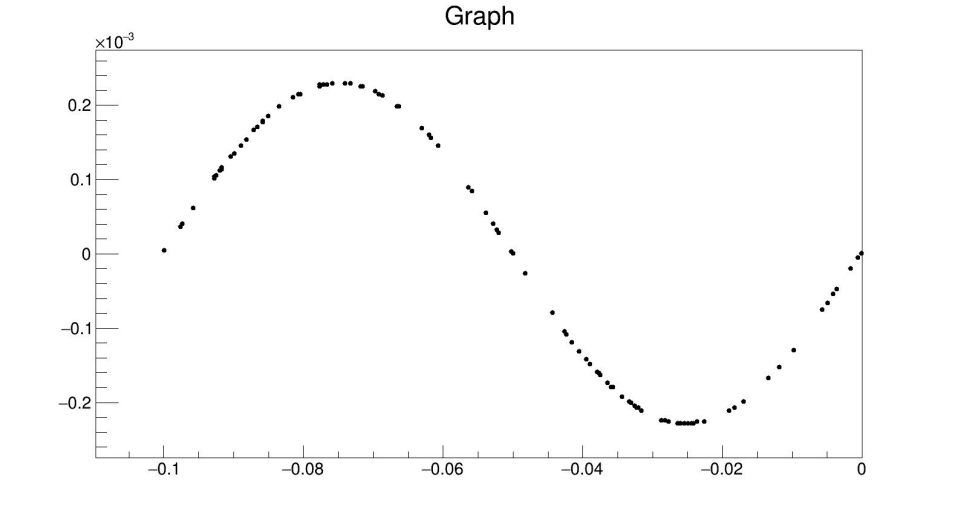


<u>Remarque</u> : D'autres fonctions auraient pu être utilisées pour effectuer un fit plus précis, comme par exemple un combinaison de gaussienne avec une exponentielle ou de gaussienne avec un polynôme.

Reconstruction

S Shape

Il convient ensuite de tracer la distribution S Shape, c'est-à-dire l'énergie reconstruite de la particule incidente - son énergie réelle en fonction de l'énergie reconstruite, selon la position x et la position y. Nous obtenons, selon x, la courbe suivante :



Caractéristiques :

- 10 couches
- énergie incidente = 50 GeV
- 100 événements

Reconstruction

S Shape

A partir de cette courbe, il sera possible de prendre en compte le biais dans la reconstruction des coordonnées du point d'impact.
Selon la valeur de la coordonnée x0, on lui soustraiera la valeur de la distribution S Shape selon x en x0. De même pour y. Cela permettra de supprimer le biais.

Pour déterminer les valeurs de la distribution en x0, nous aurions au préalable fité la fonction à l'aide d'une fonction sigmoïde :

$$f_{A,\lambda}(x) = \frac{A}{1 + e^{-\lambda x}}$$

<u>Remarque</u> : Nous n'avons pas eu le temps de faire un fit de la distribution S Shape en x, ni de tracer celle en y, mais nous voyons qu'elle a la forme attendue, et qu'elle est bien centrée en 0 selon l'axe des ordonnées.

Simulation d’une gerbe hadronique

Objectif

L’objectif ici est de simuler une gerbe hadronique. Pour ce faire, nous avons créé une fonction **SimulateHadShower** dans la classe **CaloSimulation.** La simulation de la gerbe est identique à une gerbe électromagnétique, en changeant toutefois les constantes la décrivant (a, b, longueur de radiation, rayon de Molière et sigma de la gaussienne).

Cependant, au moment de simuler cette gerbe dans le calorimètre, il faut prendre en compte que la gerbe est en fait un mélange entre une gerbe électromagnétique et une gerbe hadronique, quantifié par une variable f issue d’une distribution uniforme entre 0 et 1.
Dans notre cas :

- f=1 correspondra à une gerbe purement électromagnétique
- f=0 à une gerbe purement hadronique

Simulation d'une gerbe hadronique

CaloSimulation::SimulateHadShower(float x, float y, float energy, float f)

Le mélange entre ces deux types de gerbes se fait comme suit.
Dans chaque couche, on calcule l'énergie déposée selon les fractions considérées de composants électromagnétiques et hadroniques :

```
EnergyDepositedEm = FEm->Integral(iz*ZSize/X0, (iz+1)*ZSize/X0, 1e-12) * energy;
EnergyDepositedHad = FHad->Integral(iz*ZSize/L, (iz+1)*ZSize/L, 1e-12) * energy;
EnergyDeposited = f * EnergyDepositedEm + (1-f) * EnergyDepositedHad;
```

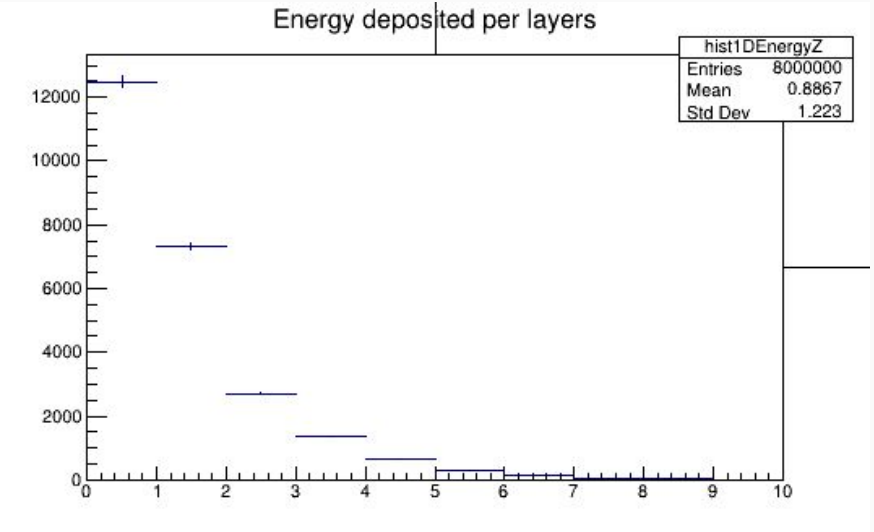
Puis on calcule l'énergie de chaque cellule à partir de gaussiennes comme précédemment, mais de paramètres différents (sigma et sigmaHad) :

```
float NewEnergy = EnergyDeposited*( f * DeltaGXEm * DeltaGYEm + (1-f) * DeltaGXHad * DeltaGYHad);
```

Simulation d’une gerbe hadronique

Visualisation de la gerbe et test de la simulation

Pour tester nos scripts, nous avons simplement remplacé dans simulate.cxx la fonction **SimulateShower** par **SimulateHadShower**. Nous avons d’abord tracé l’énergie déposée dans chaque couche :



Caractéristiques :

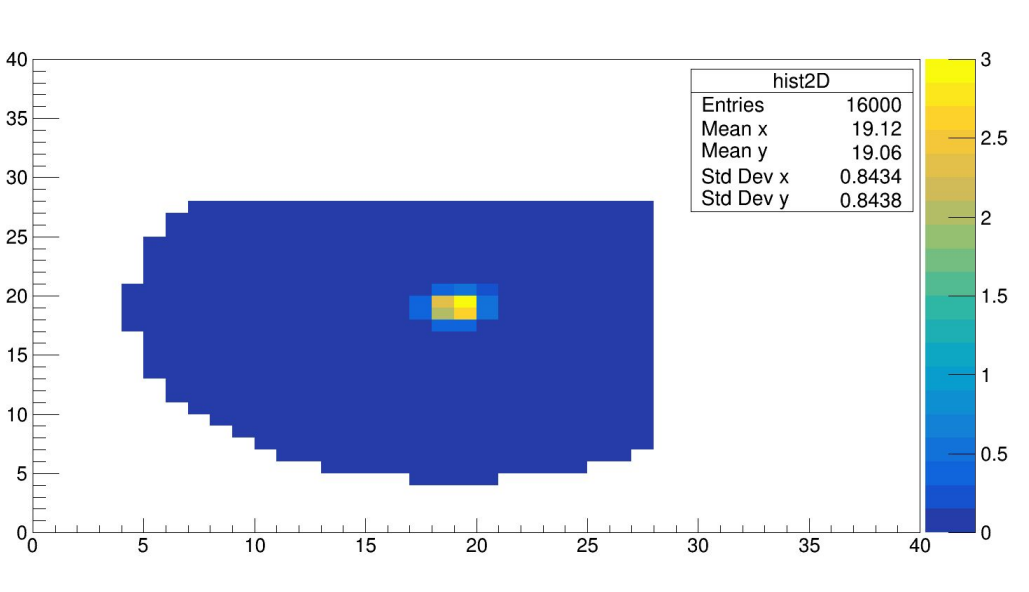
- 10 couches
- énergie incidente = 50 GeV
- 500 événements
- f = 0.5 : mélange

La perte d’énergie est plus douce et plus lente, comme attendu (on atteint la 7ème couche).

Simulation d’une gerbe hadronique

Visualisation de la gerbe et test de la simulation

Pour l’histogramme 2D, toujours pour la première couche, on obtient :



Caractéristiques :

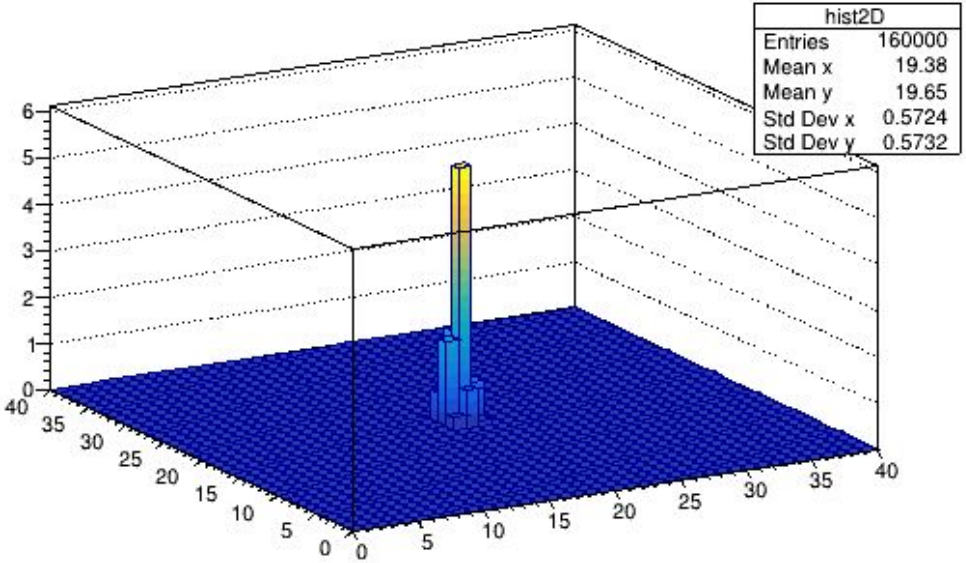
- 10 couches
- énergie incidente = 50 GeV
- 1 événement
- f = 1 : purement hadronique

Nous voyons qu’il est beaucoup plus étalé que pour une gerbe purement électromagnétique (slide 25), ce à quoi on s’attend.

Simulation d’une gerbe hadronique

Visualisation de la gerbe et test de la simulation

Derniers graphiques :



Tracé 1ère couche

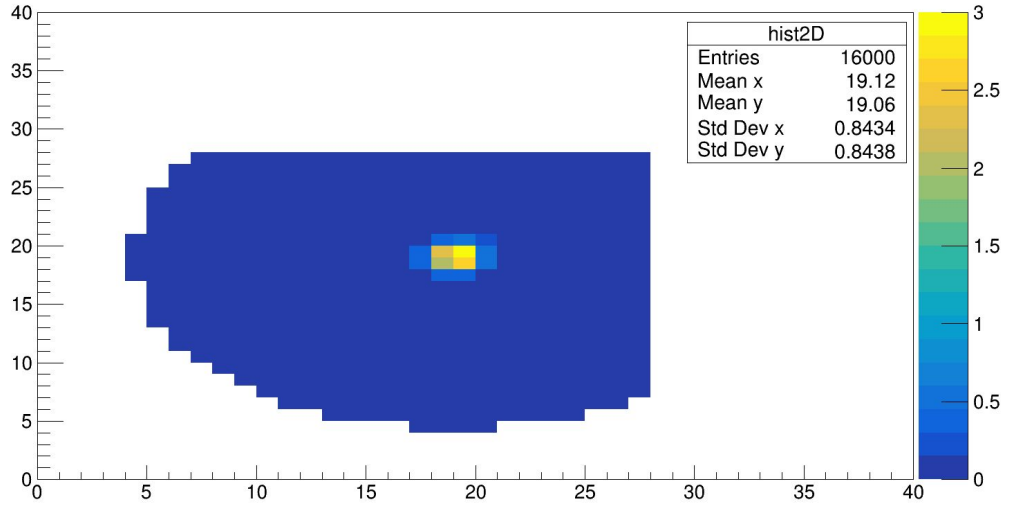
Caractéristiques :

- 10 couches
- énergie incidente = 50 GeV
- 1 événement
- f = 0 -> purement électromagnétique

Simulation d’une gerbe hadronique

Visualisation de la gerbe et test de la simulation

Derniers graphiques :



Tracé 1ère couche

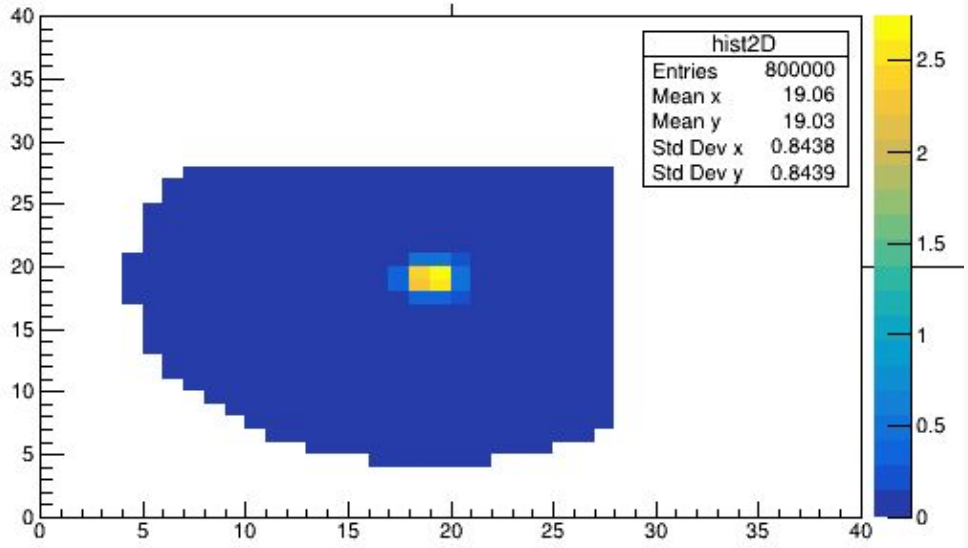
Caractéristiques :

- 10 couches
- énergie incidente = 50 GeV
- 1 événement
- f = 0.5 -> mélange

Simulation d’une gerbe hadronique

Visualisation de la gerbe et test de la simulation

Derniers graphiques :



Tracé 1ère couche

Caractéristiques :

- 10 couches
- énergie incidente = 50 GeV
- 500 événement
- f = 0.5 -> mélange

Bilan

Lors de ce projet, nous avons modélisé un calorimètre simplifié et sa réponse à une gerbe électromagnétique et hadronique. Nous avons reconstruit le point d'impact de la particule à l'entrée du calorimètre, son énergie puis nous avons évalué les biais liés à la segmentation du calorimètre.

Par la suite, on aurait pu corriger ce biais, puis essayer de différencier dans le calorimètre les gerbes hadroniques des gerbes purement électromagnétiques, en sélectionnant et combinant des variables discriminantes pour ces deux types de gerbes.