Tecniche di analisi numerica e simulazione

Progetto Esame

Lorenzo Allasia

A.A. 2024/2025
Consegna: Gennaio 2026

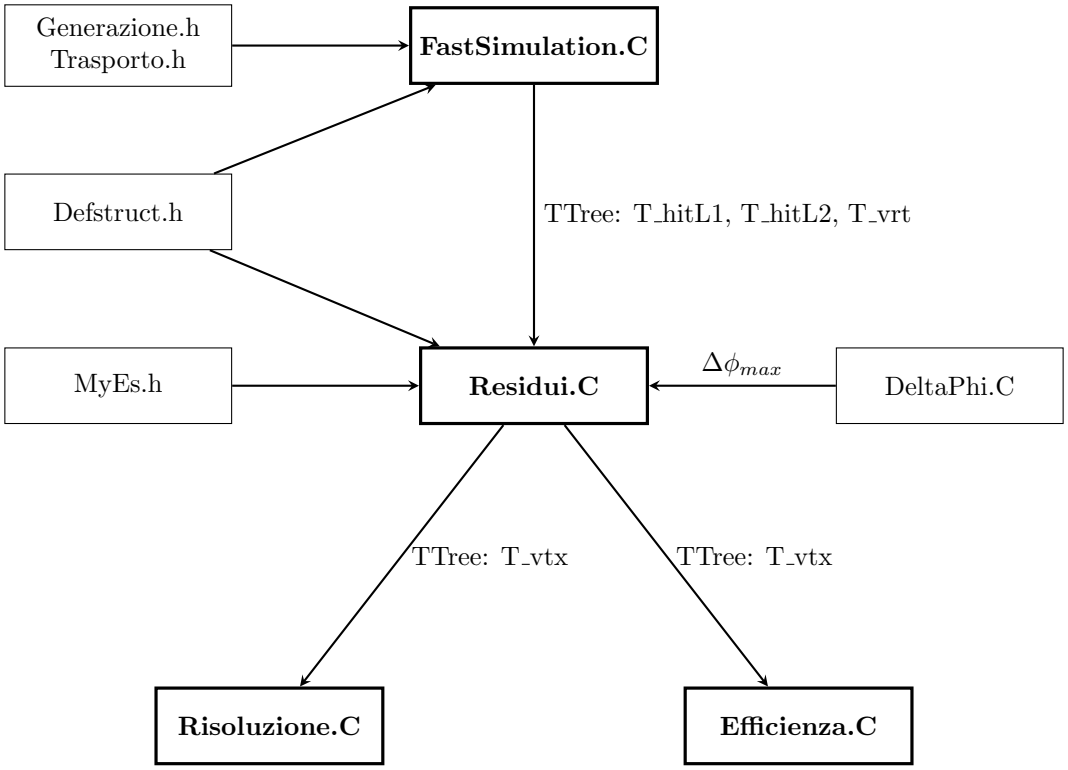
Università degli Studi di Torino

Indice

1 Introduzione

In questa "guida" del progetto finale ho introdotto sostanzialmente il flusso del codice e l'interconnessione tra i diversi moduli.
Essendo che non si doveva fare una dettagliata relazione da n^n pagine sul codice, mi sono incentrato a descrivere le funzioni che sono in mia opinione il cuore dell'algoritmo, spendendo qualche parola sulle funzioni più "complesse" e lasciando implicite quelle più banali.

Riporto il diagramma del flusso di questo programma:



2 FastSimulation.C

La prima macro è FastSimulation.C : ha il compito di simulare la generazione delle particelle, trasportarle lungo al rivelatore e salvare le hit nei due layer con un apposita incertezza data dalla ricostruzione. Ho strutturato l'algoritmo in modo che nel primo ciclo for venga generato il vertice primario con una certa molteplicità di particelle figlie.

```
for (int tot = 1; tot <= numero; ++tot) {
    double x0 = ptr->VertexSimXY();
    double y0 = ptr->VertexSimXY();
    double z0 = ptr->VertexSimZ(distr_z);
    int molti = ptr->Multiplicity(distr_m, m);
```

Il numero di vertici totali, che nel nostro caso corrisponde anche al numero di volte che due pacchetti si incontrano, viene deciso arbitrariamente attraverso la variabile "tot".

Dopodiché per ogni particella figlia, attraverso al secondo for, viene generata la direzione casuale: per l'angolo azimutale si utilizza una distribuzione uniforme, per quello zenitale viene estratta randomicamente una pseudorapidità da una distribuzione data dal professore.
Non si applicano criteri di conservazione dell'impulso totale tra le particelle figlie [1] sia perché non abbiamo informazioni sulla massa e velocità (anche se volendo si potrebbero considerare tutte uguali) e anche perché non porterebbe modifiche all'efficienza di ricostruzione di questo algoritmo. Sicuramente se ci fosse stato del pile-up poteva essere una condizione utile per la ricostruzione del vertice.

Viene riportato il secondo for:

```
for (int i = 0; i < molti; ++i) {
    double phi = ptr->Phi();
    double theta = ptr->Theta();

    versori[0] = TMath::Sin(theta) * TMath::Cos(phi);
    versori[1] = TMath::Sin(theta) * TMath::Sin(phi);
    versori[2] = TMath::Cos(theta);

    punto[0] = x0;
    punto[1] = y0;
    punto[2] = z0;

    ptr2->EquazioneRetta(punto, versori, ptr2->GetRPipe());
    ptr2->Scattering(versori, true);
```

Si può notare l'utilizzo di due array: **punto** e **versori**, sono le proprietà spaziali della mia particella che verranno "trasportate" attraverso due member function, della classe "Trasporto.h".
EquazioneRetta che ha il compito di portare il **punto** fino ad un certo raggio dato secondo le informazioni date da **versori**.
Scattering che come dice il nome modellizza lo scattering multiplo della particella contro gli strati che incontra, modificando solo **versori**.
Si vedranno in seguito le implementazioni di queste funzioni.

Il primo scontro con la Beam Pipe è inevitabile essendo sempre presente, invece lo scontro contro i Layer dei rivelatori non sono scontati avendo essi una certa lunghezza finita. Per questo motivo lungo il trasporto sono presenti degli if che mi dicono se la particella è uscita dall'accettanza del rivelatore o no.
Nel caso incrociasse il rivelatore, il punto di intersezione viene registrato nel TTree con un certo smearing, che va a simulare l'evento "vero".
Un esempio:

```
if (-H/2. <= punto[2] && punto[2] <= H/2.) {
    xhitL1.r = ptr2->GetRLayer1();
    xhitL1.phi = ptr2->SmearingPhi(punto[0], punto[1], ptr2->GetRLayer1());
    xhitL1.z = ptr2->SmearingZ(punto[2]);
```

[1] Nel sistema di riferimento del laboratorio per un collider sarebbe nulla la somma vettoriale degli impulsi.

```
        xhitL1.etichetta = tot;

        T_hitL1->Fill();
```

Infine per ogni scontro viene generato del rumore casuale, 2 hit per ogni Layer.

```
    ptr2->Rumore(&xhitL1, &xhitL2, T_hitL1, T_hitL2, tot, true);
```

Per salvare i dati sono stati usati dei TTree riempiti con delle struct definite nella classe "Defstruct.h", come per esempio:

```
    Vrt xvrt;

    ...

    TTree *T_vrt = new TTree("T_vrt","TTree della VM");
    T_vrt->Branch("vrt", &xvrt, "x0/D:y0/D:z0/D:moltiplicita/I");

    ...

    xvrt = {x0, y0, z0, molti};
    T_vrt->Fill();
```

Ogni vertice creato, corrisponde come già detto ad uno scontro tra bunch. Conoscendo la macchina acceleratrice sappiamo quale sia il rate di scontri e dunque riconosciamo temporalmente ogni hit, per avere sempre questa informazione ci portiamo dietro l'etichetta "tot" dello scontro. Questo sarà alla base del funzionamento per l'algoritmo di ricostruzione del vertice, poiché ci dirà come accoppiare la moltitudine di hit che abbiamo.

2.1 Defstruct

Prima di descrivere i metodi delle classi di generazione e trasporto, spendo due parole sul header file **Defstruct.h**. Ho definito nel header file delle struct che rappresentano le strutture dati principali del programma, in modo da poterle includere e riutilizzare in più macro senza duplicare il codice.

```
#ifndef DEFSTRUCT_H
#define DEFSTRUCT_H

    struct Hit {
        double r;
        double phi;
        double z;
        int etichetta;
    };

    struct Vrt {
        double x0;
        double y0;
        double z0;
        int moltiplicita;
    };

    struct Ric {
        double z0
        doule zRic;
        int molti;
    };

#endif
```

2.2 Generazione

In questa classe vengono implementate i metodi dedicati alla generazione del vertice di interazione e agli angoli sferici.

Per la molteplicità e la coordinata z si può scegliere il modello che descrive la distribuzione.
Per la z posso avere una gaussiana oppure una distribuzione uniforme. Per la molteplicità ho la distribuzione assegnata dal professore, una distribuzione uniforme oppure una molteplicità costante arbitraria.
La scelta avviene quando si utilizza la funzione FastSim(...) della macro FastSimulation.C.

```
double Generazione::VertexSimZ(bool distr) const{
    double z = 0;

    if (distr == true) {
        z = gRandom->Gaus(0., fRMSz);
    } else {
        z = gRandom->Uniform(-3*fRMSz, 3*fRMSz);
    }

    return z;
}
```

Ho scelto come range per la distribuzione uniforme [-3σ ,3σ], con σ la deviazione standard della gaussiana.

```
int Generazione::Multiplicity(bool distr, int nScelto) const{
    int Multi = 0;

    if (distr == true) {
        if (nScelto == 0)
            Multi = gRandom->Integer(53) + 3;
        else
            Multi = round(fHm->GetRandom());
    }
    else {
        Multi = nScelto;
    }

    return Multi;
}
```

2.3 Trasporto

In questa classe vengono implementate le funzioni per il trasporto della particella e per lo smearing della ricostruzione, entrambe caratteristiche proprie del rivelatore utilizzato.

Come detto precedentemente per ogni particella si ha il punto nello spazio ad un certo istante e le tre componenti dei versori che mi indicano la direzione. Con queste informazioni si può ottenere il punto di intersezione con un certo layer semplicemente con le equazioni parametrizzate della retta in 3 dimensioni (essendo nell'approssimazione ultra-relativistica).

Per lo scattering multiplo è più complicato e ci è stata fornita l'implementazione dal professore, nel mio caso l'ho modificata per usarla con un generico sistema di riferimento senza passare dagli angoli sferici iniziali.

```
void Trasporto::Scattering(double versore[3], bool on) const{
    if (!on) return;

    double T[3][3];

    double A = - versore[1]/TMath::Sqrt(versore[0]*versore[0] + versore[1]*versore[1]);
    double B =   versore[0]/TMath::Sqrt(versore[0]*versore[0] + versore[1]*versore[1]);

    T[0][0] = A;      T[0][1] = -versore[2]*B;                    T[0][2] = versore[0];
    T[1][0] = B;      T[1][1] = versore[2]*A;                     T[1][2] = versore[1];
    T[2][0] = 0.;     T[2][1] = (versore[0]*B - versore[1]*A);     T[2][2] = versore[2];

    double thp = fRMSspace;
    double php = gRandom->Rndm() * 2 * M_PI;

    double u[3];
    u[0] = TMath::Sin(thp) * TMath::Cos(php);
```

```
        u[1] = TMath::Sin(thp) * TMath::Sin(php);
        u[2] = TMath::Cos(thp);

        for (Int_t i = 0; i < 3; i++) {
            versore[i] = 0.;
            for (Int_t j = 0; j < 3; j++) {
                versore[i] += T[i][j] * u[j];
            }
        }
    }
```

2.4 DeltaPhi.C

Nella prossima Macro si utilizza l'unico valore della verità montecarlo che ci è permesso usare nel processo di ricostruzione del vertice, che è l'angolo di apertura tra la hit del layer 1 e quella del layer 2.
Come vederemo dopo per verificare che due hit siano collegate si utilizza questo angolo di apertura delta phi.
Ovviamente non dobbiamo usare per ogni particella il suo valore vero, ma dobbiamo trovare un certo valore statistico che vada bene per il nostro rivelatore.

Per fare ciò è stata ricreata una Macro identica a quella di FastSimulation.C, con l'unica differenza che si è creato un istogramma che va a contenere:

$$\Delta\phi = |\phi_{Layer_1} - \phi_{Layer_2}| \tag{1}$$

Ovviamente per tenere conto della ricostruzione, ai due angoli è stato precedentemente applicato lo smearing.

```
    if(-H/2. <= punto[2] && punto[2] <= H/2.) {
        double Phi1 = ptr3->SmearingPhi(punto[0], punto[1], ptr2->GetRLayer1());

        ptr2->Scattering(versori, true);
        ptr2->EquazioneRetta(punto, versori, ptr2->GetRLayer2());

        if(-H/2. <= punto[2] && punto[2] <= H/2.) {
            double Phi2 = ptr3->SmearingPhi(punto[0], punto[1], ptr2->GetRLayer2());

            double deltaPhi = TMath::Abs(Phi1 - Phi2);

            if (deltaPhi > M_PI) deltaPhi = 2 * M_PI - deltaPhi;

            hist->Fill(deltaPhi);
        }
    }
```

Vediamo alcuni risultati con diverse distribuzioni in generazione.

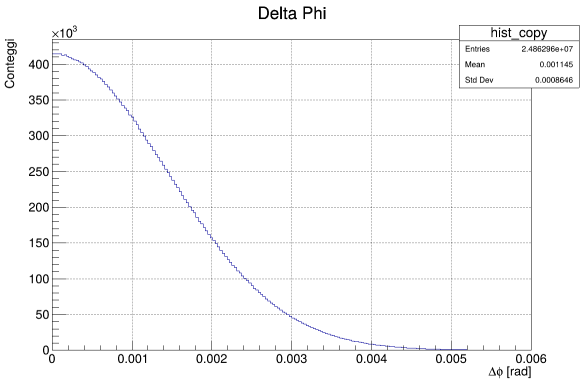


Figura 1: z Prof - m Prof

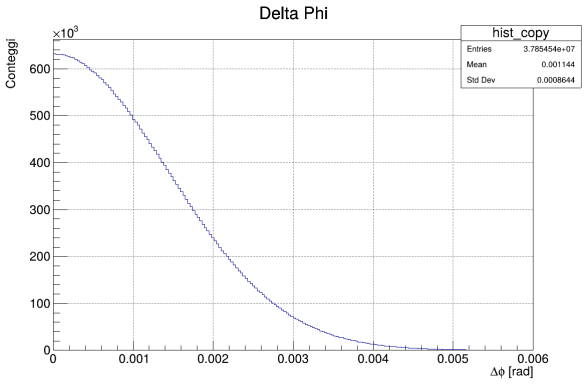


Figura 2: z Prof - m Uniforme

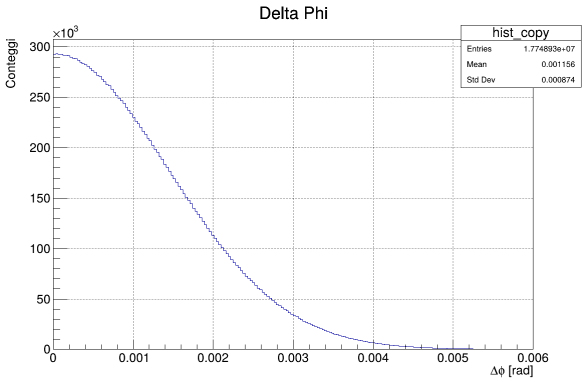


Figura 3: z Uniforme - m costante

Sulla base di questi risultati è stato scelto come valore massimo $\Delta\phi_{max} = 0.004$ rad.

3 Residui.C

Questa Macro svolge la ricostruzione dei vertici date le hit dei due layer.

Prima di parlare del cuore dell'algoritmo, bisogna mostrare un oggetto molto importante per tutto lo svolgimento.

```
    map<int, vector<Hit>> hitsByLabel;

    for (long i = 0; i < T_rec2->GetEntries(); ++i) {
        T_rec2->GetEntry(i);
        hitsByLabel[hitL2.etichetta].push_back(hitL2);
    }
```

La std::map è una struttura di dati che serve ad associare una chiave (nel nostro caso int = etichetta dello scontro) ad un valore (vettore di Hit, cioè un vettore di struct).
Dunque prendendo le informazioni dal TTree T_hitL2 le inserisco in questa struttura che può essere visualizzata come una tabella ordinata di vettori Hit per valore di etichetta.

Vediamo ora il ciclo for principale.

```
for(long ev = 0; ev < T_rec1->GetEntries(); ++ev){
    T_rec1->GetEntry(ev);

    if (lab != hitL1.etichetta){
        double media = ptr->RunWind(vertice);

        if (z_min <= media && media <= z_max) {
            T_MC->GetEntry(hitL1.etichetta - 2);

            hist[0]->Fill((media - mc.z0) * 1000);
            if (mc.moltiplicita == 6) hist[1]->Fill((media - mc.z0) * 1000);
            if (mc.moltiplicita > 44 && mc.moltiplicita < 56){
                hist[2]->Fill((media - mc.z0) * 1000);
            }

            xvtx.z0 = mc.z0;
            xvtx.molti = mc.moltiplicita;
            xvtx.zRic = media;
            T_vtx->Fill();
        }

        vertice.clear();
        lab = hitL1.etichetta;
```

```
    }

    double phi_L1 = hitL1.phi;

    map<int, vector<Hit>>::iterator it = hitsByLabel.find(hitL1.etichetta);
    if (it != hitsByLabel.end()) {
        for (const Hit& h : it->second) {
            double phi_L2 = h.phi;

            deltaPhi = TMath::Abs(phi_L1 - phi_L2);
            if (deltaPhi > TMath::Pi()) deltaPhi = 2 * TMath::Pi() - deltaPhi;

            if (deltaPhi <= Phi_VM) {
                double inter = ptr->Intersezione(hitL1.r, hitL1.z, h.r, h.z);
                if (inter >= z_min && inter <= z_max) {
                    vertice.push_back(inter);
                }
            }
        }
    }
}
```

Il for principale serve per scorrere le hit lungo il layer 1.
Per ogni hit del layer 1 si scorre la mappa alla ricerca delle hit del layer 2 tramite:

```
    map<int, vector<Hit>>::iterator it = hitsByLabel.find(hitL1.etichetta);
```

Per identificare l'etichetta giusta nella mappa si usa un iteratore (map::iterator), se trova l'etichetta cercata punterà ai vettori Hit salvati nella map.
Se però non trova nulla, possibile poiché il rivelatore ha una certa accettenza, mi restituisce un iteratore speciale (.end()), in questo caso l'if successivo mi salta il processo di ricostruzione e passa all'etichetta del layer 1 seguente.

Però per ogni corrispondenza di etichetta, devo scorrere tutti i corrispondenti hit del layer 2 e ottenere una stima del vertice iniziale.

```
    for (const Hit& h : it->second)
```

Questo ciclo for scorre ogni elemento associato alla chiave dell'iteratore e mi crea un riferimento costante h.
In questo processo si usa il valore di $\Delta\phi_{max}$ ottenuto dalla verità montecarlo, per avere una prima verifica di incompatibilità tra le hit dei due layer.
Se dov'esse risultare accettata la compatibilità si ha l'identificazione del vertice d'interazione tramite un semplice calcolo geometrico che vederemo successivamente.
Tutti i vertici ricostruiti in questo processo, vengono inseriti in un vettore **vertice**.
Una volta finito di considerare tutte le hit del layer 1 con la stessa etichetta, si passa all'etichetta successiva.
Nel passaggio incontro l'if che si accorge del cambio di etichetta, e mi avvia la running window sul vettore **vertice**, identificato precedentemente, per ottenere la miglior stima sul vertice ricostruito.
La stima del vertice ricostruito viene sottratta al valor vero della coordinata z per ottenere il residuo.
Questo procedimento avviene per tutte le entrate del layer 1, ed infine mi crea l'istogramma dei residui.

Vengono creati anche due istogrammi a molteplicità fissate per vedere come variano le caratteristiche della curva.

3.1 MyEs

In questa classe sono implementate le funzioni utilizzate nel processo di ricostruzione del vertice.

Dati due punti nello spazio 3-dimensionale sappiamo, assumendo che la traccia sia una retta, calcolare ogni punto.
A noi interessa "l'intersezione" con l'asse z, ma sappiamo che il vertice generato non è nato esattamente su quell'asse ma ha una certa distribuzione gaussiana lungo le direzioni x e y.

Pur avendo questa informazione possiamo comunque lavorare in un piano 2-dimensionale r-z, trascurando i gli spostamenti sulla x e y.

```
double MyEs::Intersezione(double r1, double z1, double r2, double z2){

    double m = (r2 - r1)/(z2 - z1);

    return z1 - r1/m;
}
```

Con queste approssimazioni si ottiene comunque un buon risultato. Avrei implementato un'altra funzione che tiene conto anche delle coordinate x e y delle hit nei layer ma come risultati sono del tutto compatibili con quelli del piano r-z (vedere Intersezione2()).

Una volta che si sono ottenuti tutti i vertici ricostruiti per una certa etichetta entra in azione la running window.

```
double MyEs::RunWind(const std::vector<double>& vertice){
    vector<double> pluto;
    if (vertice.empty()) {
        return 10000;
    }

    vector<double> v = vertice;
    sort(v.begin(), v.end());

    double W = 3.5;
    double mean = 0.;

    int n = (int)v.size();
    int j = 0;
    int best_i = 0, best_j = 0, best_count = 0;
    for (int i = 0; i < n; i++) {
        while (j < n && v[j] - v[i] <= W) j++;
        int count = j - i;
        if (count > best_count) {
            best_count = count;
            best_i = i;
            best_j = j;
        }
    }

    if (best_count < 2) {
        return 10000;
    }

    pluto.assign(v.begin() + best_i, v.begin() + best_j);

    mean = MediaVector(pluto);

    return mean;
}
```

Dato il vettore di intersezioni trovate in input, la running window mi scannerizza ogni entrata e mi mantiene i dati per cui si ha il maggior numero di eventi all'interno di una certa finestra spaziale.
Detto in maniera più comprensibile: ho assegnato alla finestra spaziale un certo valore sulla base dei vettori in input per ogni vertice, ho notato che con una larghezza di circa 3.5 mm si riesce a contenere interamente il gruppo più denso di intersezioni (cioè dove mi aspetto sia il valor vero).
Dopodiché prendo una ad una le entrate del vettore e creo una variabile j che mi dovrà portare dietro l'informazione della posizione sul vettore. Il ciclo while() serve per spostare a destra la j nel vettore finché non trova un elemento che dato un certo v[j] sia lontano da v[i] almeno 3.5mm. Una volta finito mi conta quante intersezioni ho in quell'intervallo e me lo salva come best_count se è maggiore di qualsiasi altro intervallo scannerizzato.

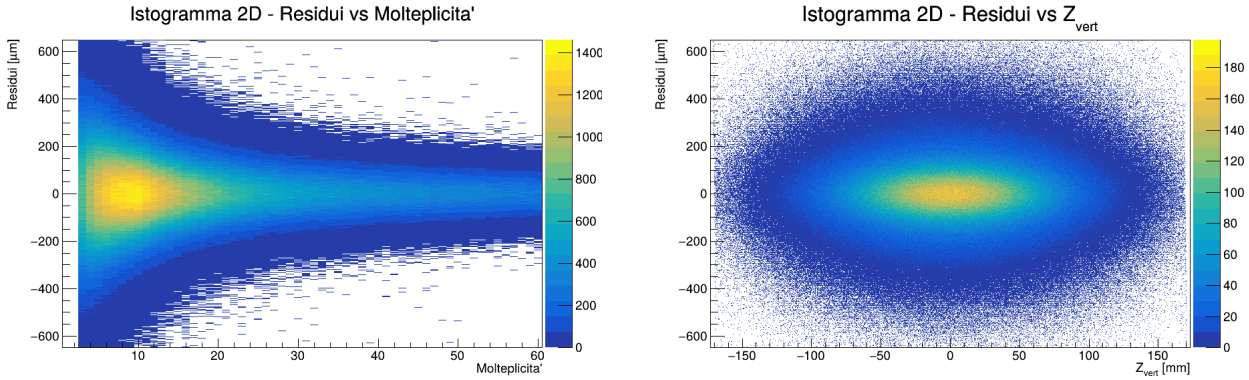
4 Risoluzione.C

In questa macro si valuta la risoluzione dei residui al variare della molteplicità e del vertice vero.
La risoluzione è estratta da istogrammi opportunamente riempiti tramite la member function di TH1 GetStdDev().

Per questo scopo si utilizzano i dati presenti nel TTree Tvtx riempito nella macro Residui.C, esso contiene per ogni vertice che siamo riusciti a ricostruire la stima del vertice, il vertice vero della generazione e la sua molteplicità originale.
Si riporta il processo principale:

```
TH2D *hist2D_M = new TH2D("hist2D_M","Istogramma 2D - Residui vs Molteplicita'",
                          60,0.5,60.5,600,-650.,650.);
hist2D_M->GetXaxis()->SetTitle("Molteplicita'");
hist2D_M->GetYaxis()->SetTitle("Residui [#mum]");
TH2D *hist2D_V = new TH2D("hist2D_V","Istogramma 2D - Residui vs Z_{vert}",
                          340,170.,170.,600,-650.,650.);
hist2D_V->GetXaxis()->SetTitle("Z_{vert} [mm]");
hist2D_V->GetYaxis()->SetTitle("Residui [#mum]");

const int nMolt = 11, nVer = 9;

double molteplicita[nMolt] = { 5., 7., 9., 12., 15., 18., 22., 26., 32., 40., 48. };
double deltaM[nMolt] = { 0.5, 0.5, 0.5, 0.5 ,1. ,1.  ,2.  ,2.  ,3.  ,3., 4. };

double vertice[nVer] = {-160., -120., -80., -40., 0., 40., 80., 120., 160.};
double deltaV[nVer] = {10., 10., 10., 10., 10., 10., 10., 10., 10.};

double y1[nMolt], y2[nVer], ey1[nMolt], ey2[nVer];

for (long ev = 0; ev < T_vtx->GetEntries(); ev++) {
    T_vtx->GetEntry(ev);
    hist2D_M->Fill(vertex.molti, (vertex.zRic - vertex.z0) * 1000);
    hist2D_V->Fill(vertex.z0, (vertex.zRic - vertex.z0) * 1000);
}

for (int i = 0; i < nMolt; i++) {

    int binM_min = hist2D_M->GetXaxis()->FindBin(molteplicita[i] - deltaM[i]);
    int binM_max = hist2D_M->GetXaxis()->FindBin(molteplicita[i] + deltaM[i]);

    TH1D *proj_M = hist2D_M->ProjectionY(Form("proj_M_%d", i), binM_min, binM_max);
    proj_M->SetDirectory(0);

    y1[i]  = proj_M->GetStdDev();
    ey1[i] = proj_M->GetStdDevError();

    delete proj_M;
}


for (int i = 0; i < nVer; i++) {

    int binV_min = hist2D_V->GetXaxis()->FindBin(vertice[i] - deltaV[i]);
    int binV_max = hist2D_V->GetXaxis()->FindBin(vertice[i] + deltaV[i]);

    TH1D *proj_V = hist2D_V->ProjectionY(Form("proj_V_%d", i),binV_min,binV_max);
    proj_V->SetDirectory(0);

    y2[i]  = proj_V->GetStdDev();
    ey2[i] = proj_V->GetStdDevError();

    delete proj_V;
}
```

Avendo bisogno di istogrammi con certi tagli sulla molteplicità e vertice vero, ho pensato di riempire istogrammi 2D e usare la funzione proiezione per crearli. Poiché di queste proiezioni non ero interessato a mantenerle in vita, dopo aver estratto la deviazione standard ho deciso di eliminarli con delete, ma prima era necessario prenderne la ownership (→SetDirectory(0)) per non creare puntatori dangling.
Riporto un esempio dei due istogrammi 2D.



I vettori riempiti nei due for sono stati usati per creare i 2 TGraphErrors finali.

5 Efficienza.C

In quest'ultima macro si è valutata l'efficienza di ricostruzione del vertice in funzione, come per la risoluzione, della molteplicità e del vertice vero.
L'efficienza è calcolata come:

$$\epsilon = \frac{\text{Numero vertici ricostruiti}}{\text{Numero vertici generati}} \tag{2}$$

Il numeratore è dato dai vertici ricostruiti in **Residui.C**, invece il denominatore è estratto dalla verità montecarlo.

Ho usato la classe TEfficiency di ROOT poiché valuta automaticamente l'errore da associare ad ogni punto, anche quando ho valori critici come $\epsilon \to 1$.
Un modo di adoperare questa classe è l'utilizzo di istogrammi, uno per gli eventi del numeratore e uno per gli eventi del denominatore della eq.2.
Gli istogrammi in funzione della molteplicità hanno delle larghezze dei bin variabili, per crearli è necessario dare in input nella definizione di TH1D un array (**edgesMolti**) che mi dai i bordi di ogni bin.
Si riporta una parte della macro dove si fa vedere la creazione degli istogrammi e del grafico dell'efficienza.

```
TH1F h_tot_M("h_tot_M","Totale;Molteplicita';Conteggi", nBinsMolti, edgesMolti);
TH1F h_pass_M("h_pass_M","Passati;Molteplicita';Conteggi", nBinsMolti, edgesMolti);
TH1F h_tot_V("h_tot_V","Totale;Vertice;Conteggi", nBinZ, minZ, maxZ);
TH1F h_pass_V("h_pass_V","Passati;Vertice;Conteggi", nBinZ, minZ, maxZ);

for (long ev = 0; ev < T_vrt->GetEntries(); ++ev) {
    T_vrt->GetEntry(ev);
    h_tot_M.Fill(MC.moltiplicita);
    h_tot_V.Fill(MC.z0);
}

for (long ev = 0; ev < T_vtx->GetEntries(); ++ev) {
    T_vtx->GetEntry(ev);
    h_pass_M.Fill(vertex.molti);
    h_pass_V.Fill(vertex.z0);
}

TEfficiency* eff = new TEfficiency(h_pass_M, h_tot_M);
eff->SetName("eff");
```

```
    eff->SetTitle("Efficienza vs molteplicita';Molteplicita';#epsilon");
    TEfficiency* eff_v = new TEfficiency(h_pass_V, h_tot_V);
    eff_v->SetName("eff_v");
    eff_v->SetTitle("Efficienza vs Z_{true};Z_{true} [mm];#epsilon");
```

6 All.C

Quest'ultima macro, come si evince dal nome, mi compila tutte e 4 le macro in automatico. Richiede in input da tastiera i parametri utili per la simulazione iniziale e in seguito usa il metodo ProcessLine della classe TRoot per eseguire su terminale le varie macro.

```
void All() {

    int n,m;
    unsigned int seed;
    bool distr_z,distr_m;
    cout<<"Inserire numero scontri:";
    cin>>n;
    cout<<"Inserire seed TRandom:";
    cin>>seed;
    cout<<"Inserire distribuzione in z [1=prof,0=uniforme]:";
    cin>>distr_z;
    cout<<"Inserire distribuzione molteplicit [1=continua,0=costante]:";
    cin>>distr_m;
    if(distr_m == true){
        cout<<"Inserire distribuzione molteplicit continua [1=prof,0=uniforme]:";
        cin>>m;
    }else{
        cout<<"Inserire la molteplicit costante:";
        cin>>m;
    }

    TString cmd =
        Form("FastSim(%d,%u,%d,%d,%d);", n, seed, (bool)distr_z, (bool)distr_m, m);

    gROOT->ProcessLine(".L Generazione.cpp++");
    gROOT->ProcessLine(".L Trasporto.cpp++");
    gROOT->ProcessLine(".L MyEs.cpp++");


    gROOT->ProcessLine(".L FastSimulation.C++");
    gROOT->ProcessLine(".L Residui.C++");
    gROOT->ProcessLine(".L Risoluzione.C++");
    gROOT->ProcessLine(".L Efficienza.C++");

    gROOT->ProcessLine(cmd);
    gROOT->ProcessLine("Residui();");
    gROOT->ProcessLine("Risoluzione();");
    gROOT->ProcessLine("Efficienza();");
}
```

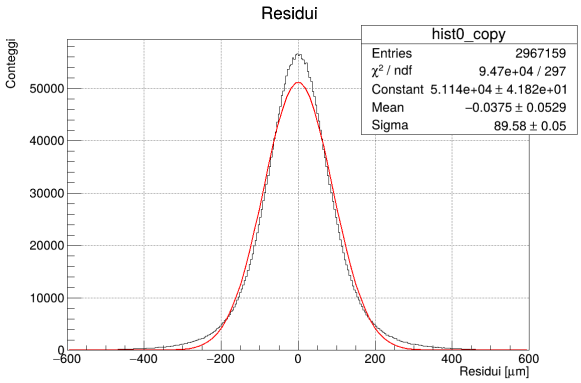
Per determinare quali distribuzioni usare per la z e la molteplicità si usano booleani, questi vengono scelti da tastiera tramite 1 o 0.

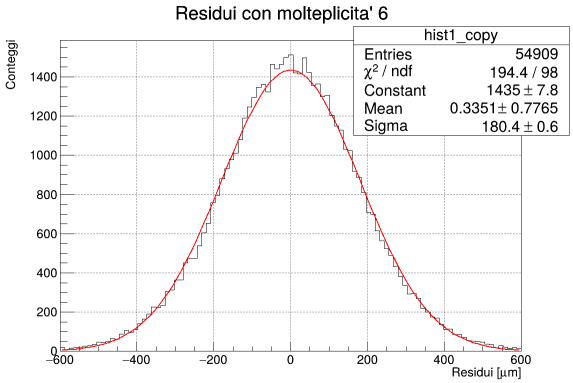
$$1 == \text{TRUE}$$
$$0 == \text{FALSE}$$

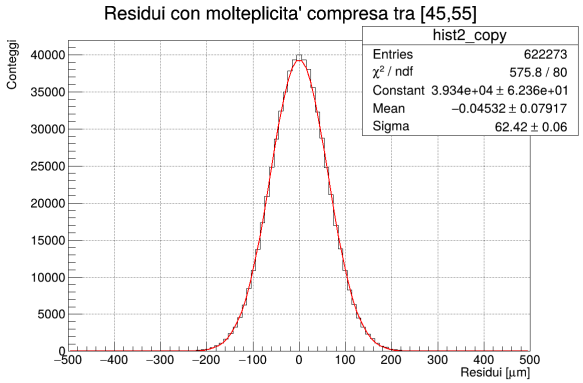
Il tutto in accordo con le funzioni della classe **Generazione.cpp**.

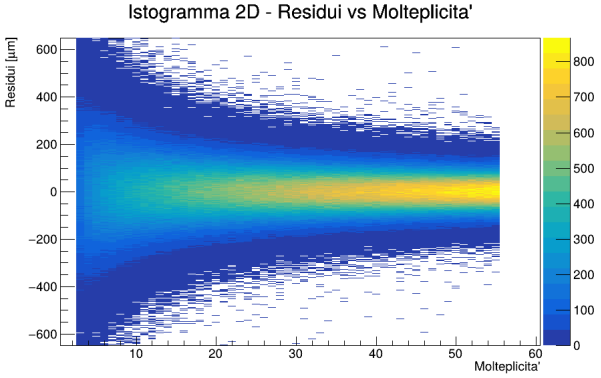
7 Esempi di Simulazioni

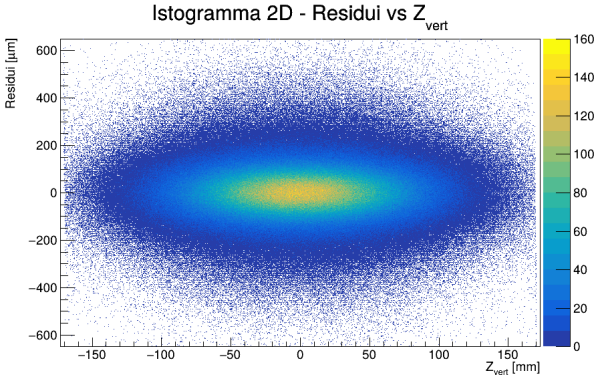
7.1 z Prof - m Uniforme - 3 mln eventi - seed 45771

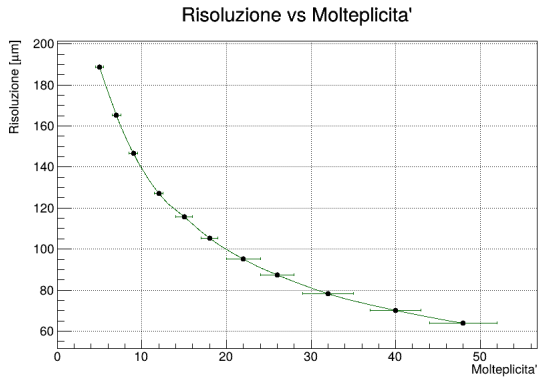


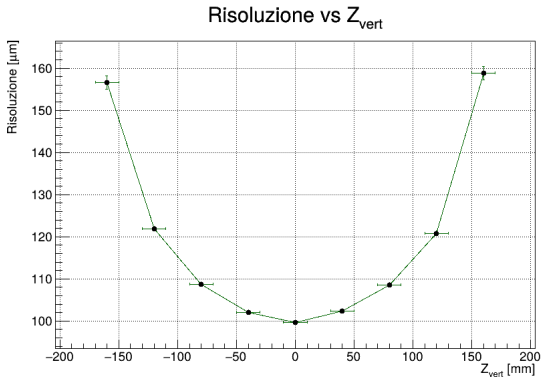


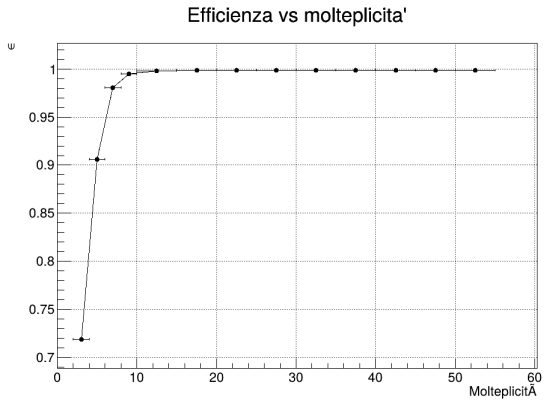


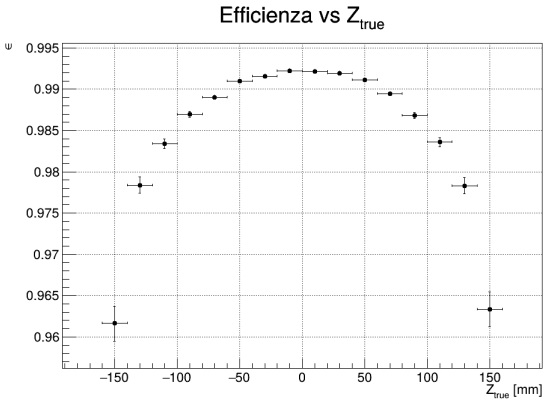




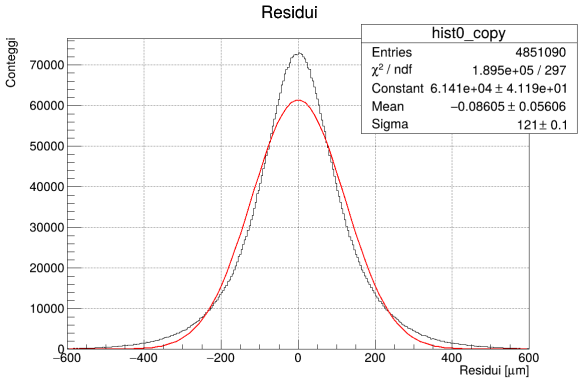

Risoluzione vs Molteplicita'
Risoluzione [µm]
Molteplicita'

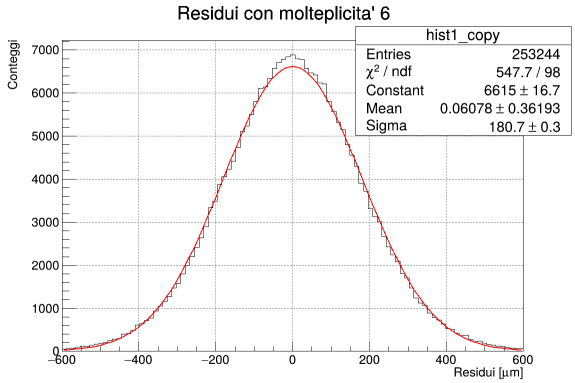

Risoluzione vs Z_{vert}
Risoluzione [µm]
Z_{vert} [mm]

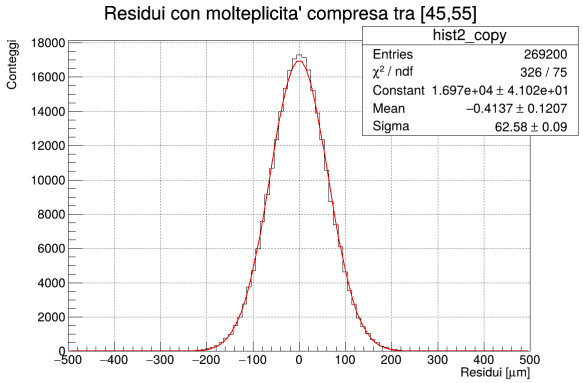

Efficienza vs molteplicita'
ε
MolteplicitÃ

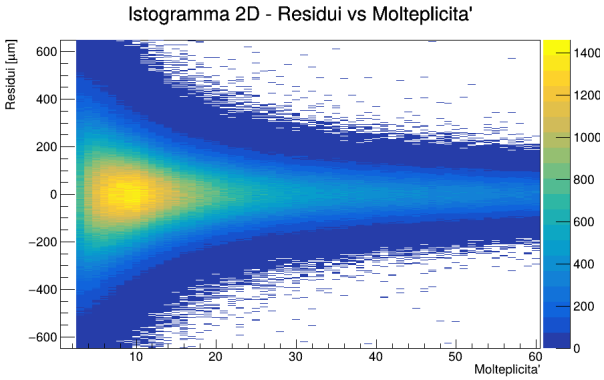

Efficienza vs Z_{true}
ε
Z_{true} [mm]

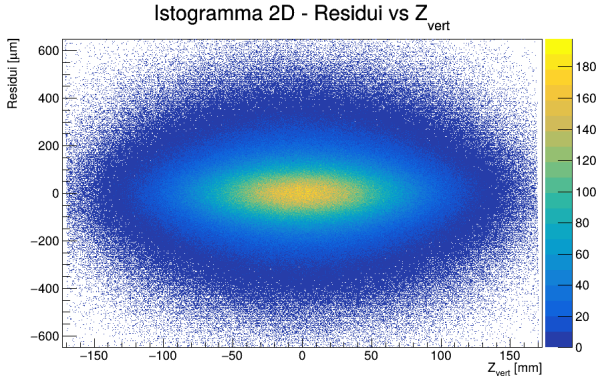
7.2 z Prof - m Prof - 5 mln eventi - seed 58485

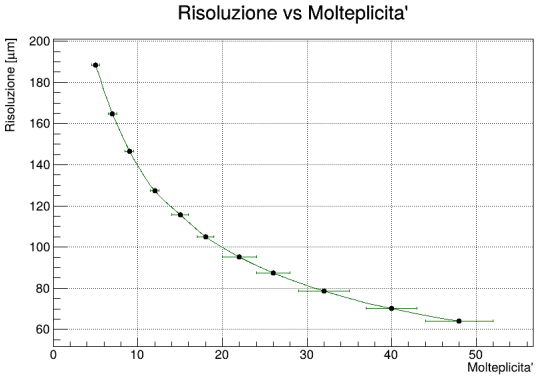


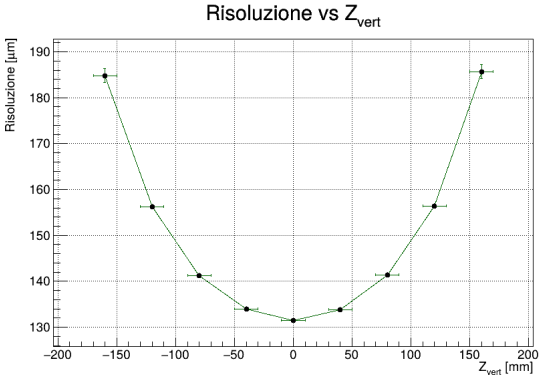


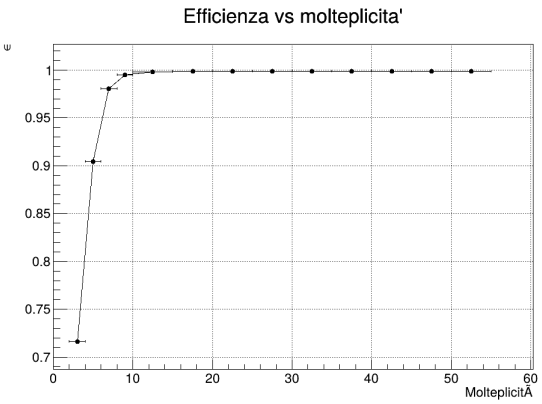


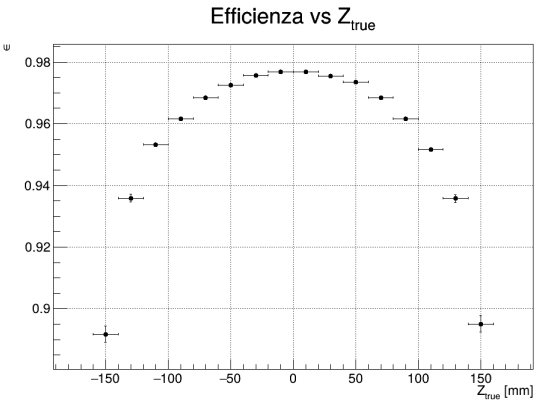




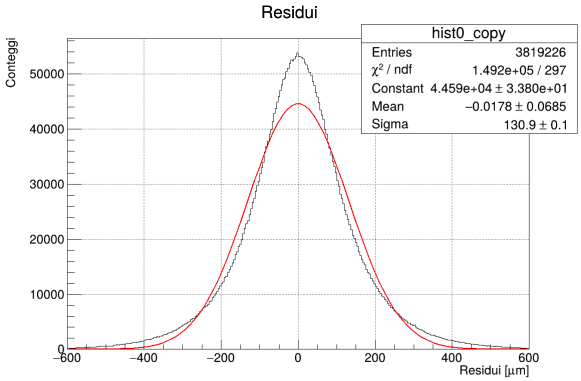

Risoluzione vs Molteplicita'
Risoluzione [µm]
Molteplicita'

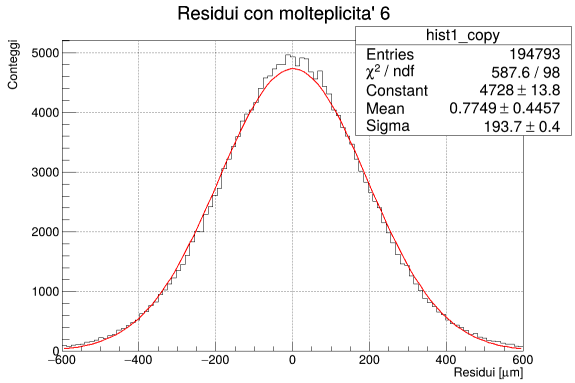

Risoluzione vs Z_vert
Risoluzione [µm]
Z_vert [mm]

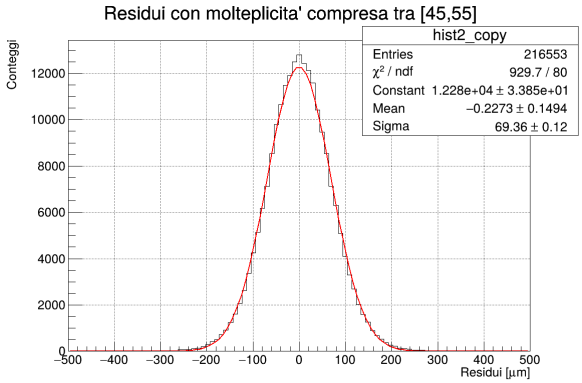

Efficienza vs molteplicita'
ε
MolteplicitÃ

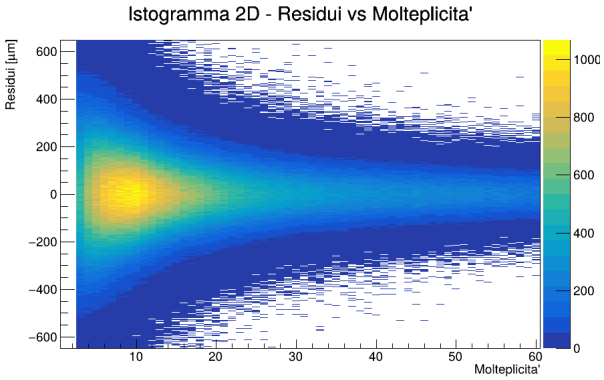

Efficienza vs Z_true
ε
Z_true [mm]

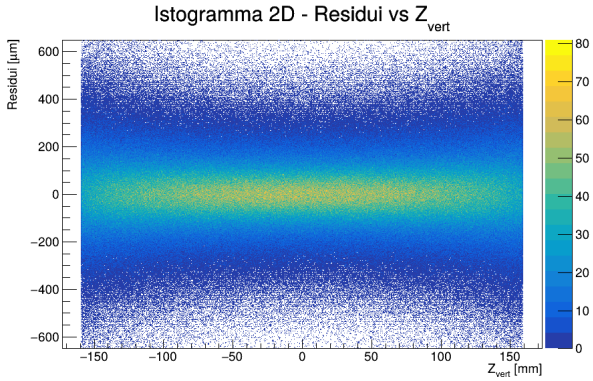
7.3 z Uniforme - m Prof - 4 mln eventi - seed 83923

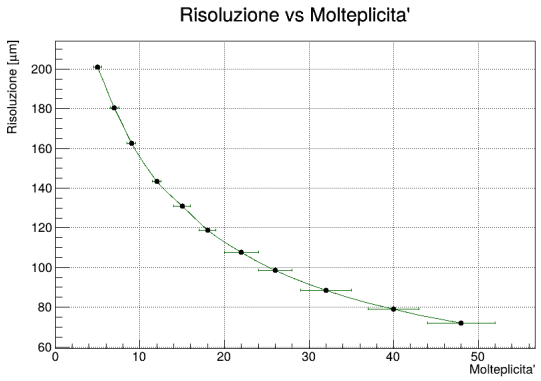


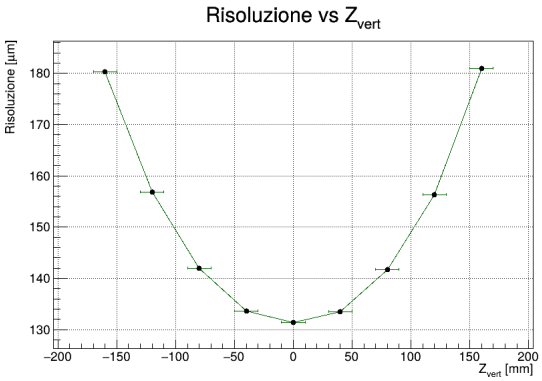


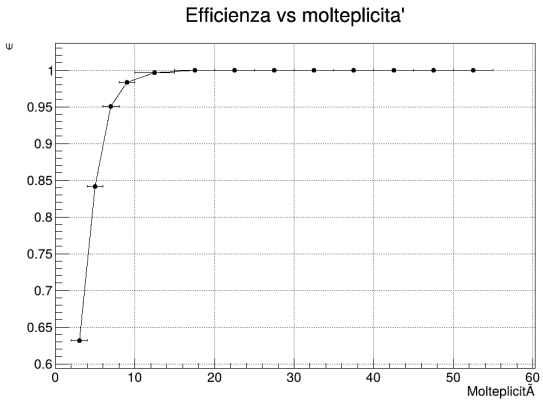


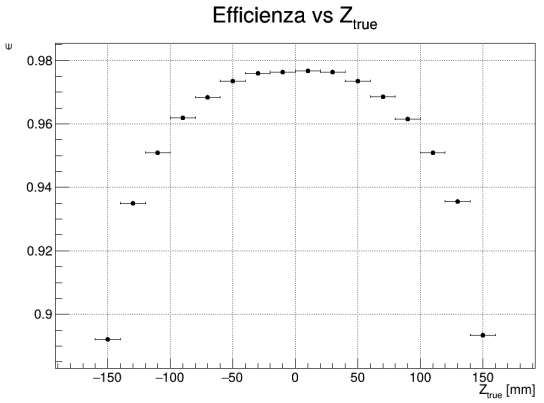




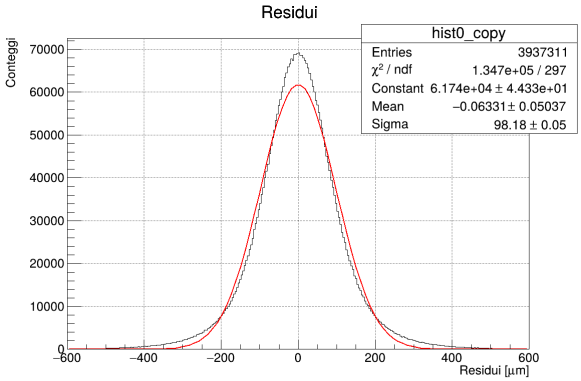

Risoluzione vs Molteplicita'
Risoluzione [µm]
Molteplicita'

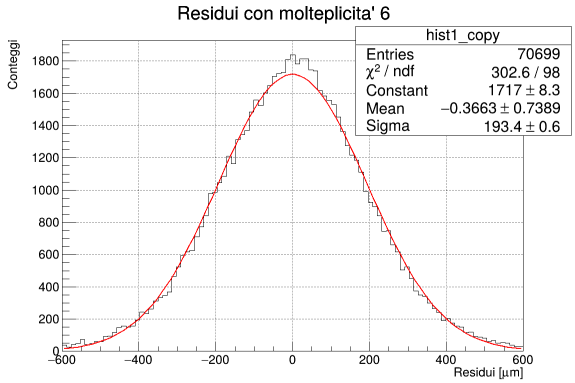

Risoluzione vs Z_vert
Risoluzione [µm]
Z_vert [mm]

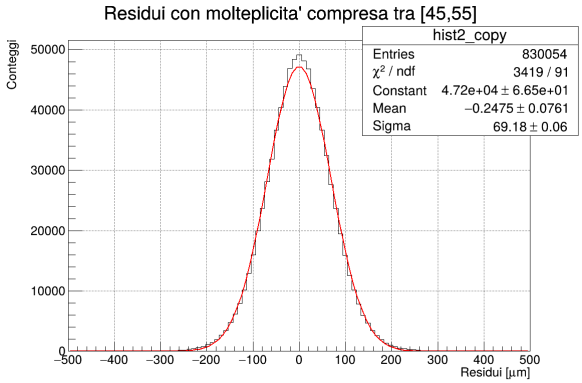

Efficienza vs molteplicita'
ε
MolteplicitÃ

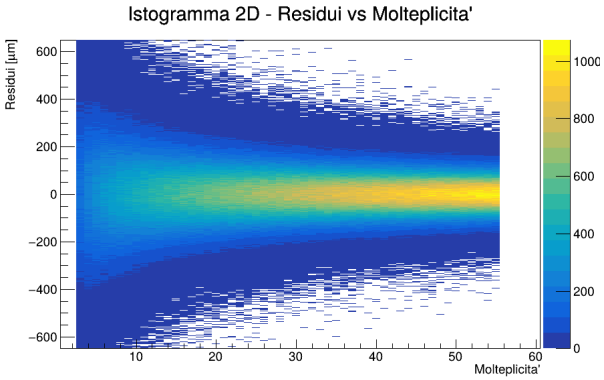

Efficienza vs Z_true
ε
Z_true [mm]

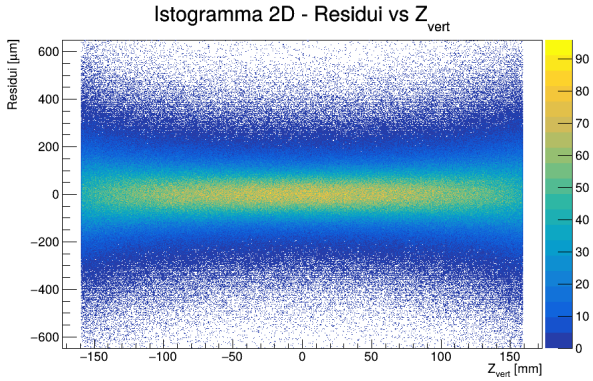
7.4 z Uniforme - m Uniforme - 4 mln eventi - seed 64585

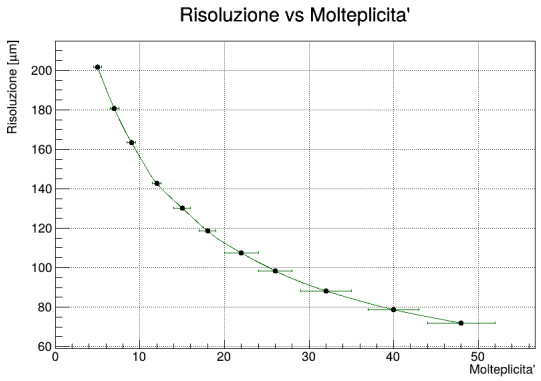


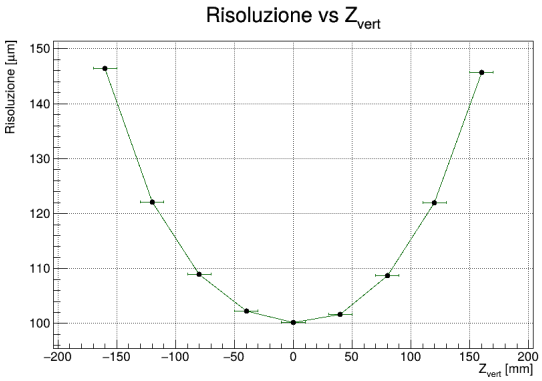


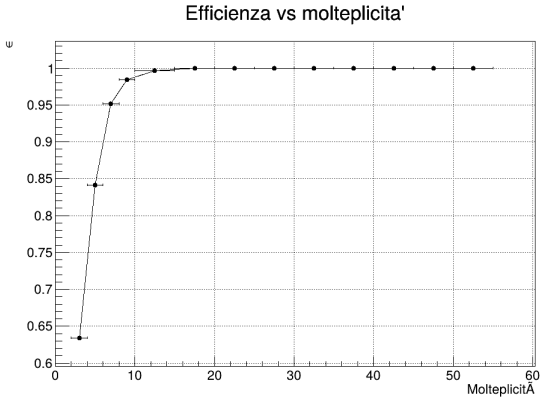


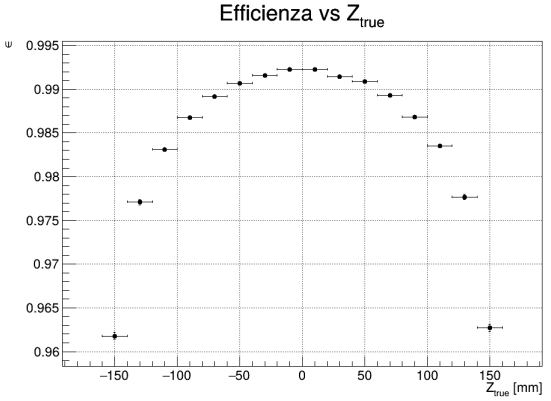





Risoluzione vs Molteplicita'
Risoluzione [µm]
Molteplicita'


Risoluzione vs Z_vert
Risoluzione [µm]
Z_vert [mm]


Efficienza vs molteplicita'
ε
MolteplicitÃ


Efficienza vs Z_true
ε
Z_true [mm]

7.5 z Prof - m costante 15 - 4 mln eventi - seed 69459