Gianluca Bianco

STUDIO E ANALISI DELLE COLLISIONI TRA PARTICELLE ELEMENTARI

INTRODUZIONE

Lo scopo del programma è quello di simulare la generazione di eventi fisici dati da urti tra particelle fondamentali. Dopo la generazione di questi eventi, un programma analizza i dati ottenuti tramite: la verifica del numero di generazioni effettuate e l'esecuzione dei fit sui grafici relativi ad ogni generazione. Gli strumenti principalmente usati sono la programmazione a oggetti tipica del c++ e le funzioni già presenti in Root.

STRUTTURA DEL CODICE

Per la corretta esecuzione del programma sono state implementate tre classi fondamentali: la classe *ParticleType*, la classe *ResonanceType* e la classe *Particle*.
La classe ParticleType ha lo scopo di creare delle istanze che portino con sé le tre caratteristiche fondamentali di una particella, ossia il nome, la massa e la carica.
La classe ResonanceType, erede della precedente, crea particelle che possiedono sia le caratteristiche della classe madre che l'attributo della larghezza della risonanza, per le particelle che ne possiedono una.
Nella classe Particle, invece, vengono implementate tra gli header file le classi precedenti. Essa ha lo scopo fondamentale di essere impiegata per la generazione degli eventi. Per fare ciò si avvale di alcuni metodi che le permettono di creare array di particelle, caratterizzate da un nome o da un numero, e dalle componenti dell'impulso. Inoltre è in grado di stabilire come debba avvenire il decadimento di particelle dotate di risonanza.
Per usufruire al meglio delle tre classi è stato allestito un programma che simuli effettivamente la generazione degli eventi, esso crea poi un file in cui vengono salvati tutti gli istogrammi che serviranno successivamente per l'analisi dei dati.
Gli istogrammi che ci interessano sono i seguenti:
- Tipo di particella.
- Distribuzione dell'angolo azimutale dell'impulso.
- Distribuzione dell'angolo polare dell'impulso.
- Distribuzione dell'impulso.
- Distribuzione dell'impulso trasverso.
- Distribuzione dell'energia.
- Distribuzione delle masse invarianti con varie combinazioni.

Infine è stata creata una funzione *Analisi* con lo scopo di disegnare i relativi istogrammi, analizzando la compatibilità con i dati attesi. Nello specifico sono stati eseguiti dei fit e delle operazioni tra grafici.

GENERAZIONE

Sono stati generati circa 10^5 eventi ognuno dei quali è costituito da 100 particelle. Circa l'80% delle particelle generate sono Pioni, un altro 10% sono Kaoni, un altro 9% Protoni e un restante 1% Risonanze. Queste risonanze sono state fatte decadere mediante il metodo Decay2Body, in combinazione tra Pioni e Kaoni. Per quanto riguarda le proprietà cinematiche sono stati utilizzati i seguenti metodi: una distribuzione uniforme per la coordinata azimutale (tra 0 e 2π), una

distribuzione uniforme per la coordinata polare (tra 0 e π) e una distribuzione esponenziale per l'impulso.

ANALISI

Di seguito sono raccolte le conclusioni ottenute mediante alcune tabelle.

Specie	Occorrenze Osservate	Occorrenze Attese
π+	$(3.989 \pm 0.002) \times 10^6$	4.000×10^6
π-	$(3.997 \pm 0.002) \times 10^6$	4.000×10^6
K+	$(0.506 \pm 0.001) \times 10^6$	0.500×10^6
K-	$(0.514 \pm 0.001) \times 10^6$	0.500×10^6
P+	$(0.448 \pm 0.001) \times 10^6$	0.450×10^6
P-	$(0.446 \pm 0.001) \times 10^6$	0.450×10^6
K*	$(0.995 \pm 0.003) \times 10^5$	0.100×10^6

Tabella 1: Confronto tra occorrenze attese e occorrenze osservate.

Distribuzione	Media ± Std	Parametro Fit	χ^2	DOF	χ^2 / DOF
Fit a distribuzione angolo polare	1.6 ± 0.9	M = 54 ± 13 q = (3.705 ± 0.002) x 10^4	370	267	1.4
Fit a distribuzione angolo azimutale	3.1 ± 1.8	m = -52 ± 7 q = (3.730 ± 0.002) x 10^4	431	267	1.6
Fit a distribuzione modulo impulso	1.003 ± 1.003	Cost = 13.12 Slope = (-0.9968 ± 0.0004)	1269	78	16.3

Tabella 2: χ^2 e χ^2 ridotto dei vari fit.

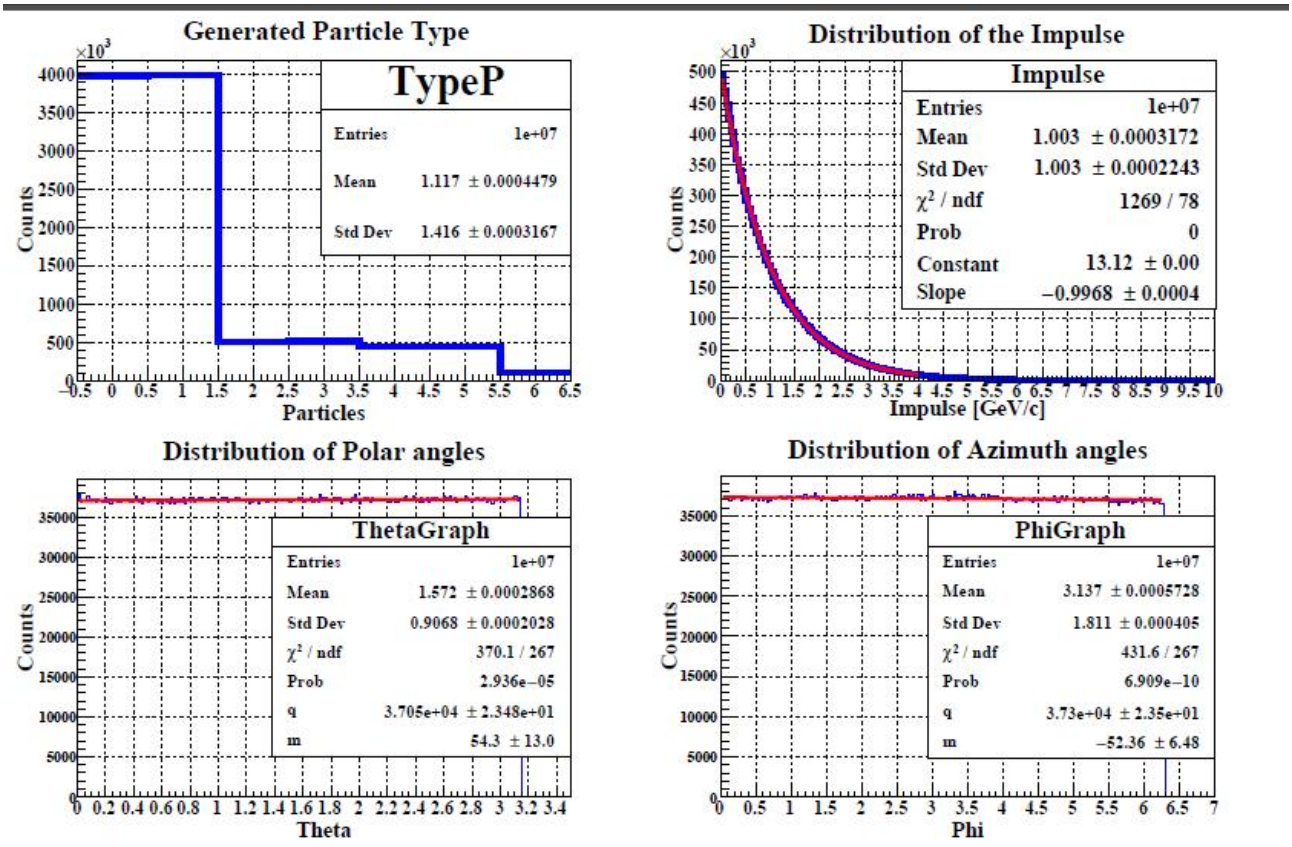
Per l'analisi del segnale della risonanza si è confrontato il risultato della sottrazione tra l'istogramma di soli Pioni e Kaoni con segno opposto e l'istogramma di soli Pioni e Kaoni con segno concorde e l'istogramma della Risonanza.

Distribuzione	Media	Sigma	Ampiezza	χ^2 / DOF
Massa Invariante vere k*	0.8371 ± 0.0007	0.1764 ± 0.0008	65.5 ± 0.4	3.2
Massa Invariante ottenuta da	0.6523 ± 0.0001	0.0198 ± 0.0001	658 ± 5	125

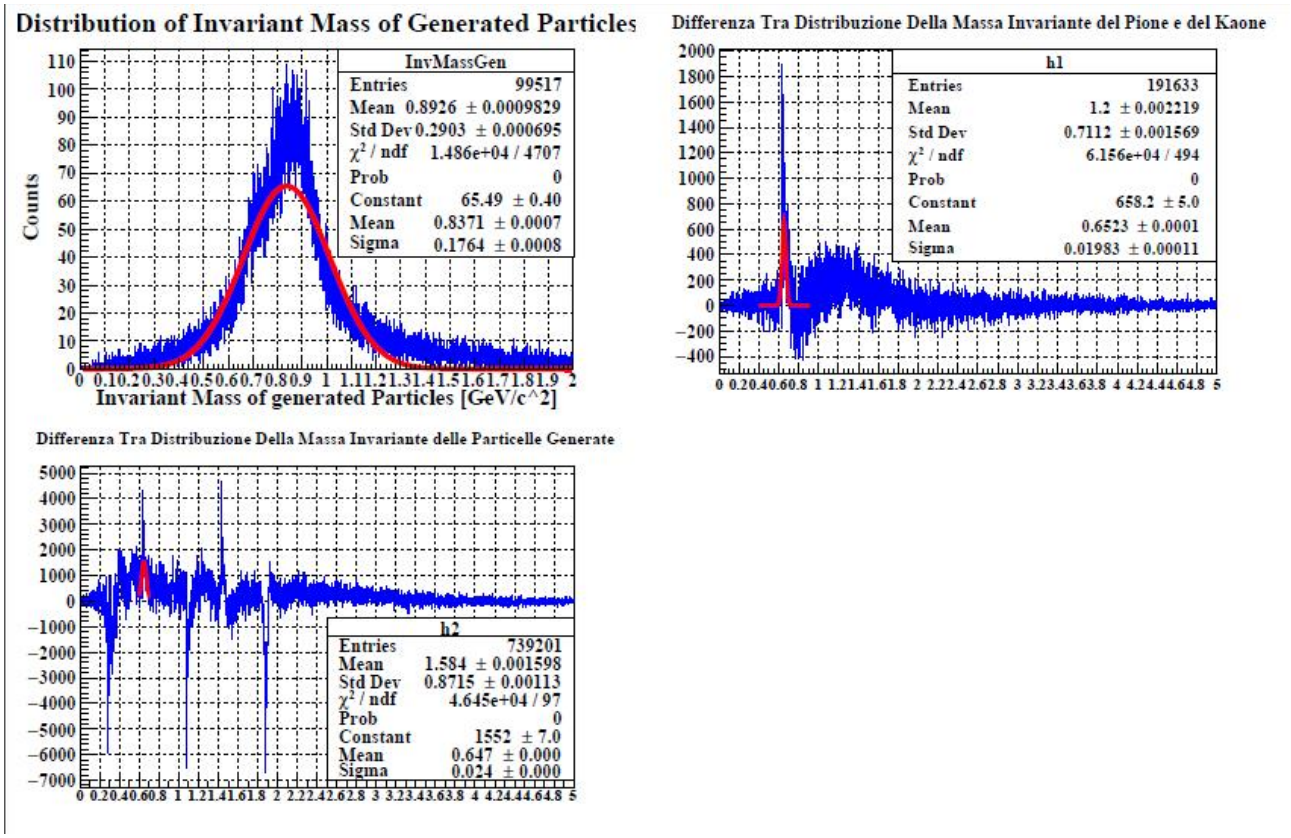
differenza delle combinazioni di carica discorde e concorde				
Massa Invariante ottenuta da differenza delle combinazioni Pioni e Kaoni con carica discorde e concorde	0.647 ± 0.000	0.024 ± 0.000	1552 ± 7	479

Tabella 3: Media, STD, Ampiezza e χ^2 ridotto delle varie combinazioni.

CANVAS A:



CANVAS B:



APPENDICE

Nella seguente appendice è riportata una copia del codice con le relative classi.

ParticleType.h

```
#ifndef PARTICLETYPE_H
#define PARTICLETYPE_H

class ParticleType
 {
  public:
  const char *GetName() const
   {
    return fName;
    };
  double GetMass() const
   {
    return fMass;
    };
  int GetCharge() const
   {
    return fCharge;
    };
  virtual double GetWidth() const
   {
```

```
   return 0;
   };
  virtual void Print();
  ParticleType (const char *Name, const double Mass, const int Charge);

  private:
  const char *fName;
  const double fMass;
  const int fCharge;
  };
```

#endif

ParticleType.c

```
#include <iostream>
#include "ParticleType.h"

using namespace std;

void ParticleType::Print()
 {
  cout << "Particle Name: " << fName << endl << "Particle Mass: " << fMass << endl << "Particle Charge: " << fCharge << endl;
  };

ParticleType::ParticleType(const char *Name, const double Mass, const int Charge):
fName (Name), fMass (Mass), fCharge (Charge) {};
```

ResonanceType.h

```
#ifndef RESONANCETYPE_H
#define RESONANCETYPE_H

#include "ParticleType.h"

class ResonanceType : public ParticleType
 {
  public:
  double GetWidth() const
   {
    return fWidth;
    };
  virtual void Print();
  ResonanceType (const char *Name, const double Mass, const int Charge, const double Width);

  private:
  const double fWidth;
  };

#endif
```

ResonanceType.c

```
#include <iostream>
#include "ResonanceType.h"

using namespace std;

void ResonanceType::Print()
 {
  ParticleType::Print();
  cout << "Particle Width: " << fWidth << endl;
  };
```

```
ResonanceType::ResonanceType(const char *Name, const double Mass, const int Charge, const double Width):
ParticleType (Name, Mass, Charge), fWidth (Width) {};
```

Particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include "ParticleType.h"
#include "ResonanceType.h"

class Particle
 {
  public:
  Particle();
  Particle (const char *Name, double Px = 0., double Py = 0., double Pz = 0.);
  Particle (int IParticle, double Px = 0., double Py = 0., double Pz = 0.);

  int GetIParticle() const
  {
   return fIParticle;
   };
   double GetPx() const
   {
    return fPx;
    };
  double GetPy() const
   {
    return fPy;
    };
  double GetPz() const
   {
    return fPz;
    };
  double GetMass() const;
  double GetCharge() const;
  double GetEnergy() const;
  double GetInvMass(Particle &) const;

  void SetParticle (int IParticle);
  void SetParticle (const char *Name);
  void SetP (double, double, double);

  static int AddParticleType (const char *Name, const double Mass, const int Charge, const double Width);
  int Decay2body(Particle &dau1,Particle &dau2) const;
  static void Printer();
  void Printex(int IParticle, const char *Name);

  static const int fMaxNumParticleType = 10;

  private:
  static ParticleType *fParticleType[fMaxNumParticleType];
  static int fNParticleType;
  int fIParticle;
  double fPx, fPy, fPz;
  static int FindParticle (const char *Name);
  void Boost(double Bx, double By, double Bz);
  };

#endif
```

Particle.c

```
#include <iostream>
#include <cmath>
#include <cstdlib>
```

```
#include "Particle.h"

using namespace std;

int Particle::fNParticleType = 0;

ParticleType* Particle::fParticleType [Particle::fMaxNumParticleType];

int Particle::FindParticle (const char *Name)
 {
  for (int i = 0; i < fNParticleType; i++)
   {
    const char *type = fParticleType[i]->GetName();
    int k = 0;
    while((type[k] == Name[k]) && type[k] != '\0' && Name[k] != '\0')
     {
      k++;
     }
    if(type[k] == Name[k]) return i;
    }
  return -1;
  }

Particle::Particle ():
fIParticle (-1), fPx (0), fPy(0), fPz(0)
{};

Particle::Particle (const char *Name, double Px, double Py, double Pz):
fPx (Px), fPy (Py), fPz (Pz)
 {
  int FP = FindParticle(Name);
  if(FP != -1)
   {
    fIParticle = FP;
    }
  else
   {
    cout << "La particella non è presente nella casella numero: " << fParticleType[fIParticle]->GetName() << endl;
    fIParticle = -1;
    }
  }

Particle::Particle (int IParticle, double Px, double Py, double Pz):
fPx (Px), fPy (Py), fPz (Pz)
 {
  if (IParticle < fNParticleType && IParticle >= 0)
   {
    fIParticle = IParticle;
    }
  else
   {
    cout <<"La particella non è presente nella casella numero: " << IParticle <<endl;
    }
  fIParticle = -1;
  }


double Particle::GetMass() const
 {
  if(fIParticle > -1)
   {
    return fParticleType[fIParticle]->GetMass();
    }
  else
   {
    return 0.;
    }
  }

double Particle::GetCharge() const
```

```
 {
  if(fIParticle > -1)
   {
    return fParticleType[fIParticle]->GetCharge();
   }
  else
   {
    return 0.;
   }
  }

double Particle::GetEnergy() const
 {
  double Mass = GetMass();
  return sqrt(fPx*fPx + fPy*fPy + fPz*fPz + Mass*Mass);
  }

double Particle::GetInvMass (Particle & p) const
 {
  return sqrt(pow(GetEnergy()+p.GetEnergy(),2) -
              (pow(fPx+p.GetPx(),2) +
               pow(fPy+p.GetPy(),2) +
               pow(fPz+GetPz(),2)
               )
              );
  }

void Particle::SetParticle (int IParticle)
 {
  if(IParticle >= 0 && IParticle < fNParticleType)
   {
    fIParticle = IParticle;
   }
  else
   {
    cout << "La particella non esiste: " << IParticle << endl;
   }
  }

void Particle::SetParticle(const char *Name)
 {
  int FP = FindParticle(Name);
  if(FP != -1)
   {
    fIParticle = FP;
   }
  else
   {
    cout << "La particella non è presente nella casella numero: " << fParticleType[fIParticle]->GetName() << endl;
   }
  }

void Particle::SetP (double Px, double Py, double Pz)
 {
  fPx = Px;
  fPy = Py;
  fPz = Pz;
  }

int Particle::AddParticleType(const char *Name,double Mass,int Charge,double Width)
 {
  if(fNParticleType < fMaxNumParticleType)
   {
    int FP = FindParticle(Name);
    if(FP != -1)
     {
      cout << "Esiste già una particella con questo nome: " << Name << endl; return 7;
     }
    if(Width > 0)
     {
```

```
    fParticleType[fNParticleType] = new ResonanceType(Name, Mass, Charge, Width);
    }
   else
    {
    fParticleType[fNParticleType] = new ParticleType(Name, Mass, Charge);
    }
   fNParticleType++;
   }
  else
   {
   cout << "Il contenitore è pieno perchè hai inserito " << fMaxNumParticleType << " particelle" << endl; return 6;
   }
  return 5;
  }

 void Particle::Boost(double Bx, double By, double Bz)
  {
  double energy = GetEnergy();
  double B2 = Bx*Bx + By*By + Bz*Bz;
  double gamma = 1.0 / sqrt(1.0 - B2);
  double Bp = Bx*fPx + By*fPy + Bz*fPz;
  double gamma2 = B2 > 0 ? (gamma - 1.0)/B2 : 0.0;
  fPx += gamma2*Bp*Bx + gamma*Bx*energy;
  fPy += gamma2*Bp*By + gamma*By*energy;
  fPz += gamma2*Bp*Bz + gamma*Bz*energy;
  }

 int Particle::Decay2body(Particle &dau1,Particle &dau2) const
  {
  if(GetMass() == 0.0)
   {
   cout << "Se la massa è zero non c'è nessun decadimento." << endl; return 1;
   }
  double massMot = GetMass();
  double massDau1 = dau1.GetMass();
  double massDau2 = dau2.GetMass();
  if(fIParticle > -1)
   {
   float x1, x2, w, y1, y2;
   double invnum = 1./RAND_MAX;
   do
    {
    x1 = 2.0 * rand()*invnum - 1.0;
    x2 = 2.0 * rand()*invnum - 1.0;
    w = x1 * x1 + x2 * x2;
    }
   while (w >= 1.0);
   w = sqrt((-2.0 * log( w )) / w );
   y1 = x1 * w;
   y2 = x2 * w;
   massMot += fParticleType[fIParticle]->GetWidth() * y1;
   }
  if(massMot < massDau1 + massDau2)
   {
   cout << "Il decadimento non può avvenire perchè la massa è troppo piccola" << endl; return 2;
   }
  double pout = sqrt((massMot*massMot - (massDau1+massDau2)*(massDau1+massDau2)) *
                     (massMot*massMot - (massDau1-massDau2)*(massDau1-massDau2))
                     ) /
                massMot*0.5;
  double norm = 2*M_PI/RAND_MAX;
  double phi = rand()*norm;
  double theta = rand()*norm*0.5 - M_PI/2.;
  dau1.SetP(pout*sin(theta)*cos(phi), pout*sin(theta)*sin(phi), pout*cos(theta));
  dau2.SetP(-pout*sin(theta)*cos(phi), -pout*sin(theta)*sin(phi), -pout*cos(theta));
  double energy = sqrt(fPx*fPx + fPy*fPy + fPz*fPz + massMot*massMot);
  double Bx = fPx/energy;
  double By = fPy/energy;
  double Bz = fPz/energy;
  dau1.Boost(Bx,By,Bz);
```

```
  dau2.Boost(Bx,By,Bz);
  return 0;
  }

void Particle::Printer()
 {
  for (int j = 0; j < fNParticleType; j++)
   {
    fParticleType[j]->Print();
   }
  }

void Particle::Printex (int IParticle, const char *Name)
 {
  cout << "Particle Type: " << fIParticle << endl << "Particle Name: " << fParticleType[fIParticle]->GetName() << endl << "Particle
P: " << "(" << fPx << ", "  << fPy << ", " << fPz << ")" << endl;
  };
```

Main.c

```
#include <iostream>
#include <cmath>
#include "ParticleType.h"
#include "ResonanceType.h"
#include "Particle.h"
#include "TRandom.h"
#include "TBenchmark.h"
#include "TH1F.h"
#include "TStyle.h"
#include "TH1D.h"
#include "TFile.h"

using namespace std;

//gBenchmark->Start("Bench");

int Main ()
 {
  Particle::AddParticleType ("Pione+", 0.13957, 1, 0);
  Particle::AddParticleType ("Pione-", 0.13957, -1, 0);
  Particle::AddParticleType ("Kaone+", 0.49367, 1, 0);
  Particle::AddParticleType ("Kaone-", 0.49367, -1, 0);
  Particle::AddParticleType ("Protone+", 0.93827, 1, 0);
  Particle::AddParticleType ("Protone-", 0.93827, -1, 0);
  Particle::AddParticleType ("K*", 0.89166, 0, 0.050);

  Particle::Printer();

  Double_t y = 0, phi, theta, P, Px, Py, Pz, CrossP = sqrt(pow(Px, 2) + pow(Py, 2));

  const Int_t N = 140, a = 1E5, b = 100, part = 7;

  Particle Particella[N];

  Int_t NK = 0, Over = 100+(2*NK), OverPlus = 100+(2*NK)+1, ArrayPos = 0;

  TH1F *TypeP = new TH1F ("TypeP", "Generated Particle Type", part, -0.5, -0.5 + part);
  TH1F *PhiGraph= new TH1F ("PhiGraph", "Distribution of Azimuth angles", 300, 0, 7);
  TH1F *ThetaGraph = new TH1F ("ThetaGraph", "Distribution of Polar angles", 300, 0, 3.5);
  TH1F *Impulse = new TH1F ("Impulse", "Distribution of the Impulse", 200, 0, 10);
  TH1F *CrossImpulse = new TH1F ("CrossImpulse", "Distribution of the Cross Impulse", 200, 0, 10);
  TH1F *Energy = new TH1F ("Energy", "Distribution of Energy", 200, 0, 10);
  TH1F *InvariantMass = new TH1F ("InvariantMass", "Distribution of Invariant Mass", 600, 0, 5);
  TH1F *InvMassDisc = new TH1F ("InvMassDisc", "Distribution of Invariant Mass with discordant charge sign", 5000, 0, 5);
  TH1F *InvMassConc = new TH1F ("InvMassConc", "Distribution of Invariant Mass with concordant charge sign", 5000, 0, 5);
  TH1F *InvMassPKDisc = new TH1F ("InvMassPKDisc", "Distribution of Invariant Mass of discordant Pions and Kions", 5000, 0,
5);
```

```
  TH1F *InvMassPKConc = new TH1F ("InvMassPKConc", "Distribution of Invariant Mass of concordant Pions and Kions", 5000,
0, 5);
  TH1F *InvMassGen = new TH1F ("InvMassGen", "Distribution of Invariant Mass of Generated Particles", 5000, 0, 2);

  for (Int_t i = 0; i < a; i++)
   {
    for (Int_t j = 0; j < b; j++)
     {
      gRandom->SetSeed();
      phi = gRandom->Uniform(0, (2*M_PI));
      theta = gRandom->Uniform(0, M_PI);
      P = gRandom->Exp(1);
      Px = P*sin(theta)*cos(phi);
      Py = P*sin(theta)*sin(phi);
      Pz = P*cos(theta);
      Particella[j].SetP(P*sin(theta)*cos(phi), P*sin(theta)*sin(phi), P*cos(theta));
      PhiGraph->Fill(phi);
      ThetaGraph->Fill(theta);
      Impulse->Fill(P);
      CrossImpulse->Fill(CrossP);
      y = gRandom->Rndm();

      if (y < 0.4)
       {
        Particella[j].SetParticle ("Pione+");
        }
      else if (y < 0.8)
       {
        Particella[j].SetParticle ("Pione-");
        }
      else if (y < 0.85)
       {
        Particella[j].SetParticle ("Kaone+");
        }
      else if (y < 0.9)
       {
        Particella[j].SetParticle ("Kaone-");
        }
      else if (y < 0.945)
       {
        Particella[j].SetParticle ("Protone+");
        }
      else if (y < 0.99)
       {
        Particella[j].SetParticle ("Protone-");
        }
      else
       {
        Particella[j].SetParticle ("K*");
        if(gRandom->Rndm() < 0.5)
         {
          Particella[Over].SetParticle("Pione-");
          Particella[OverPlus].SetParticle("Kaone+");
          Particella[j].Decay2body(Particella[Over], Particella[OverPlus]);
          }
        else
         {
          Particella[Over].SetParticle("Pione+");
          Particella[OverPlus].SetParticle("Kaone-");
          Particella[j].Decay2body(Particella[Over], Particella[OverPlus]);
          }
        InvMassGen->Fill(Particella[Over].GetInvMass(Particella[OverPlus]));
        NK++;
        }

      Energy->Fill(Particella[j].GetEnergy());
      TypeP->Fill(Particella[j].GetIParticle());
      }

    for (Int_t w = 0; w < Over; w++)
```

```
    {
     for(Int_t u = w + 1; u < OverPlus; u++)
      {
       if (
          (Particella[w].GetIParticle() != 6) && (Particella[u].GetIParticle() != 6)
          )
        {
         InvariantMass->Fill(Particella[w].GetInvMass(Particella[u]));

         if (
            Particella[w].GetCharge() !=
            Particella[u].GetCharge()
            )
          {
           InvMassDisc->Fill(Particella[w].GetInvMass(Particella[u]));
           }
         if (
            Particella[w].GetCharge() ==
            Particella[u].GetCharge()
            )
          {
           InvMassConc->Fill(Particella[w].GetInvMass(Particella[u]));
           }
         if (
            ((Particella[w].GetIParticle() == 0) && (Particella[u].GetIParticle() == 3)) ||
            ((Particella[w].GetIParticle() == 1) && (Particella[u].GetIParticle() == 2))
            )
          {
           InvMassPKDisc->Fill(Particella[w].GetInvMass(Particella[u]));
           }
         if (
            ((Particella[w].GetIParticle() == 0) && (Particella[u].GetIParticle() == 2)) ||
            ((Particella[w].GetIParticle() == 1) && (Particella[u].GetIParticle() == 3))
            )
          {
           InvMassPKConc->Fill(Particella[w].GetInvMass(Particella[u]));
           }
         }
        }
       }
      }

  TFile * File = new TFile( "Istogrammi.root", "RECREATE" );
  File->cd();
  TypeP->Write();
  PhiGraph->Write();
  ThetaGraph->Write();
  Impulse->Write();
  CrossImpulse->Write();
  Energy->Write();
  InvariantMass->Write();
  InvMassDisc->Write();
  InvMassConc->Write();
  InvMassPKDisc->Write();
  InvMassPKConc->Write();
  InvMassGen->Write();
  File->Close();
  File->ls();
  return 0;

  }

//gBenchmark->Show("Bench");
```

Analisi.c

```
#include <iostream>
#include <cmath>
```

```
#include "ParticleType.h"
#include "ResonanceType.h"
#include "Particle.h"
#include "TRandom.h"
#include "TBenchmark.h"
#include "TH1F.h"
#include "TStyle.h"
#include "TH1D.h"
#include "TFile.h"
#include "TCanvas.h"
#include "TF1.h"
#include "TStyle.h"
#include "TLegend.h"

double fi (double *x, double *y)
{return y[0] + x[0] * y[1];}

void Analisi ()
{
TFile *file = new TFile("Istogrammi.root");
file->ls();

TH1D * type = (TH1D *)file->Get("TypeP");
type->GetXaxis()->SetTitle("Particles");
type->GetXaxis()->CenterTitle(true);
type->GetYaxis()->SetTitle("Counts");
type->SetFillColor(kGreen-9);

TH1D * phi = (TH1D *)file->Get("PhiGraph");
phi->GetXaxis()->SetTitle("Phi");
phi->GetYaxis()->SetTitle("Counts");

TH1D * theta = (TH1D *)file->Get("ThetaGraph");
theta->GetXaxis()->SetTitle("Theta");
theta->GetYaxis()->SetTitle("Counts");

TH1D * impulse = (TH1D *)file->Get("Impulse");
impulse->GetXaxis()->SetTitle("Impulse [GeV/c]");
impulse->GetYaxis()->SetTitle("Counts");

TH1D * cross = (TH1D *)file->Get("CrossImpulse");
cross->GetXaxis()->SetTitle("Cross Impulse [GeV/c]");
cross->GetYaxis()->SetTitle("Counts");

TH1D * energy = (TH1D *)file->Get("Energy");
energy->GetXaxis()->SetTitle("Energy [GeV/c^2]");
energy->GetYaxis()->SetTitle("Counts");
energy->SetFillColor(kCyan-7);

TH1D * inv_mass = (TH1D *)file->Get("InvariantMass");
inv_mass->GetXaxis()->SetTitle("Invariant Mass [GeV/c^2]");
inv_mass->GetYaxis()->SetTitle("Counts");
inv_mass->SetFillColor(kRed-9);

TH1D * inv_mass_discorde = (TH1D *)file->Get("InvMassDisc");
inv_mass_discorde->GetXaxis()->SetTitle("Invariant Discord Mass [GeV/c^2]");
inv_mass_discorde->GetYaxis()->SetTitle("Counts");
inv_mass_discorde->SetFillColor(kOrange-3);

TH1D * inv_mass_concorde = (TH1D *)file->Get("InvMassConc");
inv_mass_concorde->GetXaxis()->SetTitle("Invariant Concord Mass [GeV/c^2]");
inv_mass_concorde->GetYaxis()->SetTitle("Counts");
inv_mass_concorde->SetFillColor(kOrange-3);

TH1D * inv_mass_pkd = (TH1D *)file->Get("InvMassPKDisc");
inv_mass_pkd->GetXaxis()->SetTitle("Invariant Discord Mass of P and K [GeV/c^2]");
inv_mass_pkd->GetYaxis()->SetTitle("Counts");
inv_mass_pkd->SetFillColor(kYellow-9);

TH1D * inv_mass_pkc = (TH1D *)file->Get("InvMassPKConc");
```

```
inv_mass_pkc->GetXaxis()->SetTitle("Invariant Concord Mass of P and K [GeV/c^2]");
inv_mass_pkc->GetYaxis()->SetTitle("Counts");
inv_mass_pkc->SetFillColor(kYellow-9);

TH1D * inv_mass_res = (TH1D *)file->Get("InvMassGen");
inv_mass_res->GetXaxis()->SetTitle("Invariant Mass of generated Particles [GeV/c^2]");
inv_mass_res->GetYaxis()->SetTitle("Counts");

cout <<endl;

double Pione_Positivo = type->GetBinContent(1);
double Pione_Positivo_Errore = type->GetBinError(1);
cout << "Pioni+ : " << Pione_Positivo << " +- " << Pione_Positivo_Errore
<< " generazioni -> " << (Pione_Positivo*100)/1e7 << "%" << '\n';
double Pione_Negativo = type->GetBinContent(2);
double Pione_Negativo_Errore = type->GetBinError(2);
cout << "Pioni- : " << Pione_Negativo << " +- " << Pione_Negativo_Errore
<< " generazioni -> " << (Pione_Negativo*100)/1e7 << "%" << '\n';
double Kaone_Positivo = type->GetBinContent(3);
double Kaone_Positivo_Errore = type->GetBinError(3);
cout << "Kaoni+ : " << Kaone_Positivo << " +- " << Kaone_Positivo_Errore
<< " generazioni -> " << (Kaone_Positivo*100)/1e7 << "%" << '\n';
double Kaone_Negativo = type->GetBinContent(4);
double Kaone_Negativo_Errore = type->GetBinError(4);
cout << "Kaoni- : " << Kaone_Negativo << " +- " << Kaone_Negativo_Errore
<< " generazioni -> " << (Kaone_Negativo*100)/1e7 << "%" << '\n';
double Protone_Positivo = type->GetBinContent(5);
double Protone_Positivo_Errore = type->GetBinError(5);
cout << "Protoni+ : " << Protone_Positivo << " +- " << Protone_Positivo_Errore
<< " generazioni -> " << (Protone_Positivo*100)/1e7 << "%" << '\n';
double Protone_Negativo = type->GetBinContent(6);
double Protone_Negativo_Errore = type->GetBinError(6);
cout << "Protoni- : " << Protone_Negativo << " +- " << Protone_Negativo_Errore
<< " generazioni -> " << (Protone_Negativo*100)/1e7 << "%" << '\n';
double Risonanza = type->GetBinContent(7);
double Risonanza_Errore = type->GetBinError(7);
cout << "K* : " << Risonanza << " +- " << Risonanza_Errore
<< " generazioni -> " << (Risonanza*100)/1e7 << "%" << '\n';

cout << endl;
double Somma, Errore_Somma;
Somma = Pione_Positivo + Pione_Negativo + Kaone_Positivo + Kaone_Negativo + Protone_Positivo + Protone_Negativo +
Risonanza;
Errore_Somma = Pione_Positivo_Errore + Pione_Negativo_Errore + Kaone_Positivo_Errore + Kaone_Negativo_Errore +
Protone_Positivo_Errore + Protone_Negativo_Errore + Risonanza_Errore;
cout << "Il numero di ingressi totale è il seguente: " << Somma << " +- " << Errore_Somma <<endl
<< "In percentuale: " << (Somma*100)/1e7 << "%" <<endl <<endl;

double *ptr = new double[2];
double *ptr2 = new double [3];

//CANVAS A.
TCanvas *CanvasA = new TCanvas ( "CanvasA", "Distribuzioni di abbondanza particelle, impulso, angolo polare e angolo
azimutale",
50, 10, 1200, 700);
CanvasA->Divide(2,2);
CanvasA->cd(1);
type->Draw();

CanvasA->cd(2);
impulse->Draw();
TF1 * fit_esponenziale = new TF1( "fit_esponenziale", "expo", 0, 4);
fit_esponenziale->SetParameters(1, 1);
impulse->Fit(fit_esponenziale, "R");
fit_esponenziale->GetParameters(&ptr[0]);
fit_esponenziale->SetParameters(ptr);
fit_esponenziale->Draw("same");
gStyle->SetOptFit(1111);

CanvasA->cd(3);
```

```
theta->Draw();
TF1 * gunza = new TF1 ("gunza", fi, 0, M_PI, 2);
gunza->SetParNames("q", "m");
gunza->SetParameters(10000, 0);
theta->Fit(gunza, "R");
gunza->GetParameters(&ptr[0]);
gunza->SetParameters(ptr);
gunza->Draw("same");
gStyle->SetOptFit(1111);

CanvasA->cd(4);
phi->Draw();
TF1 * funza = new TF1 ("funza", fi, 0, 2*M_PI, 2);
funza->SetParNames("q", "m");
funza->SetParameters(10000, 0);
phi->Fit(funza, "R");
funza->GetParameters(&ptr[0]);
funza->SetParameters(ptr);
funza->Draw("same");
gStyle->SetOptFit(1111);

//CANVAS B.
TCanvas *CanvasB = new TCanvas ( "CanvasB", "Distribuzione di massa invariante: particelle generate da k*, differenza di
combinazioni",
50, 10, 1200, 700);
CanvasB->Divide(2,2);

CanvasB->cd(1);
TH1F *h0 = new TH1F ("h0","Distribuzione Della Massa Invariante Delle Particelle Generate", 5000, 0, 2);
TF1 * dunza = new TF1 ("dunza", "gaus", 0, 2);
inv_mass_res->Fit(dunza, "R");
inv_mass_res->Draw();

CanvasB->cd(2);
TH1F *h1 = new TH1F ("h1","Differenza Tra Distribuzione Della Massa Invariante del Pione e del Kaone", 5000, 0, 5);
h1->Add(inv_mass_pkd, inv_mass_pkc,1, -1);
TF1 * hunza = new TF1 ("hunza", "gaus", 0.4, 0.9);
h1->Fit(hunza, "R");
h1->Draw();

CanvasB->cd(3);
TH1F *h2 = new TH1F ("h2","Differenza Tra Distribuzione Della Massa Invariante delle Particelle Generate", 5000, 0, 5);
h2->Add(inv_mass_discorde, inv_mass_concorde,1, -1);
TF1 * lunza = new TF1 ("lunza", "gaus", 0.6, 0.7);
h2->Fit(lunza, "R");
h2->Draw();
}
```