Simulazione di urti fra particelle elementari

Ahmed Jonayed, Brandi Giovanni, Campisi Dario Eugenio

Gennaio 2024

1 Introduzione

Il programma ha l'intento di simulare eventi fisici risultanti da **collisioni tra particelle elementari** ed eseguirne un'analisi statistica, sfruttando il *framework* di analisi dati ROOT.
Concretamente, il codice genera un numero pari a circa 10^7 particelle di 7 diversi tipi, secondo delle proporzioni definite, e associa ad esse un vettore quantità di moto (impulso) con modulo e direzione casuali. Al termine della generazione i dati di maggiore interesse vengono inseriti in degli istogrammi e successivamente analizzati.

2 Struttura del codice

Il programma è costituito da tre classi fondamentali, di seguito elencate.
`ParticleType`, che descrive il tipo di una particella, caratterizzato da nome, massa e carica. `ResonanceType`, derivata da ParticleType, che descrive le risonanze, differisce dalla prima per un parametro aggiuntivo: la larghezza di risonanza. Infine, `Particle` che descrive una singola particella, caratterizzata da un indice identificativo del tipo a cui appartiene, dalle componenti cartesiane della sua quantità di moto, e da un vettore statico (`fParticleTypes`) di puntatori a `ParticleType`, che assolve alla funzione di *dizionario* dei tipi di particella definiti nel codice.

Le classi `ParticleType` e `ResonanceType` presentano solo funzioni di tipo *getter* per i propri attributi e una funzione `Print()` che restituisce in standard output tutte le proprietà, mentre la classe `Particle`, più complessa, ha:

- *Getter* pubblici per tutti i suoi attributi
- *Getter* pubblici che ritornano gli attributi del ParticleType a cui la particella appartiene
- Metodi pubblici che ritornano: l'impulso trasverso, l'energia totale e la massa invariante
- *Setter* pubblici per la quantità di moto e per l'indice, quest'ultimo con due overload, in caso non sia conosciuto l'indice da assegnare ma solo il nome del tipo di particella, o viceversa
- Metodi statici per aggiungere tipi di particelle a `fParticleTypes`, e per restituire in standard output un elenco di quelli già presenti, con i relativi valori degli attributi
- Un metodo privato `FindParticleType()` che, dato il nome di un `ParticleType`, ne restituisce il corrispondente indice in `fParticleTypes`, oppure -1 se non è presente
- Un metodo pubblico `Decay2Body()` che simula la cinematica di un decadimento, e un metodo privato `Boost()` utilizzato in `Decay2Body()`

3 Generazione

Il programma produce un numero pari a 10^5 eventi, in ognuno dei quali vengono generate circa 100 particelle che si dividono in:

- Pioni± 80% (40% +, 40% -), massa $m_\pi = 0.13957\ GeV/c^2$
- Kaoni± 10% (5% +, 5% -), massa $m_K = 0.49367\ GeV/c^2$

- Protoni± 9% (4.5% +, 4.5% -), massa $m_p = 0.93827\ GeV/c^2$
- Risonanze (K*) 1%, carica nulla, massa $m_p = 0.89166\ GeV/c^2$ e larghezza di risonanza $\Gamma_{K*} = 0.050\ GeV/c^2$

Le risonanze sono particelle instabili che, quando generate, decadono in pioni e kaoni (50% pione+ e kaone-, 50% pione- e kaone+), è questa la giustificazione del "circa 100 particelle".
Ad ogni particella generata viene associato un vettore quantità di moto, il cui modulo è estratto da una popolazione esponenziale decrescente con media 1, e le cui componenti lungo i tre assi cartesiani sono determinate a partire da un angolo polare θ, estratto da una distribuzione uniforme in $[0, \pi]$ e da un angolo azimutale ϕ estratto parimenti da una distribuzione uniforme, in $[0, 2\pi]$, utilizzando le note conversioni da coordinate polari a cartesiane:

$$\begin{cases} p_x = p \sin\theta \cos\phi \\ p_y = p \sin\theta \sin\phi \\ p_z = p \cos\theta \end{cases} \tag{1}$$

Con i dati frutto della generazione vengono riempiti i seguenti istogrammi:

1. Tipi di particelle generate
2. Distribuzioni degli angoli polare e azimutale
3. Distribuzioni dell'impulso, dell'impulso trasverso e dell'energia totale delle particelle
4. Massa invariante (MI) fra tutte le particelle
5. MI fra particelle di segno opposto
6. MI fra particelle di segno concorde
7. MI fra pioni e kaoni di segno opposto
8. MI fra pioni e kaoni di segno concorde
9. MI fra le particelle di decadimento

4 Analisi

Dalla **Tabella 1** è possibile osservare che le particelle vengono correttamente generate secondo le proporzioni definite.

Specie	Occorrenze osservate	occorrenze attese
$\pi+$	4.04967×10^6	4.07992×10^6
$\pi-$	4.04871×10^6	4.07992×10^6
K+	551087	509990
K-	549676	509990
p+	499983	458991
p-	400760	458991
K*	99886	101998

Tabella 1: Abbondanza delle particelle

Per quanto concerne le distribuzioni osservate degli angoli polare e azimutale e del modulo della quantità di moto, uniformi ed esponenziale rispettivamente, di ognuna di esse è stato eseuito il relativo fit per verificarne la concordanza con i valori dati in input in fase di generazione. La **Tabella 2** riassume gli esiti dei suddetti. Come si osserva, il fit uniforme operato sulle distribuzioni di θ e ϕ ha esito positivo. Al contrario, la distribuzione dell'impulso restituisce un χ^2 ridotto piuttosto alto, esito all'apparenza errato ma corretto in realtà. La ragione di ciò è che il riempimento di tutti gli istogrammi relativi alle informazioni delle particelle (e.g. tipo della particella, impulso, energia totale) avviene dopo che la generazione di **tutte** le particelle è terminata, tenendo quindi conto anche delle particelle generate dai decadimenti delle risonanze (pari, in media, a circa il 2% delle particelle totali). L'impulso di queste ultime non è estratto da una popolazione esponenziale, ma è il risultato delle regole implementate nel metodo `Decay2Body()`, pertanto, correttamente, il fit esponenziale fallisce. È stato verificato che, non tenendo conto delle particelle di decadimento, il fit ha invece esito positivo.

Distribuzione	Parametri del Fit	χ^2/DOF
Angolo Polare (pol0)	55554 +/- 17	1.02491
Angolo Azimutale (pol0)	27777 +/- 9	1.0178
Modulo dell'impulso (expo)	-1.0075 +/- 0.0003	6.03118

Tabella 2: Angoli polari e azimutali, modulo dell'impulso

Dai dati è possibile estrarre il valore della massa e della larghezza della risonanza, così da confrontarli con i valori inseriti in input. Per fare ciò è necessario sottrarre all'istogramma 5 l'istogramma 6, equivalentemente, si può sottrarre all'istogramma 7 l'istogramma 8. Degli istogrammmi risultanti dalla sottrazione bisogna fare un fit gaussiano, la media del fit coinciderà, entro gli errori statistici, con la massa della risonanza, mentre la deviazione standard con la sua larghezza. In **Tabella 3** sono riportati gli esiti di tali operazioni.

Distribuzione e Fit	Media	Sigma	Ampiezza	χ^2/DOF
Massa invariante vere K* (fit gauss)	0.89153 +/- 0.00015	0.04985 +/- 0.00011	3193 +/- 12	1.04071
Massa invariante ottenuta da differenza delle combinazioni di carica discorde e concore (fit gauss)	0.895 +/- 0.006	0.047 +/- 0.008	59389 ± 7756	1.22989
Massa invariante ottenuta da differenza delle combinazioni πK di carica discorde e concorde (fit gauss)	0.892 +/- 0.003	0.047 ± 0.005	58633 ± 46752	0.848344

Tabella 3: Analisi della K*

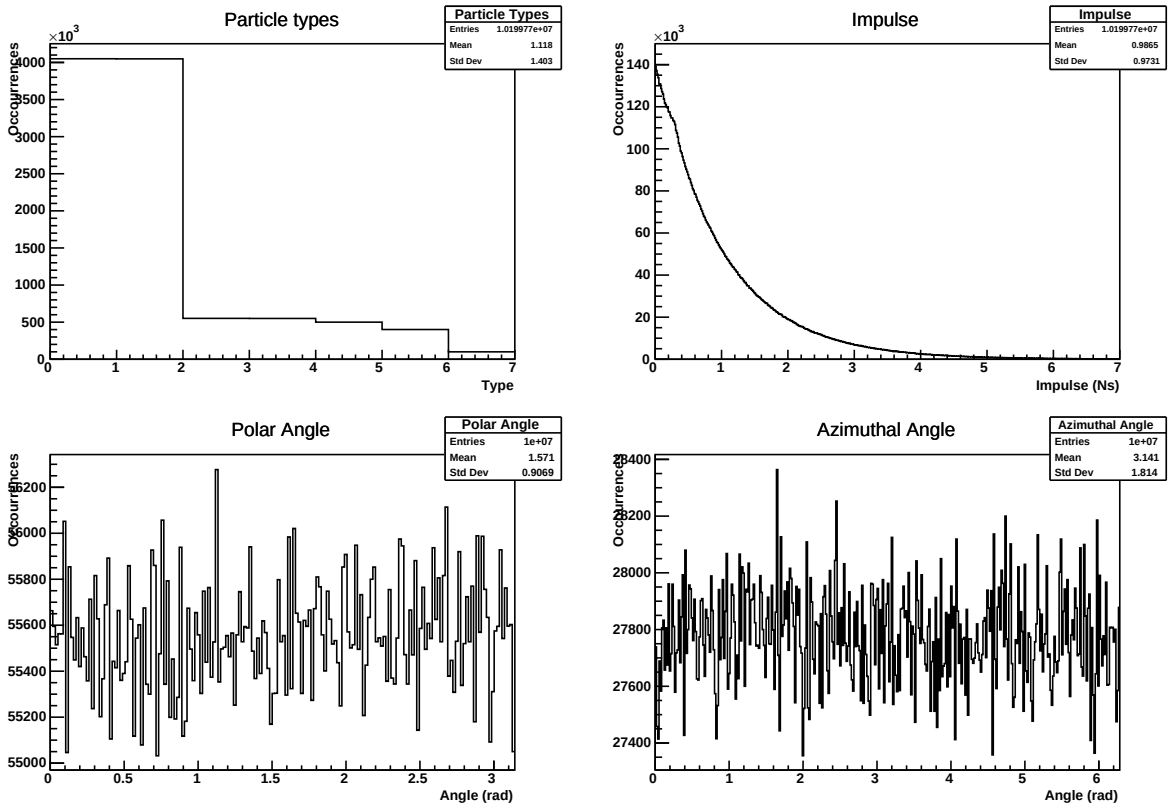

Figura 1: Distribuzioni di abbondanza di particelle, dell'impulso, angolo polare, angolo azimutale

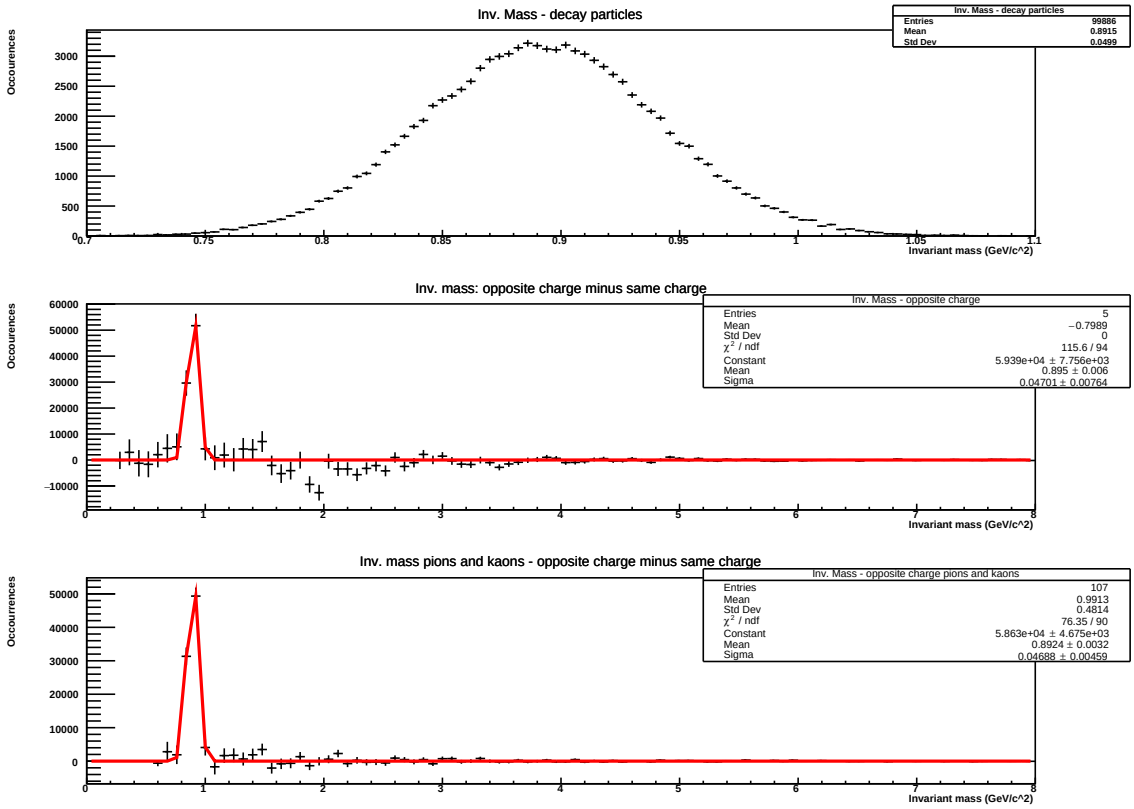

Figura 2: Massa invariante delle K+ vere, massa invariante ottenuta dalla differenza fra combinazioni di carica discorde e concorde, massa invariante ottenuta dalla differenza fra combinazioni πK di carica discorde e concorde

5 Appendice

Presentiamo di seguito il listato del codice, diviso negli 8 file che lo compongono.

5.1 particle_type.hpp

```
#ifndef PARTICLE_TYPE_HPP
#define PARTICLE_TYPE_HPP

class ParticleType {
 public:
  ParticleType(const char *name, const double mass, const int charge)
      : fName{name}, fMass{mass}, fCharge{charge} {}

  const char *GetName() const;
  double GetMass() const;
  int GetCharge() const;

  virtual double GetWidth() const { return 0; }

  virtual void Print() const;

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

#endif
```

5.2 particle_type.cpp

```
#include "particle_type.hpp"

#include <iostream>

const char *ParticleType::GetName() const { return fName; }

double ParticleType::GetMass() const { return fMass; }

int ParticleType::GetCharge() const { return fCharge; }

void ParticleType::Print() const {
  std::cout << "Name:" << fName << "\nMass:" << fMass
            << "\nCharge:" << fCharge << '\n';
}
```

5.3 resonance_type.hpp

```
#ifndef RESONANCE_TYPE_HPP
#define RESONANCE_TYPE_HPP

#include "particle_type.hpp"

class ResonanceType : public ParticleType {
 public:
  ResonanceType(const char *name, const double mass, const int charge,
```

```
                const double width)
      : ParticleType(name, mass, charge), fWidth{width} {}

  double GetWidth() const;

  void Print() const;

 private:
  const double fWidth;
};

#endif
```

5.4 resonance_type.cpp

```
#include "resonance_type.hpp"

#include <iostream>

double ResonanceType::GetWidth() const { return fWidth; }

void ResonanceType::Print() const {
  ParticleType::Print();
  std::cout << "Width:" << fWidth << '\n';
}
```

5.5 particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include <vector>

#include "particle_type.hpp"
#include "resonance_type.hpp"

class Particle {
 public:
  Particle() {}
  Particle(const char *name, double px = 0, double py = 0, double pz = 0);

  int GetIndex() const;
  int GetCharge() const;
  double GetPx() const;
  double GetPy() const;
  double GetPz() const;
  double GetP() const;
  double GetTrsP() const;
  double GetMass() const;

  void SetIndex(const int index);
  void SetIndex(const char *name);
  void SetP(double px, double py, double pz);

  static void AddParticleType(const char *name, const double mass,
                              const int charge, const double width = 0);
```

```
  static void PrintParticleTypes();
  void PrintParticleInfo() const;

  double TotalEnergy() const;
  double InvariantMass(const Particle &other) const;

  int Decay2body(Particle &dau1, Particle &dau2) const;

 private:
  static std::vector<ParticleType *> fParticleTypes;
  int fIndex;

  double fPx;
  double fPy;
  double fPz;

  int FindParticleType(const char *name) const;
  void Boost(double bx, double by, double bz);
};

#endif
```

5.6 particle.cpp

```
#include "particle.hpp"

#include <cmath>     // for M_PI
#include <cstdlib>   // for RAND_MAX
#include <iostream>

std::vector<ParticleType *> Particle::fParticleTypes;

Particle::Particle(const char *name, double px, double py, double pz)
    : fPx{px}, fPy{py}, fPz{pz} {
  if (FindParticleType(name) == -1) {
    throw std::invalid_argument("particle type not found\n");
  } else {
    fIndex = FindParticleType(name);
  }
}

int Particle::FindParticleType(const char *name) const {
  for (auto it = fParticleTypes.begin(); it != fParticleTypes.end(); ++it) {
    if ((*it)->GetName() == name) {
      return (int)(it - fParticleTypes.begin());
    }
  }
  return -1;
}

int Particle::GetIndex() const { return fIndex; }
int Particle::GetCharge() const { return fParticleTypes[fIndex]->GetCharge(); }
double Particle::GetPx() const { return fPx; }
double Particle::GetPy() const { return fPy; }
double Particle::GetPz() const { return fPz; }
double Particle::GetP() const {
  return std::sqrt(fPx * fPx + fPy * fPy + fPz * fPz);
}
```

```
double Particle::GetTrsP() const { return std::sqrt(fPx * fPx + fPy * fPy); }
double Particle::GetMass() const { return fParticleTypes[fIndex]->GetMass(); }

void Particle::SetIndex(const int index) {
  if (index >= 0 && index < (int)(fParticleTypes.size())) {
    fIndex = index;
  } else {
    std::cerr << "Index not found\n";
  }
}
void Particle::SetIndex(const char *name) {
  fIndex = FindParticleType(name);

  if (fIndex == -1) {
    std::cerr << "Particle name not found\n";
  }
}

void Particle::SetP(double px, double py, double pz) {
  fPx = px;
  fPy = py;
  fPz = pz;
}

void Particle::AddParticleType(const char *name, const double mass,
                               const int charge, const double width) {
  ParticleType *type = new ResonanceType(name, mass, charge, width);
  fParticleTypes.push_back(type);
}

void Particle::PrintParticleTypes() {
  for (auto type : fParticleTypes) {
    type->Print();
  }
}

void Particle::PrintParticleInfo() const {
  std::cout << fIndex << " " << fParticleTypes[fIndex]->GetName()
            << " Impulse (Px,Py,Pz): (" << fPx << "," << fPy << "," << fPz
            << ")\n";
}

double Particle::TotalEnergy() const {
  return std::sqrt((this->GetMass() * this->GetMass()) +
                   (fPx * fPx + fPy * fPy + fPz * fPz));
}

double Particle::InvariantMass(const Particle &other) const {
  return std::sqrt((this->TotalEnergy() + other.TotalEnergy()) *
                       (this->TotalEnergy() + other.TotalEnergy()) -
                   ((fPx + other.GetPx()) * (fPx + other.GetPx()) +
                    (fPy + other.GetPy()) * (fPy + other.GetPy()) +
                    (fPz + other.GetPz()) * (fPz + other.GetPz())));
}

int Particle::Decay2body(Particle &dau1, Particle &dau2) const {
  if (this->GetMass() == 0.0) {
    printf("Decayment cannot be preformed if mass is zero\n");
```

```
  return 1;
}

double massMot = this->GetMass();
double massDau1 = dau1.GetMass();
double massDau2 = dau2.GetMass();

if (fIndex > -1) {  // add width effect

  // gaussian random numbers

  float x1, x2, w, y1;

  double invnum = 1. / RAND_MAX;
  do {
    x1 = 2.0 * std::rand() * invnum - 1.0;
    x2 = 2.0 * std::rand() * invnum - 1.0;
    w = x1 * x1 + x2 * x2;
  } while (w >= 1.0);

  w = std::sqrt((-2.0 * log(w)) / w);
  y1 = x1 * w;

  massMot += fParticleTypes[fIndex]->GetWidth() * y1;
}

if (massMot < massDau1 + massDau2) {
  printf(
      "Decayment cannot be preformed because mass is too low in this "
      "channel\n");
  return 2;
}

double pout =
    std::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 = std::rand() * norm;
double theta = std::rand() * norm * 0.5 - M_PI / 2.;
dau1.SetP(pout * std::sin(theta) * std::cos(phi),
          pout * std::sin(theta) * std::sin(phi), pout * std::cos(theta));
dau2.SetP(-pout * std::sin(theta) * std::cos(phi),
          -pout * std::sin(theta) * std::sin(phi), -pout * std::cos(theta));

double energy =
    std::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::Boost(double bx, double by, double bz) {
  double energy = this->TotalEnergy();

  // Boost this Lorentz vector
  double b2 = bx * bx + by * by + bz * bz;
  double gamma = 1.0 / std::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;
}
```

5.7 Main.C

```
#include <iostream>

#include "TCanvas.h"
#include "TFile.h"
#include "TH1.h"
#include "TROOT.h"
#include "TRandom.h"
#include "TStyle.h"
#include "particle.hpp"
#include "particle_type.hpp"
#include "resonance_type.hpp"

void SetStyle() {
  gROOT->SetStyle("Plain");
  gStyle->SetOptStat(1111);
  gStyle->SetOptFit(111);
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(0);
}

int Main() {
  SetStyle();
  gRandom->SetSeed(314234);

  Particle::AddParticleType("pione+", 0.13957, 1);      // pione+
  Particle::AddParticleType("pione-", 0.13957, -1);     // pione-
  Particle::AddParticleType("Kaone+", 0.49367, 1);      // Kaone+
  Particle::AddParticleType("Kaone-", 0.49367, -1);     // Kaone-
  Particle::AddParticleType("protone+", 0.93827, 1);    // protone+
  Particle::AddParticleType("protone-", 0.93827, -1);   // protone-
  Particle::AddParticleType("K*", 0.89166, 0, 0.050);   // risonanza K*

  std::vector<Particle> eventParticles{};
  for (int i = 0; i < 100; ++i) {
    try {
      Particle p;
      eventParticles.push_back(p);
    } catch (const std::exception &e) {
      std::cerr << "Error:-" << e.what();
```

```
  }
}

std::vector<Particle> decayParticles{};

TH1F *hParticleTypes =
    new TH1F("Particle Types", "Particle Types", 7, 0., 7.);
TH1F *hPolar = new TH1F("Polar Angle", "Polar Angle", 180, 0., M_PI);
TH1F *hAzimuthal =
    new TH1F("Azimuthal Angle", "Azimuthal Angle", 360, 0., 2 * M_PI);
TH1F *hImpulse = new TH1F("Impulse", "Impulse", 500, 0., 7.);
TH1F *hTrsImpulse =
    new TH1F("Trasverse Impulse", "Trasverse Impulse", 500, 0., 7.);
TH1F *hEnergy = new TH1F("Energy", "Energy", 100, -10, 10);

TH1F *hIMAll = new TH1F("Invariant Mass", "Invariant Mass", 100, 0., 8.);
hIMAll->Sumw2();

TH1F *hIMOppositeCharge =
    new TH1F("Inv. Mass - opposite charge", "Inv. Mass - opposite charge",
             100, 0., 8.);
hIMOppositeCharge->Sumw2();

TH1F *hIMSameCharge = new TH1F("Inv. Mass - same charge",
                               "Inv. Mass - same charge", 100, 0., 8.);
hIMSameCharge->Sumw2();

TH1F *hIMOppositePK =
    new TH1F("Inv. Mass - opposite charge pions and kaons",
             "Inv. Mass - opposite charge pions and kaons", 100, 0., 8.);
hIMOppositePK->Sumw2();

TH1F *hIMSamePK =
    new TH1F("Inv. Mass - same charge pions and kaons",
             "Inv. Mass - same charge pions and kaons", 100, 0., 8.);
hIMSamePK->Sumw2();

TH1F *hIMDecayParticles =
    new TH1F("Inv. Mass - decay particles", "Inv. Mass - decay particles",
             100, 0.7, 1.1);
hIMDecayParticles->Sumw2();

// inizio dei 1e5 eventi
for (int n = 0; n < 1e5; ++n) {
  for (int i = 0; i < 100; ++i) {
    double theta = gRandom->Uniform(0, M_PI);
    hPolar->Fill(theta);

    double phi = gRandom->Uniform(0, 2 * M_PI);
    hAzimuthal->Fill(phi);

    double p = gRandom->Exp(1.);
    eventParticles[i].SetP(p * std::sin(theta) * std::cos(phi),
                           p * std::sin(theta) * std::sin(phi),
                           p * std::cos(theta));

    double random = gRandom->Rndm();
    if (random < 0.8) {
```

```
      if (random < 0.4) {
        eventParticles[i].SetIndex("pione+");
      } else {
        eventParticles[i].SetIndex("pione-");
      }
    } else if (random < 0.9) {
      if (random < 0.85) {
        eventParticles[i].SetIndex("Kaone+");
      } else {
        eventParticles[i].SetIndex("Kaone-");
      }
    } else if (random < 0.99) {
      if (random < 0.95) {
        eventParticles[i].SetIndex("protone+");
      } else {
        eventParticles[i].SetIndex("protone-");
      }
    } else {
      eventParticles[i].SetIndex("K*");

      try {
        Particle p1{};
        Particle p2{};

        if (std::rand() % 2) {
          p1.SetIndex("pione+");
          p2.SetIndex("Kaone-");
        } else {
          p1.SetIndex("pione-");
          p2.SetIndex("Kaone+");
        }

        eventParticles[i].Decay2body(p1, p2);

        decayParticles.push_back(p1);
        decayParticles.push_back(p2);
      } catch (const std::exception &e) {
        std::cerr << "Error:-" << e.what();
      }
    }
  } // fine della generazione delle ~100 particelle

  eventParticles.insert(eventParticles.end(), decayParticles.begin(),
                        decayParticles.end());

  // riempimento istogrammi
  for (unsigned long i = 0; i < eventParticles.size(); ++i) {
    hParticleTypes->Fill(eventParticles[i].GetIndex());
    hImpulse->Fill(eventParticles[i].GetP());
    hTrsImpulse->Fill(eventParticles[i].GetTrsP());
    hEnergy->Fill(eventParticles[i].TotalEnergy());

    // istogrammi massa invariante
    for (unsigned long j = i + 1; j < eventParticles.size(); ++j) {
      // tutte le particelle
      hIMAll->Fill(eventParticles[i].InvariantMass(eventParticles[j]));

      // segno discorde
```

```
      if (eventParticles[i].GetCharge() * eventParticles[j].GetCharge() ==
          -1) {
        hIMOppositeCharge->Fill(
            eventParticles[i].InvariantMass(eventParticles[j]));

        // pioni e kaoni discordi
        if ((eventParticles[i].GetMass() + eventParticles[j].GetMass()) ==
            0.63324) {
          hIMOppositePK->Fill(
              eventParticles[i].InvariantMass(eventParticles[j]));
        }
      }
      // segno concorde
      else if (eventParticles[i].GetCharge() *
                   eventParticles[j].GetCharge() ==
               1) {
        hIMSameCharge->Fill(
            eventParticles[i].InvariantMass(eventParticles[j]));

        // pioni e kaoni concordi
        if ((eventParticles[i].GetMass() + eventParticles[j].GetMass()) ==
            0.63324) {
          hIMSamePK->Fill(eventParticles[i].InvariantMass(eventParticles[j]));
        }
      }
    }
  }

  // particelle di decadimento
  for (unsigned long i = 0; i < decayParticles.size(); i += 2) {
    hIMDecayParticles->Fill(
        decayParticles[i].InvariantMass(decayParticles[i + 1]));
  }

  eventParticles.erase(eventParticles.end() - decayParticles.size(),
                       eventParticles.end());
  decayParticles.clear();

}  // fine dei 1e5 eventi

// inserimento degli istogrammi in un file ROOT
TFile *file = new TFile("histograms.root", "RECREATE");
TCanvas *canvas = new TCanvas("canvas");

hParticleTypes->Draw();
canvas->Print("ParticleTypes.pdf");
canvas->Print("ParticleTypes.C");
canvas->Clear();
hParticleTypes->Write();

hPolar->Draw();
canvas->Print("PolarAngle.pdf");
canvas->Print("PolarAngle.C");
canvas->Clear();
hPolar->Write();

hAzimuthal->Draw();
canvas->Print("AzimuthalAngle.pdf");
```

```
canvas->Print("AzimuthalAngle.C");
canvas->Clear();
hAzimuthal->Write();

hImpulse->Draw();
canvas->Print("Impulse.pdf");
canvas->Print("Impulse.C");
canvas->Clear();
hImpulse->Write();

hTrsImpulse->Draw();
canvas->Print("TrasverseImpulse.pdf");
canvas->Print("TrasverseImpulse.C");
canvas->Clear();
hTrsImpulse->Write();

hEnergy->Draw();
canvas->Print("Energy.pdf");
canvas->Print("Energy.C");
canvas->Clear();
hEnergy->Write();

hIMAll->Draw();
canvas->Print("InvMass.pdf");
canvas->Print("InvMass.C");
canvas->Clear();
hIMAll->Write();

hIMOppositeCharge->Draw();
canvas->Print("InvMassOppositeCharge.pdf");
canvas->Print("InvMassOppositeCharge.C");
canvas->Clear();
hIMOppositeCharge->Write();

hIMSameCharge->Draw();
canvas->Print("InvMassSameCharge.pdf");
canvas->Print("InvMassSameCharge.C");
canvas->Clear();
hIMSameCharge->Write();

hIMOppositePK->Draw();
canvas->Print("InvMassOppChargePionsKaons.pdf");
canvas->Print("InvMassOppChargePionsKaons.C");
canvas->Clear();
hIMOppositePK->Write();

hIMSamePK->Draw();
canvas->Print("InvMassSameChargePionsKaons.pdf");
canvas->Print("InvMassSameChargePionsKaons.C");
canvas->Clear();
hIMSamePK->Write();

hIMDecayParticles->Draw();
canvas->Print("InvMassDecayParticles.pdf");
canvas->Print("InvMassDecayParticles.C");
canvas->Clear();
hIMDecayParticles->Write();
```

```
  canvas->Close();
  file->Close();

  return 0;
}
```

5.8 histo_analysis.cpp

```
#include "TCanvas.h"
#include "TF1.h"
#include "TFile.h"
#include "TH1F.h"
#include "TRandom.h"
#include "TStyle.h"
#include "particle.hpp"
#include "particle_type.hpp"
#include "resonance_type.hpp"

void HistoAnalysis()
{

        TFile *file = new TFile("histograms.root");

        TH1F *hParticleTypes = (TH1F *)file->Get("Particle Types");
        TH1F *hPolar = (TH1F *)file->Get("Polar Angle");
        TH1F *hAzimuthal = (TH1F *)file->Get("Azimuthal Angle");
        TH1F *hImpulse = (TH1F *)file->Get("Impulse");
        TH1F *hTrsImpulse = (TH1F *)file->Get("Trasverse Impulse");
        TH1F *hEnergy = (TH1F *)file->Get("Energy");
        TH1F *hIMAll = (TH1F *)file->Get("Invariant Mass");
        TH1F *hIMOppositeCharge = (TH1F *)file->Get("Inv. Mass - opposite charge");
        TH1F *hIMSameCharge = (TH1F *)file->Get("Inv. Mass - same charge");
        TH1F *hIMOppositePK =
            (TH1F *)file->Get("Inv. Mass - opposite charge pions and kaons");
        TH1F *hIMSamePK =
            (TH1F *)file->Get("Inv. Mass - same charge pions and kaons");
        TH1F *hIMDecayParticles = (TH1F *)file->Get("Inv. Mass - decay particles");

        std::vector<TH1F *> histoVector{
            hParticleTypes, hPolar, hAzimuthal, hImpulse,
            hTrsImpulse, hEnergy, hIMAll, hIMOppositeCharge,
            hIMSameCharge, hIMOppositePK, hIMSamePK, hIMDecayParticles};

        // NUMERO DI INGRESSI PER OGNI ISTOGRAMMA
        std::cout << "\n\n\n**** ENTRIES ****\n";
        for (auto histo : histoVector)
        {
                std::cout << "\n " << histo->GetTitle() << ": " << histo->GetEntries();
        }

        // DISTRIBUZIONE DEI TIPI DI PARTICELLE
        std::cout << "\n\n**** PARTICLE TYPES ****\n\n";

        std::cout << " Pions (+): " << hParticleTypes->GetBinContent(1) << " i.e. "
                  << 100 * hParticleTypes->GetBinContent(1) /
                         hParticleTypes->GetEntries()
                  << "%\n";
        std::cout << " Pions (-): " << hParticleTypes->GetBinContent(2) << " i.e. "
```

```
          << 100 * hParticleTypes->GetBinContent(2) /
                hParticleTypes->GetEntries()
          << "%\n";
std::cout << " Kaons (+): " << hParticleTypes->GetBinContent(3) << " i.e. "
          << 100 * hParticleTypes->GetBinContent(3) /
                hParticleTypes->GetEntries()
          << "%\n";
std::cout << " Kaons (-): " << hParticleTypes->GetBinContent(4) << " i.e. "
          << 100 * hParticleTypes->GetBinContent(4) /
                hParticleTypes->GetEntries()
          << "%\n";
std::cout << " Protons (+): " << hParticleTypes->GetBinContent(5) << " i.e. "
          << 100 * hParticleTypes->GetBinContent(5) /
                hParticleTypes->GetEntries()
          << "%\n";
std::cout << " Protons (-): " << hParticleTypes->GetBinContent(6) << " i.e. "
          << 100 * hParticleTypes->GetBinContent(6) /
                hParticleTypes->GetEntries()
          << "%\n";
std::cout << " Resonances: " << hParticleTypes->GetBinContent(7) << " i.e. "
          << 100 * hParticleTypes->GetBinContent(7) /
                hParticleTypes->GetEntries()
          << "%\n";

// FITTING
// angolo polare
TF1 *polarFit = new TF1("polar fit function", "[0]", 0., M_PI);
hPolar->Fit(polarFit, "Q0");

// angolo azimutale
TF1 *azimuthalFit = new TF1("azimuthal fit function", "[0]", 0., 2 * M_PI);
hAzimuthal->Fit(azimuthalFit, "Q0");

// impulso
TF1 *impulseFit = new TF1("impulse fit function", "expo", 0., 7.);
hImpulse->Fit(impulseFit, "Q0");

std::cout << "\n**** FITTING ****\n\n";

std::cout << " Polar Angle\n\n"
          << "  Parameter: " << polarFit->GetParameter(0) << " +/- "
          << polarFit->GetParError(0) << "\n  Reduced Chi Square: "
          << polarFit->GetChisquare() / polarFit->GetNDF()
          << "\n  Probability: " << polarFit->GetProb();

std::cout << "\n\n Azimuthal Angle\n\n"
          << "  Parameter: " << azimuthalFit->GetParameter(0) << " +/- "
          << azimuthalFit->GetParError(0) << "\n  Reduced Chi Square: "
          << azimuthalFit->GetChisquare() / azimuthalFit->GetNDF()
          << "\n  Probability: " << azimuthalFit->GetProb();

std::cout << "\n\n Impulse\n\n"
          << "  Width: " << impulseFit->GetParameter(0) << " +/- "
          << impulseFit->GetParError(0)
          << "\n  Mean: " << impulseFit->GetParameter(1) << " +/- "
          << impulseFit->GetParError(1) << "\n  Reduced Chi Square: "
          << impulseFit->GetChisquare() / impulseFit->GetNDF()
          << "\n  Probability: " << impulseFit->GetProb();
```

```
// SOTTRAZIONE ISTOGRAMMI

TF1 FitIMDP = new TF1("Fit Massa Invariante K", "gaus", 0., 8.);
hIMDecayParticles->Fit(FitIMDP, "Q0");

TH1F *hSubtraction1 = new TH1F(*hIMOppositeCharge);
hSubtraction1->Add(hIMOppositeCharge, hIMSameCharge, 1., -1.);
TF1 *sub1Fit = new TF1("first subtraction fit", "gaus", 0., 8.);
hSubtraction1->Fit(sub1Fit, "Q0");

TH1F *hSubtraction2 = new TH1F(*hIMOppositePK);
hSubtraction2->Add(hIMOppositePK, hIMSamePK, 1., -1.);
TF1 *sub2Fit = new TF1("second subtraction fit", "gaus", 0., 8.);
hSubtraction2->Fit(sub2Fit, "Q0");

TCanvas *canvas1 = new TCanvas("canvas1");
gStyle->SetOptFit(111);
canvas1->Divide(1, 3);
canvas1->cd(1);

hIMDecayParticles->SetTitle("Inv. Mass - decay particles;
 Invariant mass (GeV/c^2); Occourences");
hIMDecayParticles->Draw();
canvas1->cd(2);
hSubtraction1->SetTitle("Inv. mass: opposite charge minus same charge;
 Invariant mass (GeV/c^2); Occourences");
hSubtraction1->Draw();
sub1Fit->Draw("same");

canvas1->cd(3);
hSubtraction2->SetTitle("Inv. mass pions and kaons - opposite charge minus
 same charge; Invariant mass (GeV/c^2); Occourrences");
hSubtraction2->Draw();
sub2Fit->Draw("same");

canvas1->Print("Subtractions.pdf");
canvas1->Print("Subtractions.C");

TCanvas *canvas2 = new TCanvas("canvas2");
canvas2->Divide(2, 2);
canvas2->cd(1);
hParticleTypes->SetTitle("Particle types; Type; Occourrences");
hParticleTypes->Draw();
canvas2->cd(2);
hImpulse->SetTitle("Impulse; Impulse (Ns); Occourrences");
hImpulse->Draw();
canvas2->cd(3);
hPolar->SetTitle("Polar Angle; Angle (rad); Occourrences");
hPolar->Draw();
canvas2->cd(4);
hAzimuthal->SetTitle("Azimuthal Angle; Angle (rad); Occourrences");
hAzimuthal->Draw();
canvas2->Print("Canvas2.pdf");
canvas2->Print("Canvas2.C");

std::cout << "\n\n**** OPERATIONS ON HISTOGRAMS ****\n\n";
```

```
        std::cout << " Inv. Mass Decay Particles \n\n"
                  << "  Width: " << FitIMDP->GetParameter(0) << " +/- "
                  << FitIMDP->GetParError(0)
                  << "\n  Mean (i.e. resonance mass): " << FitIMDP->GetParameter(1)
                  << " +/- " << FitIMDP->GetParError(1)
                  << "\n  Std Dev (i.e. resonance width): "
                  << FitIMDP->GetParameter(2) << " +/- " << FitIMDP->GetParError(2)
                  << "\n  Reduced Chi Square: "
                  << FitIMDP->GetChisquare() / FitIMDP->GetNDF()
                  << "\n  Fit Probability: " << FitIMDP->GetProb();

        std::cout << "\n\n Inv. Mass - opposite charge minus Inv. Mass -
                        same charge\n\n"
                  << "  Width: " << sub1Fit->GetParameter(0) << " +/- "
                  << sub1Fit->GetParError(0)
                  << "\n  Mean (i.e. resonance mass): " << sub1Fit->GetParameter(1)
                  << " +/- " << sub1Fit->GetParError(1)
                  << "\n  Std Dev (i.e. resonance width): "
                  << sub1Fit->GetParameter(2) << " +/- " << sub1Fit->GetParError(2)
                  << "\n  Reduced Chi Square: "
                  << sub1Fit->GetChisquare() / sub1Fit->GetNDF()
                  << "\n  Fit Probability: " << sub1Fit->GetProb();

        std::cout << "\n\n Inv. Mass - opposite charge pions and kaons minus Inv. "
                     "Mass - same charge pions and kaons\n\n"
                  << "  Width: " << sub2Fit->GetParameter(0) << " +/- "
                  << sub2Fit->GetParError(0)
                  << "\n  Mean (i.e. resonance mass): " << sub2Fit->GetParameter(1)
                  << " +/- " << sub2Fit->GetParError(1)
                  << "\n  Std Dev (i.e. resonance width): "
                  << sub2Fit->GetParameter(2) << " +/- " << sub2Fit->GetParError(2)
                  << "\n  Reduced Chi Square: "
                  << sub2Fit->GetChisquare() / sub2Fit->GetNDF()
                  << "\n  Fit Probability: " << sub2Fit->GetProb() << "\n\n\n";
}
```