Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Fisica

Laboratorio di Elettromagnetismo e Ottica
Prova di programmazione C++/ROOT
Relazione

Giuseppe Sguera

Sommario

Argomento di questa prova è stata l'implementazione di codice C++ atto a simulare e analizzare eventi fisici derivanti da collisioni di particelle elementari.

In particolare, sono stati simulati eventi di collisione a seguito dei quali viene prodotto il kaone neutro K*, una particella molto elusi-

va e di difficile individuazione, a causa dei suoi brevissimi tempi di decadimento [1].

Il corpo centrale del programma si occupa della simulazione degli eventi, effettuata mediante metodi di generazione Monte Carlo, servendosi di classi e funzioni appositamente definiti per descrivere il comportamento e le caratteristiche delle particelle generate. La grande mole di dati così prodotta verrà poi analizzata in una parte di codice separata, in cui viene fatto ampio uso del pacchetto software ROOT.

Lo scopo dell'analisi stratistica dei dati è quello di misurare indirettamente massa e lunghezza di risonanza del famigerato kaone K*, tramite l'isolamento dei suoi prodotti di decadimento dal "rumore di fondo" (tutte le altre particelle generate dall'urto) e la misura delle loro masse invarianti.

Indice

[1] si veda http://w3.lnf.infn.it/vi-presentiamo-il-kaone/

1 Struttura del codice

Il programma è scritto in C++ e fa uso delle librerie del pacchetto ROOT.

Il codice è diviso in file `.hpp` e `.cpp`. Le classi e i loro metodi sono dichiarate nei file header e definite nel codice sorgente (una coppia header/sorgente per ogni classe); i file denominati `main.cpp` contenengono il codice della generazione e dell'analisi.

1.1 Classi e metodi

Sono state implementate tre classi. Due per indicare le proprietà di base delle particelle (*ParticleType* e *ResonanceType*) e una (*Particle*) per descriverne il comportamento dopo ogni collisione.

ParticleType

La classe *ParticleType* rappresenta una particella stabile.

Le tre caratteristiche che la delineano (nome, massa e carica elettrica) sono incluse come attributi privati di tipo `const`, ciascuno dei quali avente la sua funzione *getter* e *setter*.

È stata pensata per essere una classe madre, utilizzabile come punto di partenza per l'implementazione di classi relative a particelle più complesse o aventi un maggior numero di particolarità.

ResonanceType

ResonanceType rappresenta una particella instabile, con le sue quattro caratteristiche: nome, massa, carica e larghezza di risonanza. Come per *ParticleType*, ad onguna di esse corrisponde un attributo privato `const` e due funzioni *getter* e *setter*.

Particle

Questa classe descrive il comportamento di una generica particella di tipo *ParticleType* o *ResonanceType* a seguito dell'urto.

Dopo una collisione le particelle generate acquistano un impulso, le cui componenti sono memorizzate come attributi privati della classe, e il metodo `decayTo` ne illustra l'eventuale decadimento.

Sono inoltre implementati metodi *getter* e *setter* per il modulo e le componenti della quanità di moto, la massa, l'energia e le caratteristiche del tipo di particella (tramite l'attributo `index_`, si veda più avanti).

Di rilevante importanza per la prova è la funzione membro `getInvMass`, la quale restituisce come valore la massa invariante fra i due prodotti del decadimento.

Metodi e attributi statici

Al fine di limitare l'uso di risorse da parte del programma, si è pensato di indicizzare le specie particellari generate: ad ogni *Particle* viene associato un numero (*index*) che ne indica la specie di appartenenza.

Mediante dei metodi statici è possibile usare tale indice per accedere a nome, massa, carica ed eventualmente larghezza della particella in questione.

pTypes È un array di puntatori a *ParticleType*. Ha la funzione di "tabella" che raccoglie le possibili *ParticleType* per una *Particle*.

addType Ha la funzione di riempire sequenzialmente `pTypes` attraverso l'allocazione dinamica.

findType Cerca in `pTypes` l'indice dell'elemento con nome corrispondente al nome fornito nel costruttore di *Particle*. Tale indice viene usato come valore per *index*.

1.2 Meccanismi di reimpiego

Nella scrittura del programma sono stati utilizzati due meccanismi di reimpiego codice: l'ereditarietà e l'aggregazione.

Ereditarietà

Considerando che *ResonanceType* è una tipologia *ParticleType* più specializzata (relazione "is a"), per riutilizzare il codice di *ParticleType* si è scelto di far ereditare *ResonanceType* da *ParticleType*.

È stato quindi aggiunto il metodo (virtuale) `getWidth` e in *ResonanceType* è stato ridefinito `print` (reso virtuale in *ParticleType*).

Aggregazione

Anche *Particle* è una specializzazione di *ParticleType*, tuttavia come meccanismo di reimpiego si è preferita la composizione, onde evitare la duplicazione delle proprietà di base delle particelle e un eccessivo utilizzo della memoria.

A tal scopo in *Particle* è incluso l'array statico `pTypes` atto a contenere i *ParticleType* inseriti in fase di generzione. Mediante questo array è possibile chiamarne i *getter* e accedere alle informazioni sulla specie della particella, senza doverle salvare ulteriormente in memoria. Ciascuna istanza di *Particle*, recupera le sue caratteristiche attraverso l'*index*, che corrisponde alla posizione del suo tipo in `pTypes`.

È possibile trovare il codice, dati raccolti e istogrammi di analisi su GitHub[2], assieme ad ulteriori informazioni e istruzioni per la compilazione.

In appendice si può trovare il listato integrale del codice.

2 Generazione

Sono stati simulati 10^5 eventi di collisione, ciascuno dei quali generanti 100 particelle, per un totale di 10^7 particelle. La tabella 1 ne illustra le proporzioni inserite in fase di generazione; la figura 1 mostra la distribuzione effettiva delle particelle generate.

Particella	Simbolo	Percentuale
Pione +	π^+	40%
Pione -	π^-	40%
Kaone +	K^+	5%
Kaone -	K^-	5%
Protone +	P^+	$4,5\%$
Protone -	P^-	$4,5\%$
Kaone neutro	K^0	1%

Tabella 1: Particelle generate da ogni collisione con le relative percentuali.

[2] https://github.com/KaldarrostaJazz/Physics_Laboratory

2.1 Schema di generazione

È stato impostato un doppio ciclo di generazione: quello esterno sui 10^5 eventi, quello interno sulle 100 particelle. Per la generazione di numeri pseudocasuali si è fatto uso della classe `TRandom` del pacchetto *ROOT*.

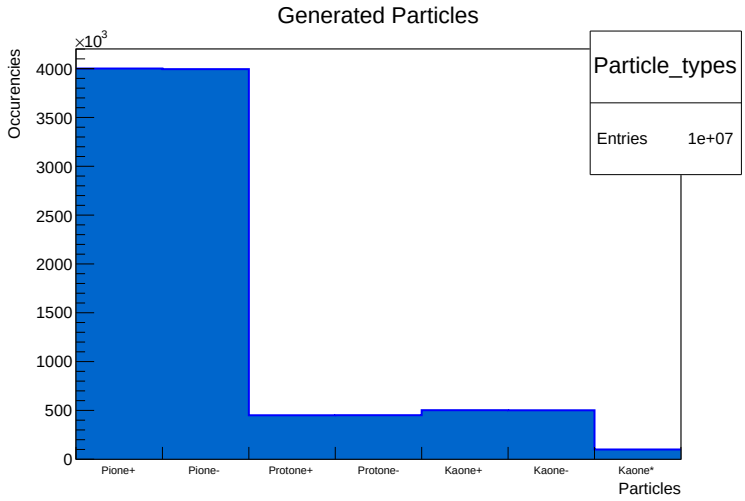

Figura 1: Distribuzione delle particelle generate dalle simulazione.

Per ogni collisione,

- si istanzia una *Particle* alla volta;
- seguendo un blocco *if* - *else if* - ... *else*, le viene assegnato un nome conformemente a quanto mostrato nella tabella 1;
- in proporzione a una distribuzione isotropa (figura 2), viene assegnato alla particella il suo impulso;
- nel caso la particella sia instabile (un kaone neutro), viene fatta decadere in una coppia π/K di carica opposta e se ne calcola la massa invariante;
- vengono inseriti i dati relativi alle proprietà delle particelle negli appositi istogrammi *ROOT* (figure 2, 3 ,4).

Le particelle generate in un evento (inclusi i prodotti di decadimenoto) vengono salvate in memoria. Prima dell'inizio dell'evento successivo, vengono calcolate le masse invarianti delle varie coppie

- particelle con carica concorde;
- particelle con carica discorde;
- π/K con carica concorde;
- π/K con carica discorde;

e vengono riempiti i rispettivi istogrammi (figura 5 e figura 6).

2.2 Proprietà cinematiche

La distribuzione degli impulsi è isotropa nello spazio. Per le componenti sono state usate le coordinate sferiche e gli angoli polare e azimutale sono stati generati secondo una distribuzione uniforme (figura 2). Il modulo dell'impulso segue invece una distribuzione esponenziale con parametro $\tau = 1$ (figura 3).

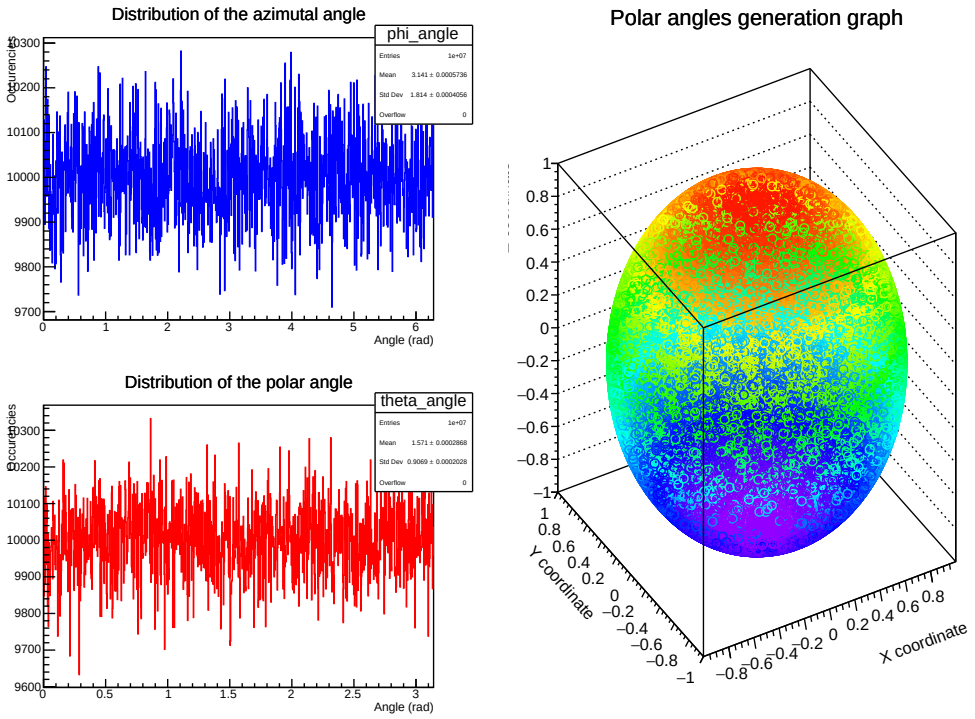

Figura 2: A destra, le distribuzioni degli angoli polari e azimutali generati. A sinistra la distribuzione degli impulsi nello spazio.

La concretezza dei valori generati e il loro grado di accordo con le *PDF* assegnate verrà trattato in 3. Gli istogrammi delle figure 2 e 3 ne mostrano le distribuzioni effettive.

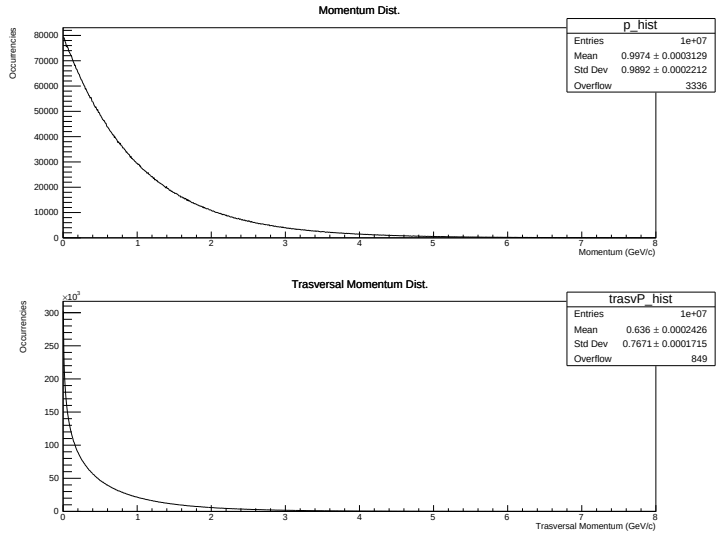

Figura 3: Sopra, la distribuzione del modulo dell'impuso. Sotto, la distribuzione del modulo dell'impuslo trasverso.

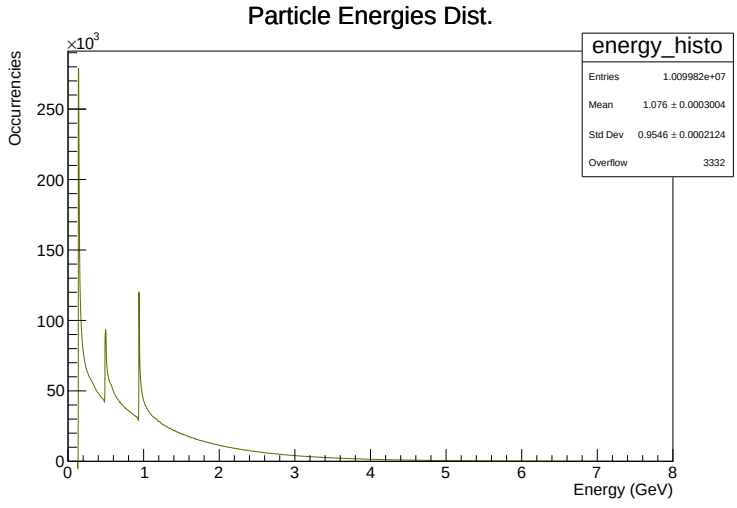

Figura 4: Distribuzione delle energie delle particelle dopo il decadimento. I picchi sono in corrispondenza delle masse delle particelle.

2.3 Decadimenti

Nel caso la particella generata sia un kaone K*, viene fatta decadere mediante il metodo `decayTo` in una coppia π/K di carica discorde (le due coppie sono equiprobabili). I prodotti del decadimento hanno impulsi opposti, in accordo col principio di conservazione della quantità di moto; le loro masse invarianti vengono raccolte in un istogramma (figura 5) che servirà come istogramma di controllo per l'analisi statistica.

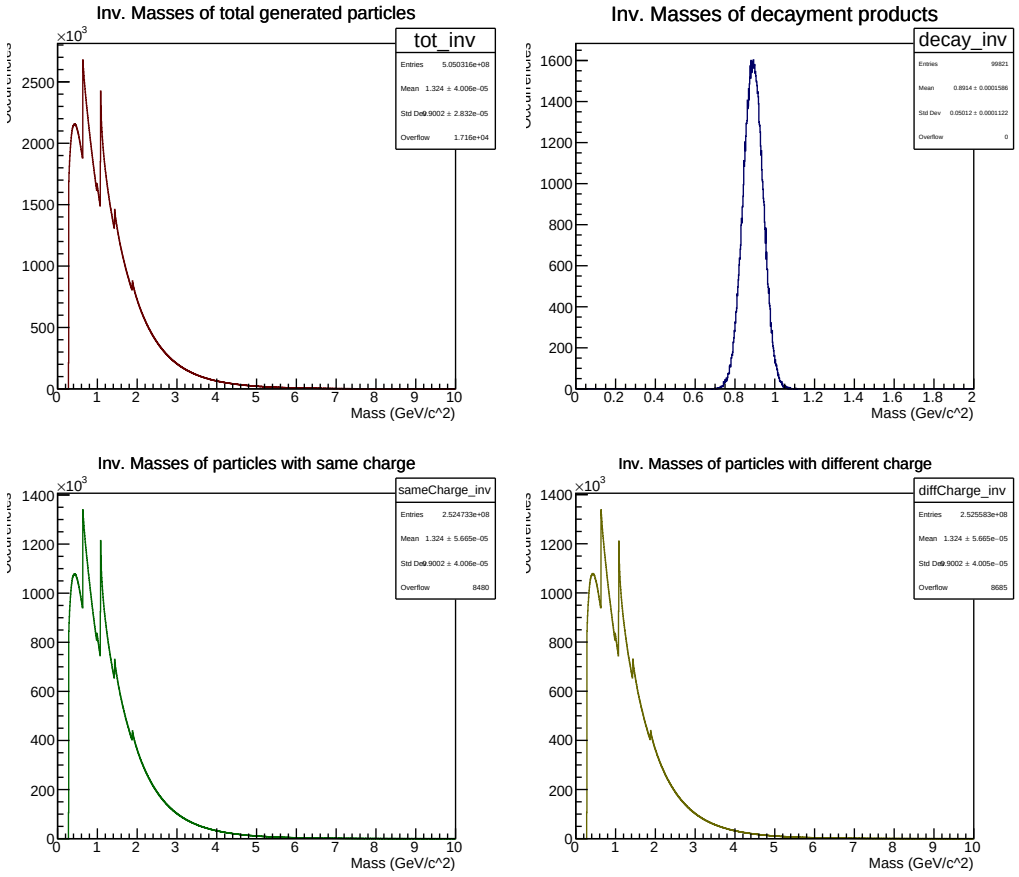

Figura 5: Le distribuzioni delle masse invarianti. Partendo da in alto a sinistra, in senso orario: tutte le particelle, solo prodotti di decadimento, coppie con carica opposta, coppie con carica concorde.

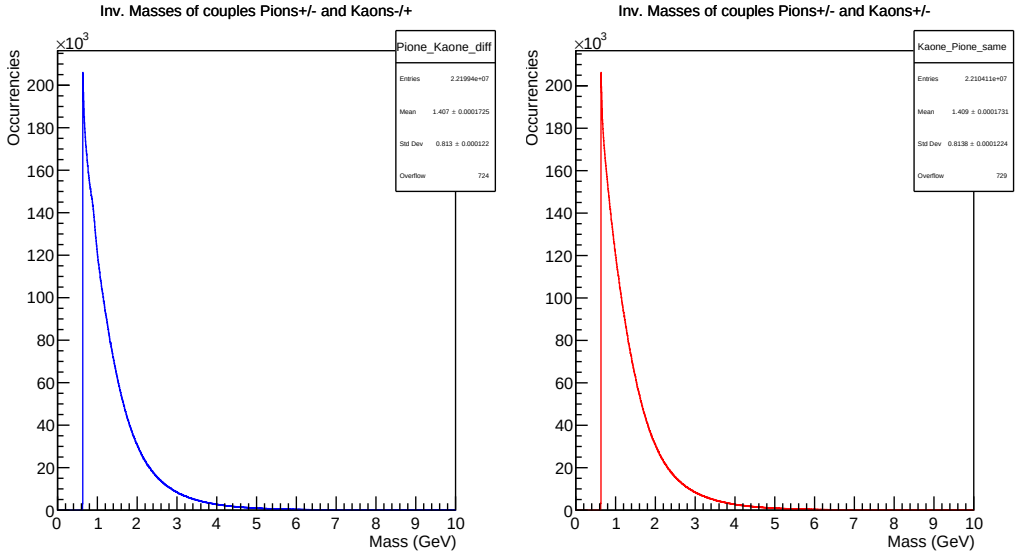

Figura 6: A sinistra, la distribuzione delle masse invarianti delle coppie π^+/K^-. A destra, la distribuzione delle masse invarianti delle coppie π^-/K^+. Dalla sottrazione dei due istogrammi si può isolare il segnale della K*.

3 Analisi

Si passa ora all'esposizione dei risultati dell'analisi della simulazione.

3.1 Concretezza dei dati generati

Il numero di particelle generato è in accordo con quanto previsto, entro l'incertezza dovuta alle fluttuazioni statistiche (tabella 2).

Specie	Occ. osservate	Occ. attese
π^+	$(4001 \pm 2)e3$	$4000e3$
π^-	$(3995 \pm 2)e3$	$4000e3$
K^+	$(501,3 \pm 0,7)e3$	$500e3$
K^-	$(502,2 \pm 0,7)e3$	$500e3$
P^+	$(450,1 \pm 0,7)e3$	$450e3$
P^-	$(450,9 \pm 0,7)e3$	$450e3$
K^0	$(99,8 \pm 0,3)e3$	$100e3$

Tabella 2: Abbondanza delle particelle.

Anche le distribuzioni di impulso e angoli polari e azimutali sono coerenti con i valori dati in input durante la fase di generazione.

La funzione pol0 (PDF uniforme) descrive appieno le distribuzioni degli angoli, tuttavia il valore di $\tilde{\chi}^2$ inferiore a 1 è indice di una leggera sovrastima delle incertezze sui conteggi, probabilmente dovuta a un numero di bin eccessivo.

Il modulo degli impulsi è stato generato seguendo una PDF esponenziale; il valore $\tilde{\chi}^2 = 0,99$ mostra un totale accordo con tale ipotesi. Come parametro della distribuzione in input al programma è stato dato $\tau = 1$ e il fit ha estratto dai dati un valore coerente entro l'errore.

La tabella 3 mostra i risulati di tali fit.

Distribuzione	Parametri fit	χ^2	DOF	χ^2/DOF
Angolo polare θ (pol0)	$9999 \pm 3rad$	989,7	990	0,99
Angolo azimutale ϕ (pol0)	$9999 \pm 3rad$	1016	999	1,02
Modulo impulso (expo)	$1,0000 \pm 0,0003GeV/c$	997,5	998	0,99

Tabella 3: Distribuzione angoli polari e azimutali, modulo dell'impulso.

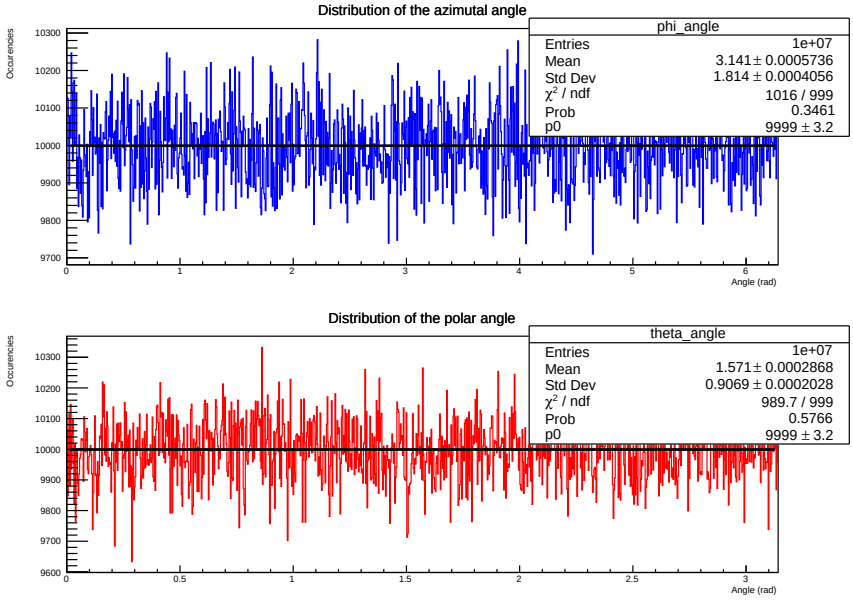

Figura 7: Fitting delle distribuzioni degli angoli. In nero il fit, a colori la distribuzione della generazione.

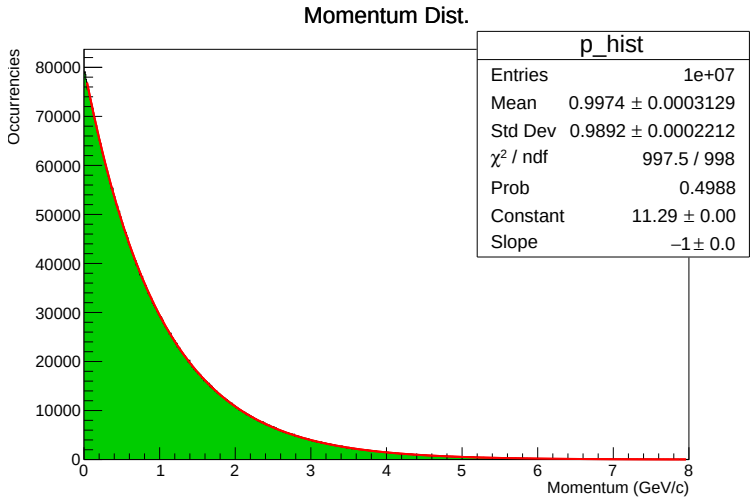

Figura 8: Fitting della distribuzione degli impulsi. In rosso il fit, in verde la distribuzione della generazione.

3.2 Analisi del segnale della risonanza

Il kaone neutro K* è un segnale difficile da individuare, sia a causa della sua breve vita media, sia a causa della sua rarità (circa 1% delle particelle totali). Il suo picco di risonanza, negli istogrammi delle masse invarianti, è coperto dai picchi delle altre combinazioni casuali (si veda la figura 5).

Tuttavia, conoscendo i prodotti di decadimento della K* è possibile isolarne il segnale in maniera indiretta e separarlo dal rumore di fondo:

1. Si calcolano le masse invarianti delle particelle con carica concorde e le si inseriscono nel relativo istogramma(H1). I prodotti della K* hanno carica opposta, dunque i picchi di questa distribuzione saranno in corrispondenza delle combinazioni accidentali tra le particelle.

2. Si calcolano le masse invarianti delle particelle con carica discorsi riempie il relativo istogramma (H2). I picchi di questo istogramma saranno in corrispondeza sia della combinazioni casuali, sia della K*.

3. Sottraendo l'istogramma H1 dall'istogramma H2 (H2 - H1) i picchi dovuti alle combinazioni casuali si elidono vicendevolmente; si ottiene una distribuzione oscillante attorno allo zero e con un netto picco in corrispondenza della risonanza della K*.

Selezionando gli istogrammi delle sole coppie π/K si ottieme una stima ancora più precisa e un picco più netto.

//	Media GeV/c^2	Sigma GeV/c^2	Ampiezza	χ^2/DOF
Istogramma di controllo	$0,8914 \pm 0,0002$	$0,0500 \pm 0,0001$	1592 ± 6	0,85
Differenza discor-di/concordi	$0,897 \pm 0,005$	$0,048 \pm 0,004$	$(53 \pm 5)e2$	0,98
Differenza π/K discor-di/concordi	$0,894 \pm 0,002$	$0,051 \pm 0,002$	$(51,8 \pm 1,8)e2$	0,97

Tabella 4: Analisi degli istogrammi sottrazione.

I valor inserit in input al programma sono $m = 0,89166 GeV/c^2$ e $\Gamma = 0,050 GeV/c^2$. La tabella 4 mostra i risultati dell'analisi. Sia sottraendo tutte

le coppie, che sottraendo solo le coppie π/K, si ottengono dei valori coerenti entro gli errori.

Risonanza e larghezza di risonanza dell'istogramma di controllo sono anch'esse in accordo con quanto inserito in fase di generazione.

Le figure 9, 10, 11 mostrano i vari istogrammi analizzati.

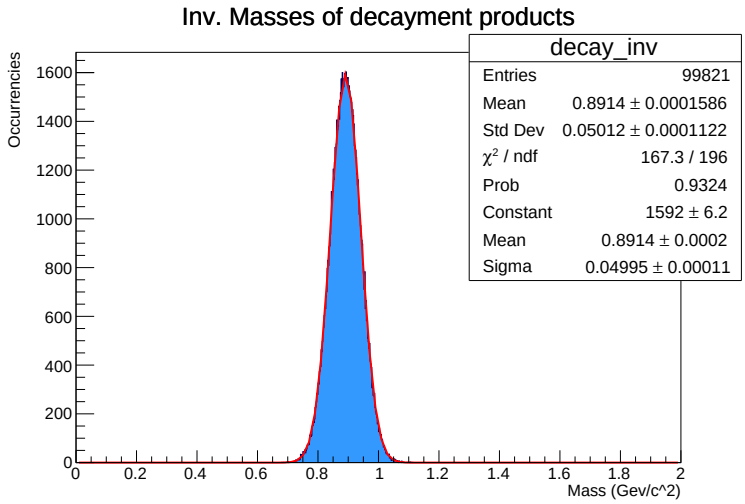

Figura 9: Istogramma di controllo. In rosso il fit, in azzurro la distribuzione delle masse invarianti dei prodotti del decadimento.

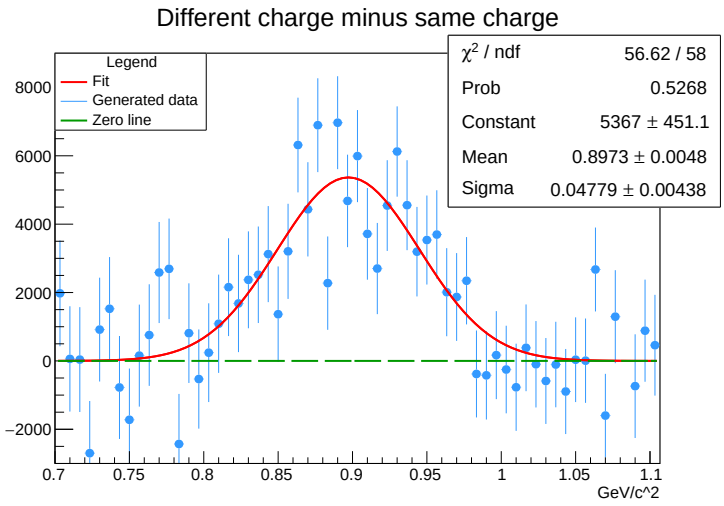

Figura 10: Sottrazione degli istogrammi delle masse invarianti delle coppie con cariche discordi e cariche concordi.

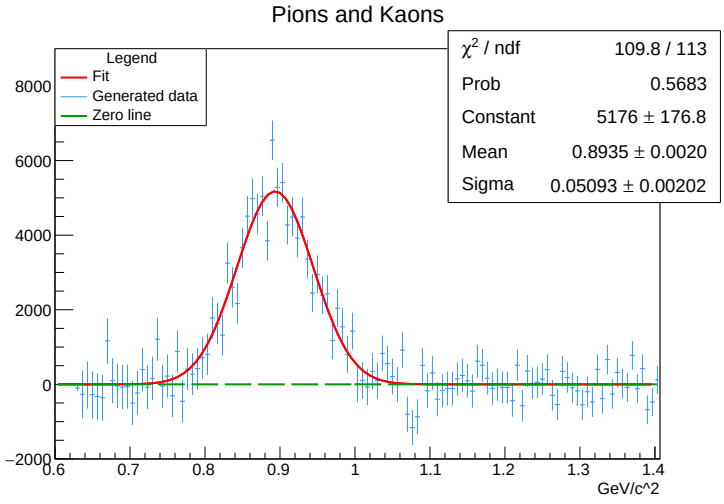

Figura 11: Sottrazione degli istogrammi delle masse invarianti delle coppie π/K.

A ParticleType.hpp

```
#ifndef PARTICLETYPE_HPP
```

```
#define PARTICLETYPE_HPP
#include <exception>
#include <iostream>
class ParticleType {
 public:
  ParticleType(const char* name, double mass, int charge);
  ParticleType(ParticleType& type);
  const char* getName() const { return name_; }
  double getMass() const { return mass_; }
  int getCharge() const { return charge_; }
  virtual double getWidth() const { return 0; }
  virtual void print() const;

 private:
  const char* name_;
  const double mass_;
  const int charge_;
};
#endif
```

B ParticleType.cpp

```
#include "ParticleType.hpp"
ParticleType::ParticleType(const char* name, double mass, int charge)
    : name_{name}, mass_{mass}, charge_{charge} {}
ParticleType::ParticleType(ParticleType& type)
    : name_{type.getName()}, mass_{type.getMass()}, charge_{type.getCharge()} {}
void ParticleType::print() const {
  std::cout << "Name: " << name_ << " Mass: " << mass_ << " Charge: " << charge_
            << '\n';
}
```

C ResonanceType.hpp

```
#ifndef RESONANCETYPE_HPP
#define RESONANCETYPE_HPP
#include "ParticleType.hpp"

class ResonanceType : public ParticleType {
```

```
 public:
  ResonanceType(const char* name, double mass, int charge, double width);
  ResonanceType(ResonanceType& type);
  double getWidth() const { return width_; }
  void print() const;

 private:
  double const width_;
};
#endif
```

D ResonanceType.cpp

```
#include "ResonanceType.hpp"
ResonanceType::ResonanceType(const char* name, double mass, int charge,
                             double width)
    : ParticleType{name, mass, charge}, width_{width} {}
ResonanceType::ResonanceType(ResonanceType& type)
    : ParticleType{type.getName(), type.getMass(), type.getCharge()},
      width_{type.getWidth()} {}
void ResonanceType::print() const {
  ParticleType::print();
  std::cout << " Width: " << width_ << '\n';
}
```

E Particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include <cmath>
#include <cstdlib>
#include <string>

#include "ResonanceType.hpp"

class Particle {
 public:
  // Constructor /////
```

```
Particle();
Particle(const char* name, double px, double py, double pz);
Particle(Particle& p);
////////////////////

// Operator = //////
Particle& operator=(const Particle& other);
////////////////////

// Decay method ////
int decayTo(Particle& p1, Particle& p2) const;
////////////////////

// Getters /////////
int getIndex() const { return index_; }
double getPx() const { return px_; }
double getPy() const { return py_; }
double getPz() const { return pz_; }
double getP() const { return std::sqrt(px_ * px_ + py_ * py_ + pz_ * pz_); }
double getMass() const { return pTypes[index_]->getMass(); }
double getEnergy() const;
double getInvMass(Particle& particle);
static int getNumTypes() { return numTypes; }
static ParticleType* getPType(int index) { return pTypes[index]; }
////////////////////

// Setters /////////
void setIndex(int index);
void setIndex(const char* name);
void setP(double px, double py, double pz);
////////////////////

// Static methods //
static void addType(const char* name, double mass, int charge,
                    double width = 0.);
static void addType(ParticleType& newType);
static void addType(ResonanceType& newType);
////////////////////

// Printers ////////
void print() const;
```

```
  static void printTypes();
  ////////////////////

 private:
  static const int maxTypes;
  static ParticleType* pTypes[];
  static int numTypes;

  static int findType(const char* name);

  void boost(double bx, double by, double bz);

  int index_;
  double px_{0.};
  double py_{0.};
  double pz_{0.};
};
inline Particle& Particle::operator=(const Particle& other) {
  this->index_ = other.getIndex();
  this->px_ = other.getPx();
  this->py_ = other.getPy();
  this->pz_ = other.getPz();
  return *this;
}
#endif
```

F Particle.cpp

```
#include "Particle.hpp"

Particle::Particle() = default;
Particle::Particle(const char* name, double px, double py, double pz)
    : px_{px}, py_{py}, pz_{pz} {
  index_ = findType(name);
  if (index_ == numTypes) {
    index_ = -1;
    std::string error{"Can't find "};
    error.append(name);
    error.append(" among the particle types.\n");
    throw std::runtime_error(error);
```

```
  }
}
Particle::Particle(Particle& p)
    : index_{p.getIndex()}, px_{p.getPx()}, py_{p.getPy()}, pz_{p.getPz()} {}

int Particle::decayTo(Particle& p1, Particle& p2) const {
  if (getMass() == 0.0) {
    throw std::runtime_error("Decayment cannot be performed if mass is zero\n");
    return 1;
  }
  double massMot = getMass();
  double massP1 = p1.getMass();
  double massP2 = p2.getMass();
  if (index_ > -1) {
    float x1;
    float x2;
    float w;
    float y1;
    float y2;
    double invnum = 1. / RAND_MAX;
    do {
      x1 = 2. * rand() * invnum - 1.;
      x2 = 2. * rand() * invnum - 1.;
      w = x1 * x1 + x2 * x2;
    } while (w >= 1.);
    w = std::sqrt((-2. * std::log(w)) / w);
    y1 = x1 * w;
    y2 = x2 * w;
    massMot += pTypes[index_]->getWidth() * y1;
  }
  if (massMot < massP1 + massP2) {
    throw std::runtime_error(
        "Decayment cannot be performed because mass is too low in this "
        "channel\n");
    return 2;
  }
  double pout =
      std::sqrt((massMot * massMot - (massP1 + massP2) * (massP1 + massP2)) *
                (massMot * massMot - (massP1 - massP2) * (massP1 - massP2))) /
      massMot * 0.5;
  double norm = 2 * M_PI / RAND_MAX;
```

```
  double phi = rand() * norm;
  double theta = rand() * norm * 0.5 - M_PI / 2.;
  p1.setP(pout * std::sin(theta) * std::cos(phi),
          pout * std::sin(theta) * std::sin(phi), pout * std::cos(theta));
  p2.setP(-pout * std::sin(theta) * std::cos(phi),
          -pout * std::sin(theta) * std::sin(phi), -pout * std::cos(theta));
  double energy =
      std::sqrt(px_ * px_ + py_ * py_ + pz_ * pz_ + massMot * massMot);
  double bx = px_ / energy;
  double by = py_ / energy;
  double bz = pz_ / energy;
  p1.boost(bx, by, bz);
  p2.boost(bx, by, bz);
  return 0;
}

double Particle::getEnergy() const {
  double p_square = px_ * px_ + py_ * py_ + pz_ * pz_;
  double result = getMass() * getMass() + p_square;
  return std::sqrt(result);
}
double Particle::getInvMass(Particle& particle) {
  double px_tot = px_ + particle.getPx();
  double py_tot = py_ + particle.getPy();
  double pz_tot = pz_ + particle.getPz();
  double p_tot_square = px_tot * px_tot + py_tot * py_tot + pz_tot * pz_tot;
  double energy_tot = getEnergy() + particle.getEnergy();
  double result = energy_tot * energy_tot - p_tot_square;
  return std::sqrt(result);
}

void Particle::setIndex(int index) {
  if (index < numTypes) {
    index_ = index;
  } else {
    std::string error{"Type "};
    error.append(std::to_string(index));
    error.append(" doesn't exist.\n");
    throw std::runtime_error(error);
  }
}
```

```
void Particle::setIndex(const char* name) {
  if (findType(name) < numTypes) {
    index_ = findType(name);
  } else {
    std::string error{name};
    error.append(" doesn't exists.\n");
    throw std::runtime_error(error);
  }
}
void Particle::setP(double px, double py, double pz) {
  px_ = px;
  py_ = py;
  pz_ = pz;
}

void Particle::addType(const char* name, double mass, int charge,
                       double width) {
  if (numTypes <= 10) {
    if (findType(name) < numTypes) {
      std::string error{"Particle "};
      error.append(name);
      error.append(" already added.\n");
      throw std::runtime_error(error);
    } else {
      if (width != 0.) {
        pTypes[numTypes] = new ResonanceType{name, mass, charge, width};
        numTypes++;
      } else {
        pTypes[numTypes] = new ParticleType{name, mass, charge};
        numTypes++;
      }
    }
  } else {
    std::string error{"Can't add "};
    error.append(name);
    error.append(". Maximum number of particles reached (");
    error.append(std::to_string(maxTypes));
    error.append(").\n");
    throw std::runtime_error(error);
  }
}
```

```
void Particle::addType(ParticleType& newType) {
  const char* name = newType.getName();
  if (numTypes <= 10) {
    if (findType(name) < numTypes) {
      std::string error{"Particle "};
      error.append(name);
      error.append(" already added.\n");
      throw std::runtime_error(error);
    } else {
      pTypes[numTypes] = new ParticleType{newType};
      numTypes++;
    }
  } else {
    std::string error{"Can't add "};
    error.append(name);
    error.append(". Maximum number of particles reached (");
    error.append(std::to_string(maxTypes));
    error.append(").\n");
    throw std::runtime_error(error);
  }
}
void Particle::addType(ResonanceType& newType) {
  const char* name = newType.getName();
  if (numTypes <= 10) {
    if (findType(name) < numTypes) {
      std::string error{"Particle "};
      error.append(name);
      error.append(" already added.\n");
      throw std::runtime_error(error);
    } else {
      pTypes[numTypes] = new ResonanceType{newType};
      numTypes++;
    }
  } else {
    std::string error{"Can't add "};
    error.append(name);
    error.append(". Maximum number of particles reached (");
    error.append(std::to_string(maxTypes));
    error.append(").\n");
    throw std::runtime_error(error);
  }
```

```
}

void Particle::print() const {
  std::cout << "Index: " << index_ << " Name: " << pTypes[index_]->getName()
            << " Vector P: (" << px_ << ',' << py_ << ',' << pz_ << ")\n";
}
void Particle::printTypes() {
  for (int i = 0; i != numTypes; ++i) {
    pTypes[i]->print();
  }
}

const int Particle::maxTypes = 10;
ParticleType* Particle::pTypes[maxTypes];
int Particle::numTypes{0};

int Particle::findType(const char* name) {
  int index = 0;
  // auto first = pTypes[0];
  // auto last = pTypes[numTypes];
  for (; index != numTypes; ++index) {
    if (pTypes[index]->getName() == name) {
      return index;
    }
  }
  return index;
}

void Particle::boost(double bx, double by, double bz) {
  double energy = getEnergy();
  double b2 = bx * bx + by * by + bz * bz;
  double gamma = 1. / std::sqrt(1. - b2);
  double bp = bx * px_ + by * py_ + bz * pz_;
  double gamma2 = b2 > 0 ? (gamma - 1.) / b2 : 0.;
  px_ += gamma2 * bp * bx + gamma * bx * energy;
  py_ += gamma2 * bp * by + gamma * by * energy;
  pz_ += gamma2 * bp * bz + gamma * bz * energy;
}
```

G main.cpp (generazione)

```
#include <TCanvas.h>
#include <TFile.h>
#include <TGraph2D.h>
#include <TH1F.h>
#include <TMath.h>
#include <TPad.h>
#include <TPaveStats.h>
#include <TRandom.h>
#include <TStyle.h>

#include "Particle.hpp"
#include "ParticleType.hpp"
#include "ResonanceType.hpp"

int main() {
  gStyle->SetPalette(1);
  gStyle->SetOptStat("neMRo");

  TRandom* Random = new TRandom();
  Random->SetSeed();

  // Adding the particle types
  Particle::addType("Pione+", 0.13957, 1);
  Particle::addType("Pione-", 0.13957, -1);
  Particle::addType("Protone+", 0.93827, 1);
  Particle::addType("Protone-", 0.93827, -1);
  Particle::addType("Kaone+", 0.49367, 1);
  Particle::addType("Kaone-", 0.49367, -1);
  Particle::addType("Kaone*", 0.89166, 0, 0.050);

  // Histogram of the generated particles
  //
  TH1::SetDefaultSumw2(kTRUE);  // every time TH1 object is created
                                // automatically calls TH1::Sumw2();
  //
  TH1F* types = new TH1F("Particle_types", "Generated Particles", 7, -0.5, 6.5);
  types->GetXaxis()->SetTitle("Particles");
  types->GetYaxis()->SetTitle("Occurencies");
  types->GetXaxis()->SetBinLabel(1, "Pione+");
```

```
types->GetXaxis()->SetBinLabel(2, "Pione-");
types->GetXaxis()->SetBinLabel(3, "Protone+");
types->GetXaxis()->SetBinLabel(4, "Protone-");
types->GetXaxis()->SetBinLabel(5, "Kaone+");
types->GetXaxis()->SetBinLabel(6, "Kaone-");
types->GetXaxis()->SetBinLabel(7, "Kaone*");
types->SetFillColor(kAzure + 2);
types->SetLineColor(kBlue);
types->SetLineWidth(2);
types->Sumw2(kFALSE);

// Histograms and graph of momentum directions (theta and phi angle)
TH1F* phi_hist = new TH1F("phi_angle", "Distribution of the azimutal angle",
                          1000, 0., 2. * TMath::Pi());
phi_hist->GetXaxis()->SetTitle("Angle (rad)");
phi_hist->GetYaxis()->SetTitle("Occurencies");
phi_hist->SetFillColor(0);
phi_hist->SetLineColor(kBlue);
phi_hist->SetLineWidth(1);
phi_hist->Sumw2(kFALSE);
TH1F* theta_hist = new TH1F("theta_angle", "Distribution of the polar angle",
                            1000, 0., TMath::Pi());
theta_hist->GetXaxis()->SetTitle("Angle (rad)");
theta_hist->GetYaxis()->SetTitle("Occurencies");
theta_hist->SetFillColor(0);
theta_hist->SetLineColor(kRed);
theta_hist->SetLineWidth(1);
theta_hist->Sumw2(kFALSE);
TGraph2D* angles_graph = new TGraph2D(1E5);
angles_graph->SetTitle(
    "Polar angles generation graph; X coordinate; Y coordinate; Z "
    "coordinate");
angles_graph->GetXaxis()->SetTitleOffset(2.);
angles_graph->GetYaxis()->SetTitleOffset(2.);
angles_graph->GetZaxis()->SetTitleOffset(2.);
angles_graph->SetMarkerStyle(kOpenCircle);
int point_count{0};

// Histogram of momentum and trasversal momentum values
TH1F* p_hist = new TH1F("p_hist", "Momentum Dist.", 1000, 0., 8.);
p_hist->GetXaxis()->SetTitle("Momentum (GeV/c)");
```

```
p_hist->GetYaxis()->SetTitle("Occurrencies");
p_hist->SetLineColor(kBlack);
p_hist->SetLineWidth(1);
p_hist->Sumw2(kFALSE);
TH1F* trasvP_hist =
    new TH1F("trasvP_hist", "Trasversal Momentum Dist.", 1000, 0., 8.);
trasvP_hist->GetXaxis()->SetTitle("Trasversal Momentum (GeV/c)");
trasvP_hist->GetYaxis()->SetTitle("Occurrencies");
trasvP_hist->SetLineColor(kBlack);
trasvP_hist->SetLineWidth(1);
trasvP_hist->Sumw2(kFALSE);

// Histogram of particle energies
TH1F* energy_histo =
    new TH1F("energy_histo", "Particle Energies Dist.", 1000, 0., 8.);
energy_histo->GetXaxis()->SetTitle("Energy (GeV)");
energy_histo->GetYaxis()->SetTitle("Occurrencies");
energy_histo->SetLineColor(kYellow + 3);
energy_histo->SetLineWidth(1);
energy_histo->Sumw2(kFALSE);

// Histogram of Invariant Masses
TH1F* decay_inv =
    new TH1F("decay_inv", "Inv. Masses of decayment products", 1000, 0., 2.);
decay_inv->GetXaxis()->SetTitle("Mass (Gev/c^2)");
decay_inv->GetYaxis()->SetTitle("Occurrencies");
decay_inv->SetLineColor(kBlue + 3);
decay_inv->SetLineWidth(1);
decay_inv->Sumw2(kFALSE);
TH1F* tot_inv = new TH1F(
    "tot_inv", "Inv. Masses of total generated particles", 1500, 0., 10.);
tot_inv->GetXaxis()->SetTitle("Mass (GeV/c^2)");
tot_inv->GetYaxis()->SetTitle("Occurencies");
tot_inv->SetLineColor(kRed + 3);
tot_inv->SetLineWidth(1);
TH1F* sameCharge_inv =
    new TH1F("sameCharge_inv", "Inv. Masses of particles with same charge",
             1500, 0., 10.);
sameCharge_inv->GetXaxis()->SetTitle("Mass (GeV/c^2)");
sameCharge_inv->GetYaxis()->SetTitle("Occurencies");
sameCharge_inv->SetLineColor(kGreen + 3);
```

```
sameCharge_inv->SetLineWidth(1);
TH1F* diffCharge_inv =
    new TH1F("diffCharge_inv",
             "Inv. Masses of particles with different charge", 1500, 0., 10.);
diffCharge_inv->GetXaxis()->SetTitle("Mass (GeV/c^2)");
diffCharge_inv->GetYaxis()->SetTitle("Occurencies");
diffCharge_inv->SetLineColor(kYellow + 3);
diffCharge_inv->SetLineWidth(1);
TH1F* piK_inv =
    new TH1F("Pione_Kaone_diff",
             "Inv. Masses of couples Pions+/- and Kaons-/+", 1500, 0., 10.);
piK_inv->GetXaxis()->SetTitle("Mass (GeV)");
piK_inv->GetYaxis()->SetTitle("Occurrencies");
piK_inv->SetLineColor(kBlue);
TH1F* Kpi_inv =
    new TH1F("Kaone_Pione_same",
             "Inv. Masses of couples Pions+/- and Kaons+/-", 1500, 0., 10.);
Kpi_inv->GetXaxis()->SetTitle("Mass (GeV)");
Kpi_inv->GetYaxis()->SetTitle("Occurrencies");
Kpi_inv->SetLineColor(kRed);

// Loop generating the particles
for (int i = 0; i != 1E5; ++i) {
  Particle array[130];
  int count = 0;

  for (int j = 0; j != 100; ++j) {
    Particle particle;

    double phi = Random->Uniform(0., 2. * TMath::Pi());
    double theta = Random->Uniform(0., TMath::Pi());
    double p = Random->Exp(1.);
    double px = p * std::sin(theta) * std::cos(phi);
    double py = p * std::sin(theta) * std::sin(phi);
    double pz = p * std::cos(theta);
    particle.setP(px, py, pz);

    double r = Random->Rndm();
    if (r < 0.8) {
      double a = Random->Rndm();
      if (a < 0.5) {
```

```
    particle.setIndex("Pione+");
  } else {
    particle.setIndex("Pione-");
  }
  array[count++] = particle;
  energy_histo->Fill(particle.getEnergy());
}
if (0.8 <= r && r < 0.9) {
  double a = Random->Rndm();
  if (a < 0.5) {
    particle.setIndex("Kaone+");
  } else {
    particle.setIndex("Kaone-");
  }
  array[count++] = particle;
  energy_histo->Fill(particle.getEnergy());
}
if (0.9 <= r && r < 0.99) {
  double a = Random->Rndm();
  if (a < 0.5) {
    particle.setIndex("Protone+");
  } else {
    particle.setIndex("Protone-");
  }
  array[count++] = particle;
  energy_histo->Fill(particle.getEnergy());
}
if (0.99 <= r && r < 1) {
  particle.setIndex("Kaone*");
  Particle p1;
  Particle p2;
  double a = Random->Rndm();
  if (a < 0.5) {
    p1.setIndex("Pione+");
    p2.setIndex("Kaone-");
    particle.decayTo(p1, p2);
  } else {
    p1.setIndex("Pione-");
    p2.setIndex("Kaone+");
    particle.decayTo(p1, p2);
  }
```

```
      array[count++] = p1;
      array[count++] = p2;
      energy_histo->Fill(p1.getEnergy());
      energy_histo->Fill(p2.getEnergy());
      decay_inv->Fill(p1.getInvMass(p2));
    }

    p_hist->Fill(p);
    trasvP_hist->Fill(std::sqrt(px * px + py * py));
    types->Fill(particle.getIndex());
    phi_hist->Fill(phi);
    theta_hist->Fill(theta);

    if (point_count < 1E5) {
      double x = std::sin(theta) * std::cos(phi);
      double y = std::sin(theta) * std::sin(phi);
      double z = std::cos(theta);
      angles_graph->SetPoint(point_count++, x, y, z);
    }
  }
  for (int k = 0; k != count; ++k) {
    int k_charge = Particle::getPType(array[k].getIndex())->getCharge();
    for (int l = k + 1; l != count; ++l) {
      double invMass = array[k].getInvMass(array[l]);
      tot_inv->Fill(invMass);
      int l_charge = Particle::getPType(array[l].getIndex())->getCharge();
      if (k_charge == l_charge) {
        sameCharge_inv->Fill(invMass);
      } else if (k_charge != l_charge) {
        diffCharge_inv->Fill(invMass);
      }
      if ((array[k].getIndex() == 0 && array[l].getIndex() == 5) ||
          (array[k].getIndex() == 1 && array[l].getIndex() == 4)) {
        piK_inv->Fill(invMass);
      }
      if ((array[k].getIndex() == 0 && array[l].getIndex() == 4) ||
          (array[k].getIndex() == 1 && array[l].getIndex() == 5)) {
        Kpi_inv->Fill(invMass);
      }
    }
```

```
  }
}

// Plotting on canvas
TCanvas* types_canva = new TCanvas("types_canva");
types_canva->SetTitle("Particelle Generate");
types->Draw("HIST SAME");
gPad->Update();
TPaveStats* st = (TPaveStats*)types->FindObject("stats");
st->SetOptStat(11);

TCanvas* angles_canva = new TCanvas("angles_canva");
angles_canva->SetWindowSize(1200, 900);
angles_canva->SetTitle("Angoli Generati");
angles_canva->Divide(2, 1);
angles_canva->cd(1)->Divide(1, 2);
angles_canva->cd(1)->cd(1);
phi_hist->Draw("HIST SAME");
angles_canva->cd(1)->cd(2);
theta_hist->Draw("HIST SAME");
angles_canva->cd(2);
angles_graph->Draw("PCOL");

TCanvas* p_canva = new TCanvas("p_canvas");
p_canva->SetWindowSize(1200, 900);
p_canva->SetTitle("Momentum Dist.");
p_canva->Divide(1, 2);
p_canva->cd(1);
p_hist->Draw("HIST SAME C");
p_canva->cd(2);
trasvP_hist->Draw("HIST SAME C");

TCanvas* energy_canva = new TCanvas("energy_canva");
energy_canva->SetTitle("Energis Dist.");
energy_histo->Draw("HIST SAME C");

TCanvas* invMass_canva = new TCanvas("invMass_canva");
invMass_canva->SetWindowSize(1000, 1000);
invMass_canva->Divide(2, 2);
invMass_canva->cd(1);
tot_inv->Draw("HIST SAME");
```

```
  invMass_canva->cd(2);
  decay_inv->Draw("HIST SAME");
  invMass_canva->cd(3);
  sameCharge_inv->Draw("HIST SAME");
  invMass_canva->cd(4);
  diffCharge_inv->Draw("HIST SAME");

  TCanvas* couples_canva = new TCanvas("couples_canva");
  couples_canva->SetWindowSize(1200, 600);
  couples_canva->Divide(2, 1);
  couples_canva->cd(1);
  piK_inv->Draw("HIST SAME");
  couples_canva->cd(2);
  Kpi_inv->Draw("HIST SAME");

  // Saving histos on .root file
  TFile* file = new TFile("../rootfiles/data.root", "RECREATE");
  types->Write();
  phi_hist->Write();
  theta_hist->Write();
  p_hist->Write();
  trasvP_hist->Write();
  energy_histo->Write();
  tot_inv->Write();
  decay_inv->Write();
  sameCharge_inv->Write();
  diffCharge_inv->Write();
  piK_inv->Write();
  Kpi_inv->Write();
  file->Close();

  // Printing pdfs
  types_canva->Print("../pdfs/types_canva.pdf");
  angles_canva->Print("../pdfs/angles_canva.pdf");
  p_canva->Print("../pdfs/p_canva.pdf");
  energy_canva->Print("../pdfs/energy_canva.pdf");
  invMass_canva->Print("../pdfs/invMass_canva.pdf");
  couples_canva->Print("../pdfs/couples_canva.pdf");

  return 0;
}
```

H test.cpp

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "Particle.hpp"
#include "ParticleType.hpp"
#include "ResonanceType.hpp"
#include "doctest.h"

TEST_CASE("ParticleType, ResonanceType") {
  ParticleType particle{"Particella", 1.7, 1};
  ResonanceType resonance{"Risonanza", 2.5, -1, 0.7};
  ParticleType* particelle[2];
  particelle[0] = &particle;
  particelle[1] = &resonance;
  particelle[0]->print();
  particelle[1]->print();
  std::cout << std::endl;

  CHECK(particle.getName() == "Particella");
  CHECK(particle.getMass() == 1.7);
  CHECK(particle.getCharge() == 1);
  CHECK(resonance.getName() == "Risonanza");
  CHECK(resonance.getMass() == 2.5);
  CHECK(resonance.getCharge() == -1);
  CHECK(resonance.getWidth() == 0.7);
}

TEST_CASE("Particle: addType on the heap") {
  Particle::addType("Protone", 938.272, 1);
  Particle::addType("Elettrone", 0.511, -1);
  Particle::addType("Neutrone", 939.565, 0);
  Particle::addType("Pione+", 139.6, 1, 6.11);
  Particle::addType("Pione-", 139.6, -1, 6.11);
  Particle::addType("Kaone", 497.648, 0, 19.23);

  CHECK_THROWS(Particle::addType("Protone", 12.3, 2, 4.5));
}

TEST_CASE("Particles properties") {
  Particle particella_1{"Protone", 1., 1., 1.};
  Particle particella_2{"Elettrone", 1., 1., 1.};
```

```
  Particle particella_3{"Neutrone", 1., 1., 1.};
  Particle particella_4{"Pione+", 1., 1., 1.};
  Particle particella_5{"Pione-", 1., 1., 1.};
  Particle particella_6{"Kaone", 1., 1., 1.};

  particella_1.print();
  particella_2.print();
  particella_3.print();
  particella_4.print();
  particella_5.print();
  particella_6.print();
  std::cout << std::endl;

  CHECK(particella_1.getIndex() == 0);
  CHECK(particella_2.getIndex() == 1);
  CHECK(particella_3.getIndex() == 2);
  CHECK(particella_4.getIndex() == 3);
  CHECK(particella_5.getIndex() == 4);
  CHECK(particella_6.getIndex() == 5);

  CHECK(particella_4.getMass() == doctest::Approx(139.6));
  CHECK(particella_2.getEnergy() == doctest::Approx(1.805857414));

  particella_2.setIndex("Kaone");
  CHECK(particella_2.getIndex() == 5);

  CHECK_THROWS(particella_1.setIndex(11));
  CHECK_THROWS(particella_1.setIndex("StdExcept"));

  particella_5.setP(2., 3.4, 0.6);
  CHECK(particella_5.getPx() == doctest::Approx(2.));
  CHECK(particella_5.getPy() == doctest::Approx(3.4));
  CHECK(particella_5.getPz() == doctest::Approx(0.6));
}

TEST_CASE("pTypes properties") {
  CHECK(Particle::getNumTypes() == 6);

  CHECK(Particle::getPType(0)->getName() == "Protone");
  CHECK(Particle::getPType(1)->getName() == "Elettrone");
  CHECK(Particle::getPType(2)->getName() == "Neutrone");
```

```
  CHECK(Particle::getPType(3)->getName() == "Pione+");
  CHECK(Particle::getPType(4)->getName() == "Pione-");
  CHECK(Particle::getPType(5)->getName() == "Kaone");

  CHECK(Particle::getPType(0)->getMass() == doctest::Approx(938.272));
  CHECK(Particle::getPType(1)->getMass() == doctest::Approx(0.511));
  CHECK(Particle::getPType(2)->getMass() == doctest::Approx(939.565));
  CHECK(Particle::getPType(3)->getMass() == doctest::Approx(139.6));
  CHECK(Particle::getPType(4)->getMass() == doctest::Approx(139.6));
  CHECK(Particle::getPType(5)->getMass() == doctest::Approx(497.648));

  CHECK(Particle::getPType(0)->getCharge() == 1);
  CHECK(Particle::getPType(1)->getCharge() == -1);
  CHECK(Particle::getPType(2)->getCharge() == 0);
  CHECK(Particle::getPType(3)->getCharge() == 1);
  CHECK(Particle::getPType(4)->getCharge() == -1);
  CHECK(Particle::getPType(5)->getCharge() == 0);

  Particle::printTypes();
  std::cout << std::endl;
}

TEST_CASE("addType from reference") {
  ResonanceType muone{"Muone", 105.6, -1, 0.457};
  ParticleType tauone{"Tauone", 1776.86, -1};
  Particle::addType(muone);
  Particle::addType(tauone);

  CHECK(Particle::getNumTypes() == 8);

  CHECK(Particle::getPType(6)->getName() == "Muone");
  CHECK(Particle::getPType(7)->getName() == "Tauone");
  CHECK(Particle::getPType(6)->getMass() == doctest::Approx(105.6));
  CHECK(Particle::getPType(7)->getMass() == doctest::Approx(1776.86));
  CHECK(Particle::getPType(6)->getCharge() == -1);
  CHECK(Particle::getPType(7)->getCharge() == -1);

  Particle::printTypes();
  std::cout << std::endl;
}
```

I main.cpp (analisi)

```
#include <TCanvas.h>
#include <TF1.h>
#include <TFile.h>
#include <TH1.h>
#include <TLegend.h>
#include <TPad.h>
#include <TPaveStats.h>
#include <TStyle.h>

#include <cmath>
#include <iomanip>
#include <iostream>

int main() {
  //
  // Style
  //
  gStyle->SetOptStat("neMR");
  gStyle->SetOptFit(1111);

  //
  // Reading .root file and getting histos
  //
  TFile* file = new TFile("../rootfiles/data.root");

  TH1F* types = (TH1F*)file->Get("Particle_types");
  TH1F* phi_hist = (TH1F*)file->Get("phi_angle");
  TH1F* theta_hist = (TH1F*)file->Get("theta_angle");
  TH1F* p_hist = (TH1F*)file->Get("p_hist");
  TH1F* trasvP_hist = (TH1F*)file->Get("trasvP_hist");
  TH1F* energy_histo = (TH1F*)file->Get("energy_histo");
  TH1F* tot_inv = (TH1F*)file->Get("tot_inv");
  TH1F* decay_inv = (TH1F*)file->Get("decay_inv");
  TH1F* sameCharge_inv = (TH1F*)file->Get("sameCharge_inv");
  TH1F* diffCharge_inv = (TH1F*)file->Get("diffCharge_inv");
  TH1F* piK_inv = (TH1F*)file->Get("Pione_Kaone_diff");
  TH1F* Kpi_inv = (TH1F*)file->Get("Kaone_Pione_same");

  std::cout << std::endl;
```

```
//
// Printing histos properties
//

types->Print();
phi_hist->Print();
theta_hist->Print();
p_hist->Print();
trasvP_hist->Print();
energy_histo->Print();
tot_inv->Print();
decay_inv->Print();
sameCharge_inv->Print();
diffCharge_inv->Print();
piK_inv->Print();
Kpi_inv->Print();

std::cout << std::endl;

//
// Particle types generation
//
std::cout << std::setw(60)
          << "Checking concretness of particles generation...\n\n";

std::cout << std::setw(22) << "|Particle type|" << std::setw(21)
          << "Particles generated|" << std::setw(21)
          << "Particles expected|\n";

std::cout << std::setw(8) << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(1) << '|' << std::setw(20)
          << std::left << types->GetBinContent(1) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.4 << " +/- " << std::setw(7)
          << std::left << std::sqrt(types->GetEntries() * 0.4) << '|'
          << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(2) << '|' << std::setw(20)
          << std::left << types->GetBinContent(2) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.4 << " +/- " << std::setw(7)
```

```
          << std::left << std::sqrt(types->GetEntries() * 0.4) << '|'
          << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(3) << '|' << std::setw(20)
          << std::left << types->GetBinContent(3) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.045 << " +/- "
          << std::setw(7) << std::left
          << std::sqrt(types->GetEntries() * 0.045) << '|' << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(4) << '|' << std::setw(20)
          << std::left << types->GetBinContent(4) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.045 << " +/- "
          << std::setw(7) << std::left
          << std::sqrt(types->GetEntries() * 0.045) << '|' << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(5) << '|' << std::setw(20)
          << std::left << types->GetBinContent(5) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.05 << " +/- "
          << std::setw(7) << std::left
          << std::sqrt(types->GetEntries() * 0.05) << '|' << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(6) << '|' << std::setw(20)
          << std::left << types->GetBinContent(6) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.05 << " +/- "
          << std::setw(7) << std::left
          << std::sqrt(types->GetEntries() * 0.05) << '|' << std::endl;

std::cout << std::setw(8) << std::right << '|' << std::setw(13) << std::left
          << types->GetXaxis()->GetBinLabel(7) << '|' << std::setw(20)
          << std::left << types->GetBinContent(7) << '|' << std::setw(8)
          << std::left << types->GetEntries() * 0.01 << " +/- "
          << std::setw(7) << std::left
          << std::sqrt(types->GetEntries() * 0.01) << '|' << std::endl;

std::cout << std::endl;

//
```

```
// Operations on histos and fittings
//
std::cout << std::setfill('-') << std::setw(60) << std::right
          << "FITTING HISTOGRAMS AND CHECKING MONTECARLO GEN."
          << std::setw(30) << '\n';

// Canva 1: Angles
TCanvas* canva1 = new TCanvas("canva1", "Angles");
canva1->Divide(1, 2);

canva1->cd(1);
phi_hist->Fit("pol0");
TF1* f_phi = phi_hist->GetFunction("pol0");
f_phi->SetLineColor(kBlack);
phi_hist->Draw();
f_phi->Draw("SAME");

std::cout << std::endl;

canva1->cd(2);
theta_hist->Fit("pol0");
TF1* f_theta = theta_hist->GetFunction("pol0");
f_theta->SetLineColor(kBlack);
theta_hist->Draw();
f_theta->Draw("SAME");

std::cout << std::endl;

// Canva 2: Momentum
TCanvas* canva2 = new TCanvas("canva2", "Momentum");
p_hist->Fit("expo");
TF1* f_mom = p_hist->GetFunction("expo");
f_mom->SetLineColor(kRed);
p_hist->SetFillColor(kGreen + 1);
p_hist->Draw();
f_mom->Draw("SAME");

//
// Operations with histos
//
TH1F* diff_same = new TH1F("diff_same", "Different charge minus same charge",
```

```
                              1500, 0., 10.);
diff_same->Add(diffCharge_inv, sameCharge_inv, 1, -1);
diff_same->SetLineColor(kAzure + 1);
diff_same->SetMarkerColor(kAzure + 1);

TF1* zero = new TF1("zero", "0", 0., 10.);
zero->SetLineColor(kGreen + 2);
zero->SetLineStyle(9);
zero->SetLineWidth(2);

TH1F* pion_kaon = new TH1F("pion_kaon", "Pions and Kaons", 1500, 0., 10.);
pion_kaon->Add(piK_inv, Kpi_inv, 1, -1);
pion_kaon->SetLineColor(kAzure + 1);
pion_kaon->SetMarkerColor(kAzure + 1);
pion_kaon->SetAxisRange(0.2, 2., "X");

TCanvas* canva3 = new TCanvas("canva3", "All particles");
diff_same->Fit("gaus", "", "", 0.6, 1.2);
TF1* func = diff_same->GetFunction("gaus");
func->SetLineColor(kRed);
TLegend* leg3 = new TLegend(0.1, 0.75, 0.3, 0.9);
leg3->SetHeader("Legend", "C");
leg3->AddEntry(func, "Fit", "L");
leg3->AddEntry(diff_same, "Generated data", "L");
leg3->AddEntry(zero, "Zero line", "L");
diff_same->Draw();
leg3->Draw("SAME");
func->Draw("SAME");
zero->Draw("SAME");
gPad->Update();
TPaveStats* st3 = (TPaveStats*)diff_same->FindObject("stats");
st3->SetY1NDC(0.6);
st3->SetOptStat(0);

TCanvas* canva4 = new TCanvas("canva4", "All particles");
TH1F* diff_same_ranged = new TH1F(*diff_same);
diff_same_ranged->SetAxisRange(0.7, 1.1, "X");
diff_same_ranged->SetAxisRange(-3000, 9000, "Y");
diff_same_ranged->SetMarkerStyle(kFullCircle);
diff_same_ranged->SetMarkerColor(kAzure + 1);
diff_same_ranged->Fit("gaus", "", "", 0.6, 1.2);
```

```
TF1* fitFunc = diff_same_ranged->GetFunction("gaus");
fitFunc->SetLineColor(kRed);
TLegend* leg4 = new TLegend(0.1, 0.75, 0.3, 0.9);
leg4->SetHeader("Legend", "C");
leg4->AddEntry(fitFunc, "Fit", "L");
leg4->AddEntry(diff_same_ranged, "Generated data", "L");
leg4->AddEntry(zero, "Zero line", "L");
diff_same_ranged->Draw();
leg4->Draw("SAME");
fitFunc->Draw("SAME");
zero->Draw("SAME");
gPad->Update();
TPaveStats* st4 = (TPaveStats*)diff_same_ranged->FindObject("stats");
st4->SetOptStat(0);
st4->SetY1NDC(0.6);

std::cout << std::endl;

TCanvas* canva5 = new TCanvas("canva5", "Pions and Kaons");
pion_kaon->Fit("gaus", "", "", 0.6, 1.4);
TF1* funky = pion_kaon->GetFunction("gaus");
funky->SetLineColor(kRed);
TLegend* leg5 = new TLegend(0.1, 0.75, 0.3, 0.9);
leg5->SetHeader("Legend", "C");
leg5->AddEntry(funky, "Fit", "L");
leg5->AddEntry(pion_kaon, "Generated data", "L");
leg5->AddEntry(zero, "Zero line", "L");
pion_kaon->Draw();
funky->Draw("SAME");
zero->Draw("SAME");
leg5->Draw("SAME");
gPad->Update();
TPaveStats* st5 = (TPaveStats*)pion_kaon->FindObject("stats");
st5->SetOptStat(0);
st5->SetY1NDC(0.6);

std::cout << std::endl;

TCanvas* canva6 = new TCanvas("canva6", "Pions and Kaons");
TH1F* pion_kaon_ranged = new TH1F(*pion_kaon);
pion_kaon_ranged->SetAxisRange(0.6, 1.4, "X");
```

```
  pion_kaon_ranged->SetAxisRange(-2000, 9000, "Y");
  pion_kaon_ranged->SetLineColor(kAzure + 1);
  pion_kaon_ranged->SetMarkerColor(kAzure + 1);
  pion_kaon_ranged->Fit("gaus", "", "", 0.6, 1.4);
  TF1* fitFunky = pion_kaon_ranged->GetFunction("gaus");
  fitFunky->SetLineColor(kRed);
  TLegend* leg6 = new TLegend(0.1, 0.75, 0.3, 0.9);
  leg6->SetHeader("Legend", "C");
  leg6->AddEntry(fitFunky, "Fit", "L");
  leg6->AddEntry(pion_kaon_ranged, "Generated data", "L");
  leg6->AddEntry(zero, "Zero line", "L");
  pion_kaon_ranged->Draw();
  fitFunky->Draw("SAME");
  zero->Draw("SAME");
  leg6->Draw("SAME");
  gPad->Update();
  TPaveStats* st6 = (TPaveStats*)pion_kaon_ranged->FindObject("stats");
  st6->SetOptStat(0);
  st6->SetY1NDC(0.6);

  //
  // Printing canvas
  //
  canva1->Print("analysis_pdfs/angles_fit.pdf");
  canva2->Print("analysis_pdfs/momentum_fit.pdf");
  canva3->Print("analysis_pdfs/all_fit.pdf");
  canva4->Print("analysis_pdfs/all_particular.pdf");
  canva5->Print("analysis_pdfs/pk_fit.pdf");
  canva6->Print("analysis_pdfs/pk_particular.pdf");

  TCanvas* canva7 = new TCanvas();
  decay_inv->Fit("gaus");
  canva7->Print("analysis_pdfs/decay_fit.pdf");
}
```