Simulazione delle collisioni di particelle elementari

DIFA UniBo - Laboratorio di Elettromagnetismo e Ottica - Modulo 3

Simone Pasquini

28 dicembre 2022

Indice

1 Introduzione

Tale programma ha lo scopo di simulare e analizzare collisioni tra particelle elementari. Il progetto è implementato seguendo le principali *coding conventions* del linguaggio **C++**, con particolare attenzione rivolta al modello della programmazione a oggetti, che include sia il polimorfismo dinamico sia l'ereditarietà virtuale.
ROOT è il framework di supporto utilizzato per la gestione dei dati raccolti; tale software, scritto in linguaggio C++, consente di gestire molteplici tecniche di *generazione Monte Carlo* e metodi di analisi dati. In questo caso, si è preferito l'uso della modalità compilata di ROOT (e non interpretata), per garantire maggiore efficienza e miglior controllo del codice prodotto.
Le funzionalità suddette di ROOT, dunque, consentono di intuire il principale obiettivo del progetto: generare eventi di collisione tra particelle secondo proporzioni ben definite e verificare la correttezza complessiva della simulazione.
Dalla generazione di eventi, si può estrarre un segnale particolarmente interessante emesso da una particella instabile, chiamata **K***, descritta più dettagliatamente nel seguito; pertanto, il fine ultimo dell'analisi, è l'estrazione e lo studio di suddetto segnale, grazie al quale si possono misurare sia la **massa invariante** di K* stessa, sia la sua **larghezza di risonanza**.

I tipi di particelle scelti per la simulazione sono **pioni** (π^+, π^-), **kaoni** $\left(\mathrm{K}^+, \mathrm{K}^-\right)$, **protoni** ($\mathrm{p}^+$, p^-) - sono coppie di particelle e rispettive antiparticelle, di segno opposto - e la particella di risonanza **K***[1].
Mentre i primi tre tipi sono stabili, il quarto è instabile e decade in modo equiprobabile in due prodotti di decadimento: $\tau^+\mathrm{K}^-$ e $\tau^-\mathrm{K}^+$. Dai prodotti finali di decadimento $\tau\mathrm{K}$ è possibile ricostruire la massa invariante di K*, ossia la sua massa a riposo.

Naturalmente, la simulazione tiene conto della conservazione relativistica dell'energia, come verrà esposto successivamente.

2 Struttura del codice

Il programma è suddiviso in vari header file e file di implementazione, sia per gestire al meglio i blocchi di base necessari per il corretto funzionamento della simulazione, sia per garantire modularità nello sviluppo di codice. Nello specifico, il programma è ripartito nelle seguenti classi:

- **particleType:** definisce come attributi privati le tre proprietà di base delle particelle simulate: nome, massa e carica; queste ultime sono immutabili, infatti sono dichiarate `const`, così come tutti gli attributi che non variano durante l'esecuzione del programma.
  Nella parte pubblica, vi sono i metodi che servono a ottenere o stampare a schermo le proprietà suddette, come il metodo `Print()`, e naturalmente un costruttore parametrico.
  Da notare è l'uso della keyword `virtual`, che permette alla funzione dichiarata in

[1] La risonanza individua un picco caratteristico delle particelle instabili, come la K*, descritto dalla larghezza di risonanza Γ, legata alla vita media τ della particella tramite la relazione: $\Gamma = \frac{h}{2\pi\tau}$, ove $h = 6.626 \cdot 10^{-34}$ Js è la costante di Planck.

tal modo di essere ridefinita in eventuali classi derivate, e fa si che vi sia un forte legame dinamico tra funzione e oggetto che la invoca; ciò è proprio alla base del *polimorfismo al runtime*.

- **resonanceType:** definisce le stesse proprietà di `particleType`, con l'aggiunta della "Width" (la larghezza di risonanza tipica delle particelle instabili).
  Visto che `resonanceType` è un tipo "specializzato" di `particleType` secondo una relazione "*is a*"[2], la prima è implementata come classe derivata della seconda seguendo il principio di reimpiego del codice per ereditarietà: quest'ultimo permette di risparmiare la scrittura di codice aggiuntivo, visto che la "classe figlia" è in grado di importare le funzioni membro pubbliche senza la necessità di scriverne di nuove (eventualmente si possono ridefinire, come avviene in questo caso). Come anticipato, i metodi di `resonanceType` sono tutti ridefiniti a partire da quelli implementati nella classe base, per mezzo della keyword `override`.

- **particle:** descrive le proprietà di base sopra elencate e le caratteristiche cinematiche delle particelle, infatti tra gli attributi vi sono le tre componenti cartesiane di un impulso (le collisioni avvengono in uno spazio tridimensionale).
  Dato che nella simulazione è elevato il numero delle particelle con cui si lavora, ma il numero dei tipi è limitato, si è scelto di implementare `particle` reimpiegando il codice per composizione e non per ereditarietà, come avvenuto per le classi sopra descritte: ciò è più vantaggioso soprattutto in termini di memoria, in quanto se si utilizzasse l'ereditarietà le proprietà di base delle particelle si ripeterebbero per ogni oggetto particela creato e, visto che ne devono essere generate molte, ciò non è decisamente conveniente.
  Pertanto, `particle` al suo interno include un array (della *STD library*, per avere maggiore sicurezza nella gestione delle risorse di memoria) di puntatori[3] a oggetti `ParticleType` (o `ParticleType`), implementato come membro privato statico, chiamato `fParticleType`, grazie al quale si fa corrispondere a ogni tipo di particella una posizione all'interno dell'array[4].
  Nella classe `particle` vi sono metodi che permettono di aggiungere, stampare o manipolare i tipi di particelle con i rispettivi dati. Alcuni di questi sono privati, come `FindParticle`, indispensabile per l'implementazione di tutte le funzioni membro che forniscono o modificano le caratteristiche dei tipi di particelle riconducendosi all'array statico. Da notare è il metodo statico `AddParticleType`, che permette di aggiungere sequenzialmente dei tipi di particella all'array suddetto, attraverso l'allocazione dinamica di memoria sullo *heap*.
  I metodi `Decay2Body` e `Boost` gestiscono rispettivamente il decadimento della particella madre K* nelle figlie pione e kaone, e garantiscono una corretta applicazione delle trasformazioni di Lorentz nel sistema di riferimento in cui ogni particella è ferma (tenendo conto del contributo relativistico degli eventi).
  Gli altri metodi di `particle` permettono di manipolare vari dati, tra cui le componenti della quantità di moto delle particelle; da notare sono sia i *"Setters"* (come l'overload del metodo `SetIndex` che permette di scegliere l'indice di `fParti-`

[2] In altre parole, `resonanceType` è un `particleType` senza l'attributo Width.

[3] Per rendere più efficienti l'ereditarietà virtuale e il polimorfismo al runtime si è lavorato con puntatori e non con istanze.

[4] In sostanza, `fParticleType` funge da "tabella dei tipi di particelle", con cui si fa corrispondere a un indice dell'array uno dei 7 tipi generati.

`cleType` una volta che gli si passa il nome di una particella o la posizione della stessa all'interno del medesimo array statico) sia i *"Getters"*, utili a ottenere dati importanti, come l'energia totale (attraverso `GetEnergy`) o la massa invariante tra due particelle che decadono (per mezzo di `GetInvMass`).

- **test.cpp:** definisce i test sviluppati secondo il modello *unit testing* di DOCTEST. Questa è una parte essenziale del programma, in quanto permette di verificare sia se le funzioni sviluppate svolgono correttamente il loro compito, sia se vi sono problemi nella gestione della memoria[5], una possibilità non remota vista la presenza di puntatori a oggetti.

3 Generazione

La generazione Monte Carlo delle variabili della simulazione avviene attraverso la classe *TRandom* di ROOT, che consente di estrarre singolarmente variabili secondo delle distribuzioni predefinite (uniforme, gaussiana, esponenziale, etc.). Durante il corso della generazione, tramite il metodo `Fill` di ROOT, un array di supporto e due cicli `for` annidati, tutte le variabili generate vengono inserite in vari istogrammi, salvati in un *ROOT File* che consentirà l'analisi dei dati in un momento successivo.

La generazione avviene all'interno del file **simulation.cpp**. Si è scelto di simulare 10^5 eventi, in ciascuno dei quali si generano casualmente 100 particelle secondo proporzioni ben definite: sul totale di 10^7 particelle generate, π^+ e π^- corrispondono all'80%, K^+ e K^- al 10%, p^+ e p^- al 9% e la K*, l'unica particella instabile, al rimanente 1%.

Per ogni particella, si generano le direzioni dell'angolo azimutale θ e polare φ attraverso due distribuzioni uniformi (affinché vi sia omogeneità e isotropia nella direzione del moto delle particelle[6]) con dominio rispettivamente $[0, 2\pi]$ e $[0, \pi]$.

Il modulo dell'impulso tridimensionale di ciascuna particella è generato da una distribuzione esponenziale di media 1; dato che l'impulso, fin qui, è rappresentato tramite coordinate sferiche, si è optato per un cambiamento di coordinate da sferiche a cartesiane. Dopo il passaggio di coordinate, si è proceduto con il riempimento di altri tre istogrammi, il primo contenente il modulo dell'impulso, il secondo il modulo dell'impulso trasverso e il terzo la distribuzione di energia delle particelle[7].

Ogni particella di risonanza K* decade o in una coppia τ^+K^- o in τ^-K^+ con probabilità rispettivamente del 50% e 50%. Come già anticipato, dai prodotti di decadimento τK si può ricostruire la massa invariante di K*. Affinché ciò avvenga, si sono creati cinque istogrammi di massa invariante: il primo include la massa invariante calcolata tra tutte le particelle (anche i prodotti di decadimento) di carica discorde, il secondo contiene la

[5] Eventuali memory leaks sono stati ricercati in fase di debugging aggiungendo le opzioni di compilazione -fsanitize=address, -Wall e -Wextra.

[6] Nessuna direzione delle particelle è privilegiata, al momento della generazione.

[7] L'energia E delle particelle è data formalmente da $E = \sqrt{m^2 + \left\|\vec{P}\right\|^2}$, con m la massa di una particella e $\vec{P}$ il vettore quantità di moto.

massa invariante calcolata tra tutte le particelle (inclusi i prodotti di decadimento) di carica concorde, il terzo raccoglie le masse invarianti tra τ^+K^- τ^-K^+, il quarto quelle tra τ^+K^+ τ^-K^-, e il quinto contenente un istogramma "di benchmark", con le masse invarianti tra i prodotti generati dal decadimento della K*; quest'ultimo, in particolare, serve a verificare che la gestione di decadimento, le formule di massa invariante ed energia e le classi sopra definite siano state implementate correttamente.

Tutti gli istogrammi sopra presentati, vengono illustrati nella successiva sezione di analisi.

4 Analisi

Aprendo in modalità lettura il *ROOT File* contenente i dati della simulazione, si può procedere con l'analisi dei 12 istogrammi salvati, avvenuta nella macro indipendente **analysis.cpp**, al fine di separare in modo netto la parte di generazione con quella di analisi.
I fit intercorsi nella sezione di analisi avvengono tutti attraverso la definizione di funzioni di classe `TF1`, a partire da funzioni predefinite di ROOT.

Analizzando l'istogramma dei tipi di particelle generati (Figura 1) si può estrarre il contenuto di ogni bin e confrontare le occorrenze effettivamente osservate con quelle attese (queste ultime ricavate dalla generazione secondo proporzioni definite). I risultati, dotati di una buona corrispondenza, sono presentati in Tabella 1.

Abbondanze delle particelle		
Specie	Occorrenze osservate	Occorrenze attese
π^+	$(3999.5 \pm 2.0) \cdot 10^3$	$4 \cdot 10^6$
π^-	$(4001.9 \pm 2.0) \cdot 10^3$	$4 \cdot 10^6$
K^+	$(4993.7 \pm 7.1) \cdot 10^2$	$5 \cdot 10^5$
K^-	$(5001.6 \pm 7.1) \cdot 10^2$	$5 \cdot 10^5$
p^+	$(4490.2 \pm 6.7) \cdot 10^2$	$4.5 \cdot 10^5$
p^-	$(4496.7 \pm 6.7) \cdot 10^2$	$4.5 \cdot 10^5$
K*	$(1004.0 \pm 3.2) \cdot 10^2$	10^5

Tabella 1: Occorrenze delle particelle generati e attese, con relativi errori, su 10^7 eventi generati.

Nella Tabella 2, si può verificare numericamente che le distribuzioni degli angoli azimutali e polari (i cui istogrammi si trovano nella parte rimanente della Figura 1), così come la distribuzione esponenziale del modulo dell'impulso (Figura 2), sono coerenti con quanto simulato in fase di generazione, visto che il rapporto χ^2/DOF è molto vicino all'unità.

Distribuzione angoli azimutale (θ) e polare (φ), modulo dell'impulso					
Distribuzione	Parametri del fit	χ^2	χ^2 prob.	DOF	χ^2/DOF
Fit a distribuzione angolo θ	Const: 9999.0 ± 3.2	1048	0.14	999	1.0
Fit a distribuzione angolo φ	9999.1 ± 3.2	888.6	0.99	999	0.89
Fit a distribuzione modulo impulso	Const: 12.2061 ± 0.0004	461.7	0.8769	498	0.927014
	Tau, [c/GeV]: -1.0000 ± 0.0003				

Tabella 2: Parametri di fit delle distribuzioni di θ e φ e del modulo dell'impulso, con relativi errori. Per le distribuzioni degli angoli si è usata la funzione predefinita di ROOT "pol0" ($f(x) = \text{Const}$), mentre per il modulo dell'impulso la funzione "expo" $\left(f(x) = e^{\text{Const}+\text{Tau}\cdot x}\right)$, dove il parametro "Tau" è l'antireciproco della media della distribuzione esponenziale.

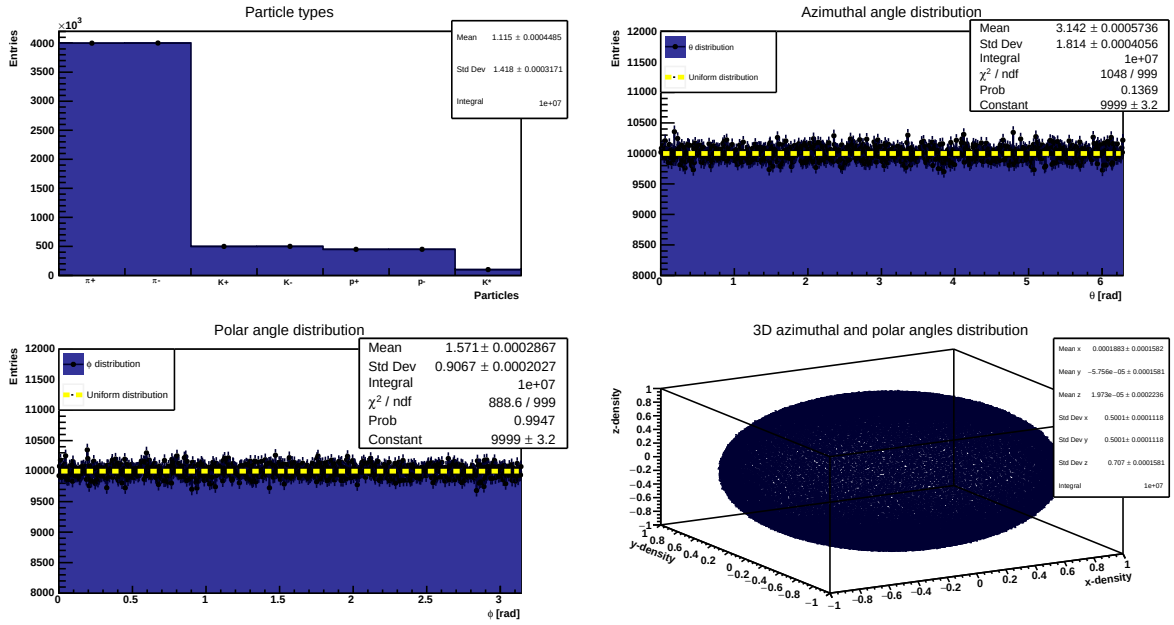

Figura 1: Da sinistra a destra: in alto, i tipi di particelle generate e distribuzione dell'angolo azimutale e relativo fit uniforme, in basso, distribuzione dell'angolo polare e relativo fit uniforme e distribuzione tridimensionale degli angoli azimutale e polare, per verificare visivamente l'indipendenza tra i due angoli, garantendo omogeneità e isotropia delle direzioni delle particelle.

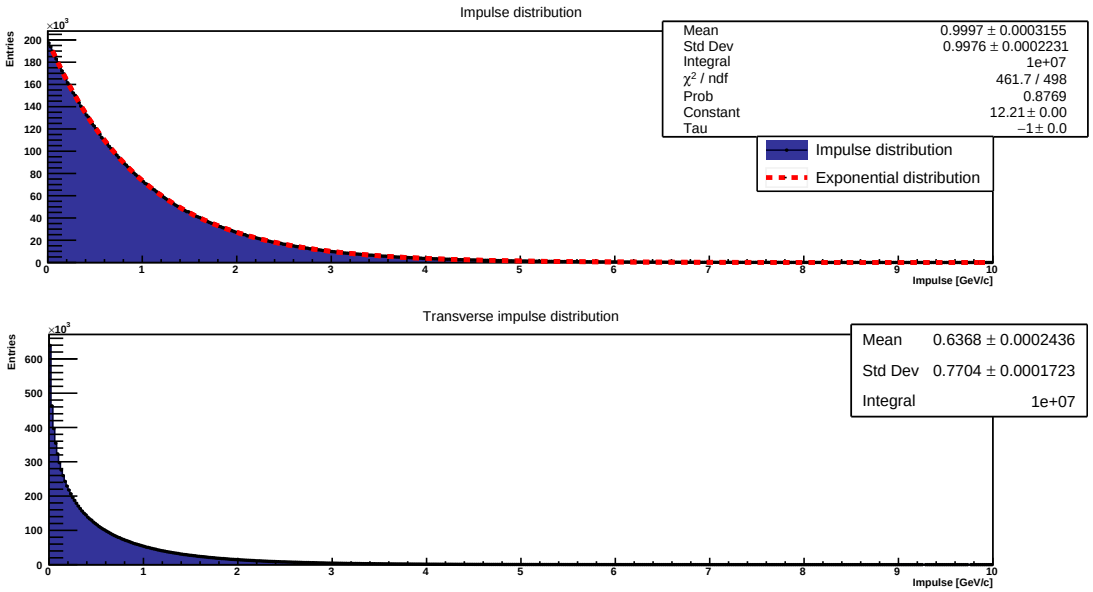

Figura 2: In alto la distribuzione del modulo dell'impulso e relativo fit esponenziale, in basso la distribuzione del modulo dell'impulso trasverso.

Il segnale di risonanza della K* è piuttosto debole, pertanto non si può estrarre con facilità senza i contributi dei suoi prodotti di decadimento e delle particelle di carica discorde (particella e relativa antiparticella).

Il picco di risonanza di K* è sommerso dal fondo delle combinazioni accidentali, eppure, manipolando gli istogrammi tramite il metodo `Add` di ROOT, agendo in modo da operare una differenza di istogrammi, si possono ottenere due istogrammi da cui ricavare la massa e la larghezza di risonanza di K*.
In pratica, per isolare il segnale ed estrarre i parametri di massa e larghezza di risonanza di K*, si devono considerare gli **istogrammi di massa invariante** della fase di generazione, agendo in questo modo:

- sottrarre gli istogrammi contenenti le coppie di particelle con carica opposta con particelle dotate di stessa carica (Figura 4);
- sottrarre gli istogrammi contenenti solo coppie di particelle τK *decadute* con carica opposta con particelle τK *decadute* dotate di stessa carica (Figura 5).

Il rapporto χ^2/DOF è migliore nella seconda differenza (si veda la Tabella 3).
Si assume una distribuzione gaussiana per il segnale della K*[8]; dalle Figura 4 e Figura 5 si possono ricavare la media e la sigma della distribuzione, che corrispondono rispettivamente alla massa e alla larghezza di risonanza di K*.

I risultati dei due fit sopra descritti vengono confrontati con l'istogramma di benchmark in Figura 3, contenente l'istogramma della "vera" massa invariante tra i prodotti del decadimento di K*. I risultati, riassunti nella Tabella 3, mostrano un buon accordo, entro gli errori, delle masse di risonanza di K*; osservando la differenza di istogrammi ottenuta dalle combinazioni di tutte le particelle, si nota una discrepanza nel dato della larghezza di risonanza rispetto al dato atteso. Tale divergenza rende il rapporto χ^2/DOF lievemente migliore nelle combinazioni di massa invariante dei soli prodotti di decadimento τK.

[8] In realtà questa è un'approssimazione (accettabile) della vera distribuzione di risonanza della K*, meglio rappresentata dalla distribuzione di Breit-Wigner.

Analisi della K*					
Distribuzione fit	Media ($\bar{m}$) $\left[\mathrm{GeV/c^2}\right]$	Sigma (σ) $\left[\mathrm{GeV/c^2}\right]$	Ampiezza (A)	χ^2 prob.	χ^2/DOF
Massa invariante vere K*	0.8916± 0.0002	0.0503± 0.0001	$(12.0 \pm 4.6)\cdot 10^3$	0.77	0.79
Massa invariante ottenuta da differenza delle combinazioni di carica discorde e concorde	0.893± 0.022	0.088± 0.024	$(9.2 \pm 2.0)\cdot 10^3$	0.91	0.67
Massa invariante ottenuta da differenza delle combinazioni πK decadute di carica discorde e concorde	0.8937± 0.0064	0.0533± 0.0075	$(67.1 \pm 7.4)\cdot 10^2$	0.83	0.74

Tabella 3: Masse Invarianti ottenute attraverso combinazioni differenti (indicate in tabella) di differenze di istogrammi. I parametri, e i relativi errori, derivano da un fit gaussiano secondo la funzione $\left(f(x) = A \cdot e^{-0.5\cdot\left(\frac{x-\bar{m}}{\sigma}\right)^2}\right)$.

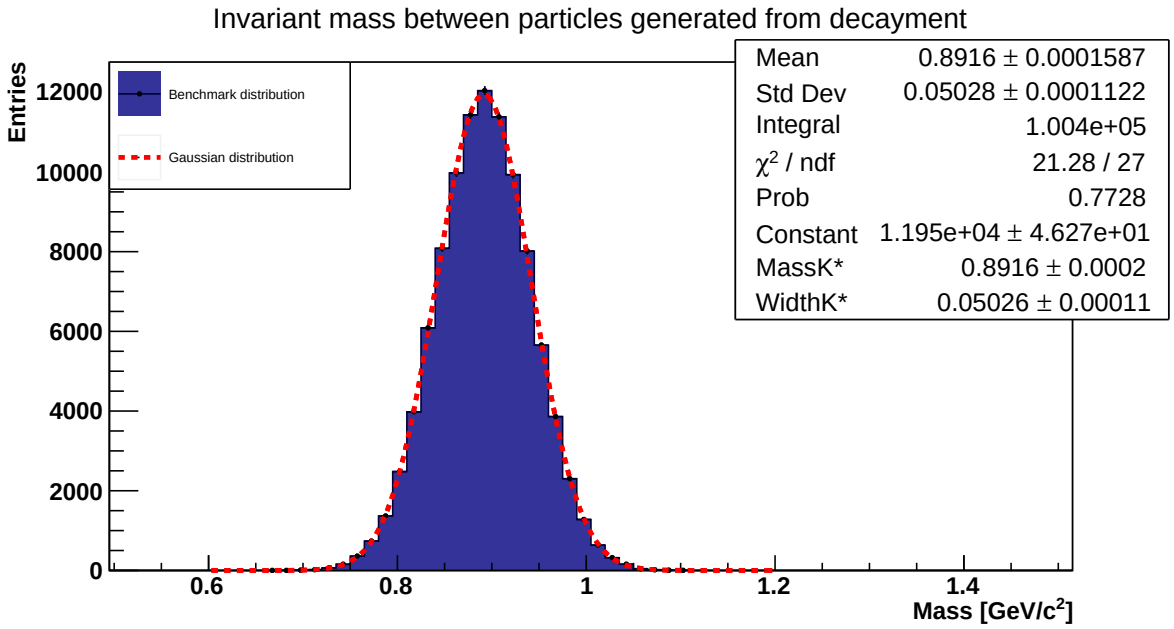

Figura 3: Istogramma di benchmark: massa invariante tra i prodotti generati dal decadimento della K* e relativo fit gaussiano.

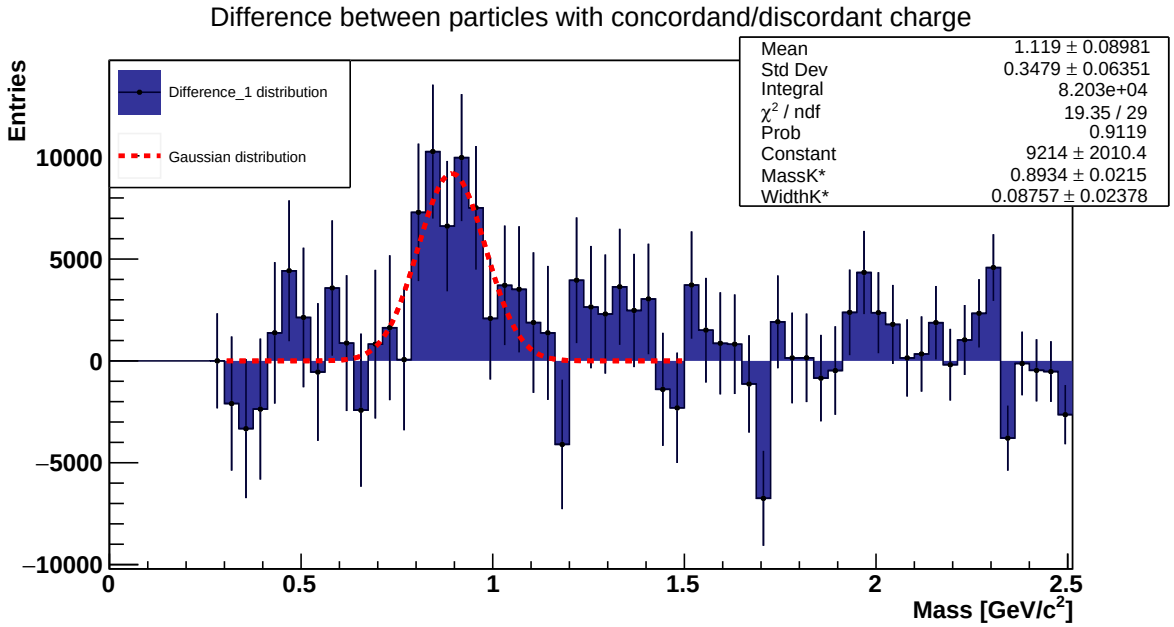

Figura 4: Differenza tra tutte le particelle con carica concorde e discorde e relativo fit gaussiano.

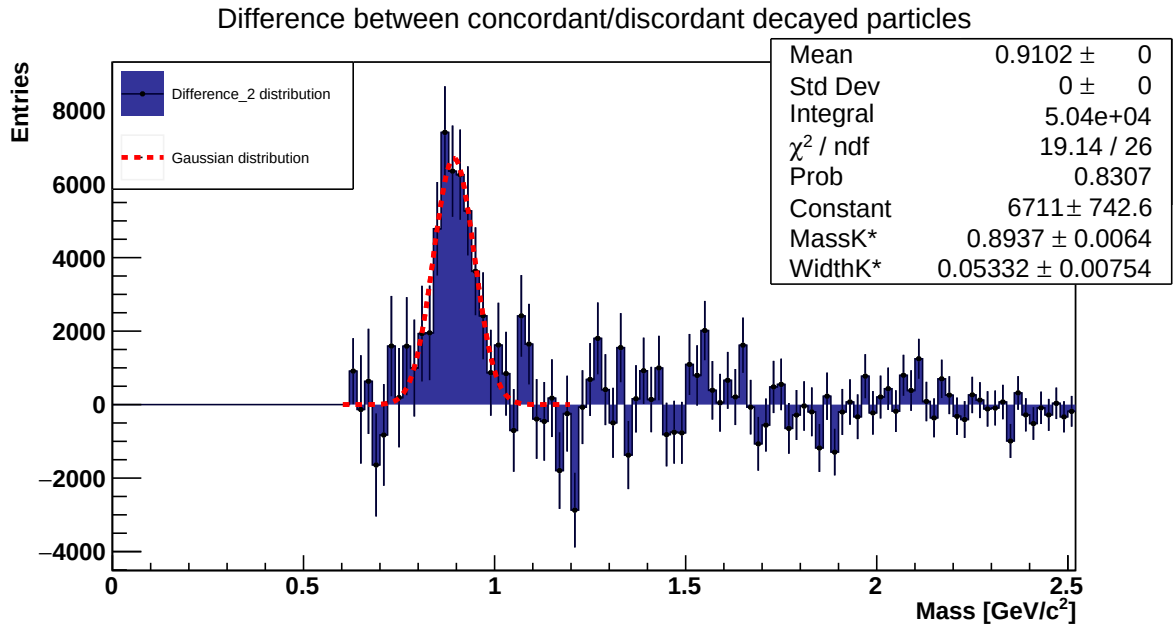

Figura 5: Differenza tra le coppie τK con carica concorde e discorde e relativo fit gaussiano.

4.1 Grafici suppletivi

Si propongono di seguito i grafici citati nella simulazione ma non utilizzati nella sezione di analisi.

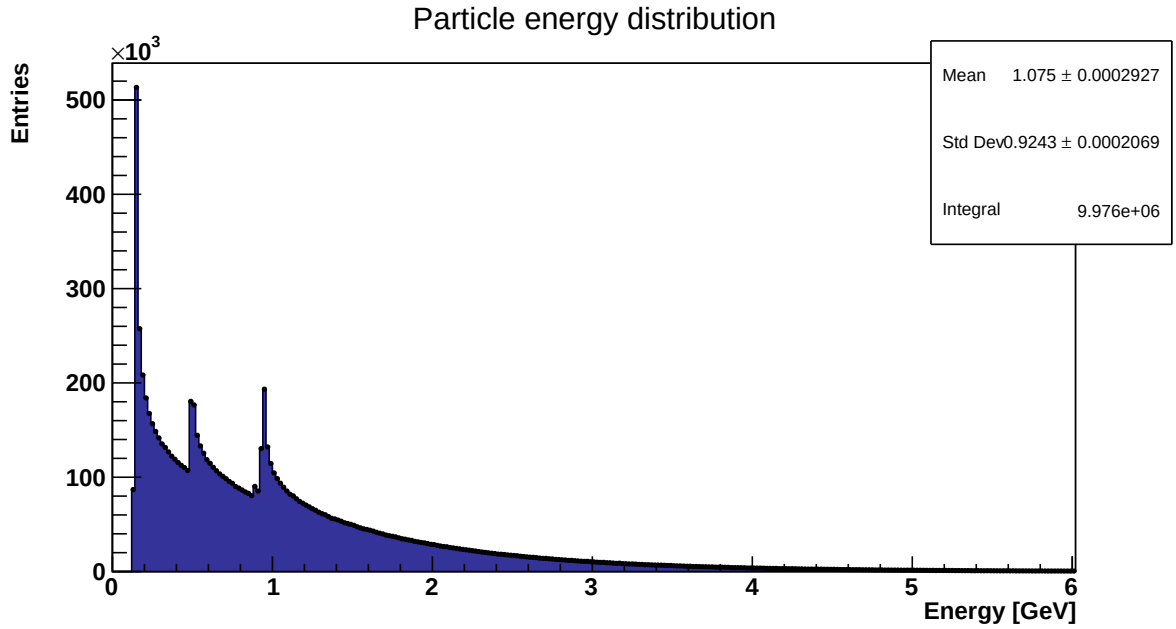

Figura 6: Distribuzione dell'energia delle particelle.

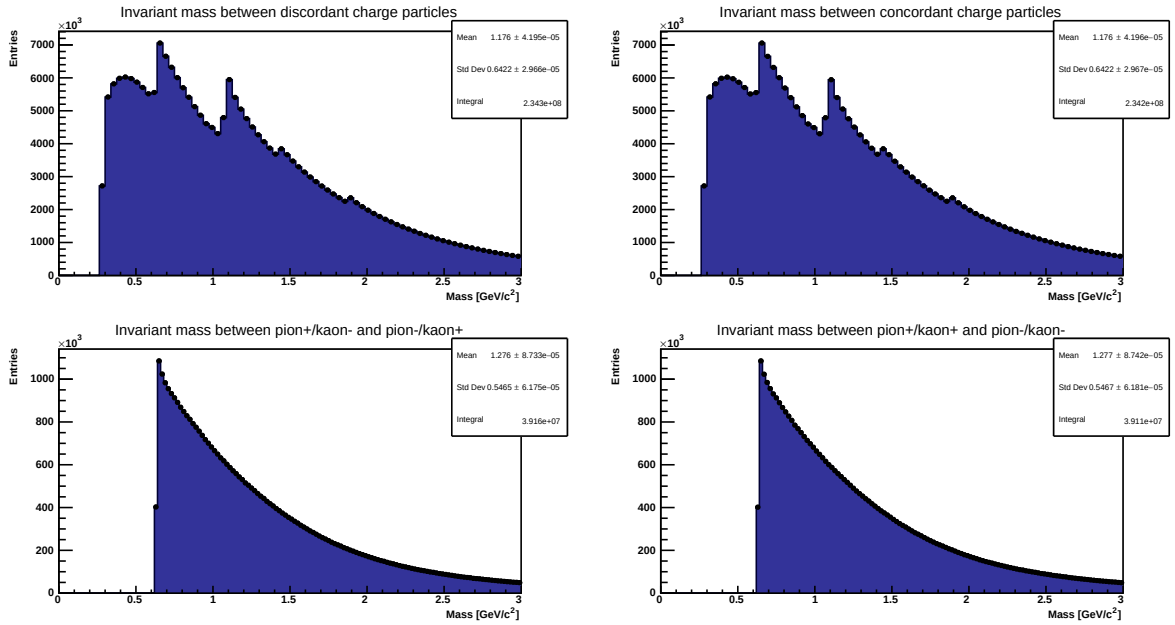

Figura 7: Combinazioni (indicate nei titoli dei grafici) delle masse invarianti tra particelle concordi, discordi e prodotti di decadimento τK.

5 Appendice

Si presenta di seguito il codice delle classi (con relativi header file e file di implementazione), dei programmi di generazione e analisi.

Code 1: Header file di Particle Type.

```
#ifndef PARTICLETYPE_HPP
#define PARTICLETYPE_HPP

#include <iomanip>
#include <iostream>

class ParticleType {
 public:
  ParticleType(std::string const& name, double mass, int charge);

  // Getters for member data

  std::string const GetName() const;

  double GetMass() const;

  int GetCharge() const;

  virtual double GetWidth() const;

  // Printer for all data
  virtual void Print() const;
```

```
23
24  private:
25   std::string const fName;
26   double const fMass;
27   int const fCharge;
28 };
29
30 #endif
```

Code 2: File di implementazione di Particle Type.

```
1  #include "particleType.hpp"
2
3  ParticleType::ParticleType(std::string const& name, double mass,
       int charge)
4      : fName{name}, fMass{mass}, fCharge{charge} {}
5
6  std::string const ParticleType::GetName() const { return fName; }
7
8  double ParticleType::GetMass() const { return fMass; }
9
10 int ParticleType::GetCharge() const { return fCharge; }
11
12 double ParticleType::GetWidth() const { return 0; }
13
14 void ParticleType::Print() const {
15   using namespace std;
16
17   // Printing data with same spacing
18   cout << left << setw(9) << "\nName:" << fName << left << setw(9)
19        << "\nMass:" << fMass << left << setw(9) << "\nCharge:" <<
           fCharge
20        << '\n';
21 }
```

Code 3: Header file di Resonance Type.

```
1  #ifndef RESONANCETYPE_HPP
2  #define RESONANCETYPE_HPP
3  #include "particleType.hpp"
4
5  class ResonanceType : public ParticleType {
6   public:
7    ResonanceType(std::string const& name, double mass, int charge,
       double width);
8
9    double GetWidth() const override;
10   void Print() const override;
11
12  private:
13   double const fWidth;
14 };
15
```

```
#endif
```

Code 4: File di implementazione di Particle Type.

```
#include "resonanceType.hpp"

ResonanceType::ResonanceType(std::string const& name, double mass,
    int charge,
                             double width)
    : ParticleType{name, mass, charge}, fWidth{width} {}

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

void ResonanceType::Print() const {
  using namespace std;

  ParticleType::Print();
  cout << left << setw(8) << "Width:" << fWidth << '\n';
}
```

Code 5: Header file di Particle.

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include <vector>

#include "particleType.hpp"
#include "resonanceType.hpp"

class Particle {
 public:
  Particle(std::string const& name, double px = 0., double py =
     0.,
           double pz = 0.);

  // Default constructor
  Particle();

  // Method used to add particles in fParticleType
  static void AddParticleType(std::string const& name, double mass
     , int charge,
                              double width = 0.);

  void SetIndex(int index);

  void SetIndex(std::string const& name);

  void SetP(double px, double py, double pz);

  // Getters and printers

  int GetIndex() const;
```

```
30
31    static void PrintParticle();
32
33    void PrintIndex() const;
34
35    double GetPx() const;
36
37    double GetPy() const;
38
39    double GetPz() const;
40
41    double GetMass() const;
42
43    double GetEnergy() const;
44
45    double GetInvMass(Particle const& p) const;
46
47    // Used to get fParticleType's size in tests and in member
          funtions above
48    static int GetSize();
49
50    int GetCharge() const;
51
52    // Decay in two particles
53    int Decay2Body(Particle& dau1, Particle& dau2) const;
54
55  private:
56    static std::vector<ParticleType*> fParticleType;
57
58    int fIndex;
59    double fPx;
60    double fPy;
61    double fPz;
62
63    static int FindParticle(std::string const& particleName);
64
65    void Boost(double bx, double by, double bz);
66 };
67
68 #endif
```

Code 6: File di implementazione di Particle class.

```
1  #include "particle.hpp"
2
3  #include <algorithm>
4  #include <cassert>
5  #include <cmath>
6
7  Particle::Particle(std::string const& name, double px, double py,
       double pz)
8      : fPx{px}, fPy{py}, fPz{pz} {
```

```
9   auto index = FindParticle(name);
10
11  // Converting unsigned int in signed int to make comparisons
        without narrowing
12  auto size = GetSize();
13
14  if (index == size) {
15    std::cerr << "No matches found for particle \'" << name << "
          \'\n";
16  } else {
17    fIndex = index;
18  }
19
20  assert(fIndex != size);
21 }
22
23 Particle::Particle() { fIndex = -1; }
24
25 std::vector<ParticleType*> Particle::fParticleType{};
26
27 int Particle::FindParticle(std::string const& particleName) {
28  auto it = std::find_if(
29      fParticleType.begin(), fParticleType.end(),
30      [&](ParticleType* p) { return particleName == p->GetName();
          });
31
32  // Findng iterator position
33  return it - fParticleType.begin();
34 }
35
36 int Particle::GetIndex() const { return fIndex; }
37
38 void Particle::AddParticleType(std::string const& name, double
       mass, int charge,
39                                double width) {
40  auto index = FindParticle(name);
41  auto size = GetSize();
42
43  if (index != size) {
44    std::cerr << "Particle \'" << name << "\' already inserted\n";
45  } else {
46    ResonanceType* resT = new ResonanceType{name, mass, charge,
          width};
47    fParticleType.push_back(resT);
48
49    std::cout << "Inserted particle \'" << name << "\' in index "
          << index
50              << '\n';
51  }
52 }
53
```

```
54 void Particle::SetIndex(int index) {
55   auto size = GetSize();
56
57   // < -1 is used because default constructor sets fIndex to -1
58   if (index >= size || index < -1) {
59     std::cerr << "Particle not found\n";
60   } else {
61     fIndex = index;
62   }
63 }
64
65 void Particle::SetIndex(std::string const& name) {
66   SetIndex(FindParticle(name));
67 }
68
69 void Particle::PrintParticle() {
70   std::for_each(fParticleType.begin(), fParticleType.end(),
71                 [](ParticleType* p) { p->Print(); });
72 }
73
74 void Particle::PrintIndex() const {
75   using namespace std;
76
77   // Printing data with same spacing
78   cout << left << setw(10) << "\nIndex:" << fIndex << left << setw
       (10)
79        << "\nName" << fParticleType[fIndex]->GetName() << left <<
           setw(10)
80        << "\nPx:" << fPx << left << setw(10) << "\nPy:" << fPy <<
           left
81        << setw(10) << "\nPz:" << fPz << '\n';
82 }
83
84 double Particle::GetPx() const { return fPx; }
85
86 double Particle::GetPy() const { return fPy; }
87
88 double Particle::GetPz() const { return fPz; }
89
90 double Particle::GetMass() const { return fParticleType[fIndex]->
    GetMass(); }
91
92 double Particle::GetEnergy() const {
93   return sqrt(GetMass() * GetMass() + (fPx * fPx + fPy * fPy + fPz
       * fPz));
94 }
95
96 double Particle::GetInvMass(Particle const& p) const {
97   return sqrt((GetEnergy() + p.GetEnergy()) * (GetEnergy() + p.
       GetEnergy()) -
98               ((fPx + p.fPx) * (fPx + p.fPx) + (fPy + p.fPy) * (
```

```
                  fPy + p.fPy) +
99                (fPz + p.fPz) * (fPz + p.fPz)));
100 }
101
102 void Particle::SetP(double px, double py, double pz) {
103   fPx = px;
104   fPy = py;
105   fPz = pz;
106 }
107
108 int Particle::GetSize() {
109   int size = fParticleType.size();
110   return size;
111 }
112
113 int Particle::GetCharge() const { return fParticleType[fIndex]->
     GetCharge(); }
114
115 int Particle::Decay2Body(Particle& dau1, Particle& dau2) const {
116   if (GetMass() == 0.) {
117     printf("Decayment cannot be preformed if mass is zero\n");
118     return 1;
119   }
120
121   double massMot = GetMass();
122   double massDau1 = dau1.GetMass();
123   double massDau2 = dau2.GetMass();
124
125   if (fIndex > -1) {  // add width effect
126
127     // gaussian random numbers
128
129     float x1, x2, w, y1, y2;
130
131     double invnum = 1. / RAND_MAX;
132     do {
133       x1 = 2.0 * rand() * invnum - 1.0;
134       x2 = 2.0 * rand() * invnum - 1.0;
135       w = x1 * x1 + x2 * x2;
136     } while (w >= 1.0);
137
138     w = sqrt((-2.0 * log(w)) / w);
139     y1 = x1 * w;
140     y2 = x2 * w;
141
142     massMot += fParticleType[fIndex]->GetWidth() * y1;
143   }
144
145   if (massMot < massDau1 + massDau2) {
146     printf(
147         "Decayment cannot be preformed because mass is too low in
```

```
            this "
148         "channel\n");
149     return 2;
150   }
151
152   double pout =
153       sqrt(
154           (massMot * massMot - (massDau1 + massDau2) * (massDau1 +
                massDau2)) *
155           (massMot * massMot - (massDau1 - massDau2) * (massDau1 -
                massDau2))) /
156       massMot * 0.5;
157
158   double norm = 2 * M_PI / RAND_MAX;
159
160   double phi = rand() * norm;
161   double theta = rand() * norm * 0.5 - M_PI / 2.;
162   dau1.SetP(pout * sin(theta) * cos(phi), pout * sin(theta) * sin(
        phi),
163             pout * cos(theta));
164   dau2.SetP(-pout * sin(theta) * cos(phi), -pout * sin(theta) *
        sin(phi),
165             -pout * cos(theta));
166
167   double energy = sqrt(fPx * fPx + fPy * fPy + fPz * fPz + massMot
        * massMot);
168
169   double bx = fPx / energy;
170   double by = fPy / energy;
171   double bz = fPz / energy;
172
173   dau1.Boost(bx, by, bz);
174   dau2.Boost(bx, by, bz);
175
176   return 0;
177 }
178
179 void Particle::Boost(double bx, double by, double bz) {
180   double energy = GetEnergy();
181
182   // Boost this Lorentz vector
183   double b2 = bx * bx + by * by + bz * bz;
184   double gamma = 1.0 / sqrt(1.0 - b2);
185   double bp = bx * fPx + by * fPy + bz * fPz;
186   double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;
187
188   fPx += gamma2 * bp * bx + gamma * bx * energy;
189   fPy += gamma2 * bp * by + gamma * by * energy;
190   fPz += gamma2 * bp * bz + gamma * bz * energy;
191 }
```

Code 7: File di implementeazione dei test attraverso il programma DOCTEST.

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN

#include "doctest.h"
#include "particle.hpp"
#include "particleType.hpp"
#include "resonanceType.hpp"

TEST_CASE("ParticleType and ResonanceType") {
  ParticleType const pT1{"e-", 0.511, -1};
  ParticleType const pT2{"p+", 10e3, 1};

  ResonanceType const rT1{"k+", 0.423, -1, 0.7};
  ResonanceType const rT2{"k-", 0.836, 1, 0.6};

  SUBCASE("Testing methods") {
    CHECK(pT1.GetName() == ("e-"));
    CHECK(pT1.GetMass() == doctest::Approx(0.511));
    CHECK(pT1.GetCharge() == -1);
    CHECK(pT1.GetWidth() == 0);

    CHECK(pT2.GetName() == ("p+"));
    CHECK(pT2.GetMass() == doctest::Approx(10e3));
    CHECK(pT2.GetCharge() == 1);
    CHECK(pT2.GetWidth() == 0);

    CHECK(rT1.GetName() == ("k+"));
    CHECK(rT1.GetMass() == doctest::Approx(0.423));
    CHECK(rT1.GetCharge() == -1);
    CHECK(rT1.GetWidth() == doctest::Approx(0.7));

    CHECK(rT2.GetName() == ("k-"));
    CHECK(rT2.GetMass() == doctest::Approx(0.836));
    CHECK(rT2.GetCharge() == 1);
    CHECK(rT2.GetWidth() == doctest::Approx(0.6));

    // Testing output streams directly on screen
    pT1.Print();
    pT2.Print();
    rT1.Print();
    rT2.Print();
  }

  SUBCASE("Testing virtual function") {
    // const here is needed
    ParticleType const* a[2]{&pT1, &rT1};

    // Name, mass and charge are printed for ParticleType, while
       name, mass,
    // charge and width are shown for ResonanceType
    for (int i{}; i != 2; ++i) {
```

```
50        a[i]->Print();
51      }
52    }
53  }
54
55  TEST_CASE("Particle") {
56    // Separating outputs for better readability
57    std::string s{"\n****************** Line break
        ******************\n"};
58    std::cout << s << '\n';
59
60    Particle::AddParticleType("e-", 0.511, -1);
61    Particle::AddParticleType("p+", 10e3, 1);
62    Particle::AddParticleType("k+", 0.423, -1, 0.7);
63
64    // Checking error in Particle::Particle
65    Particle pTest1{"Not_There"};
66
67    // Checking right insertion
68    Particle p1{"e-", 2.3e3, -3.43e3, -6.32e3};
69    Particle p2{"p+"};
70    Particle const p3{"k+", 1e3, -1e3, 2.1e3};
71
72    CHECK(Particle::GetSize() == 3);
73
74    CHECK(p1.GetIndex() == 0);
75    CHECK(p2.GetIndex() == 1);
76    CHECK(p3.GetIndex() == 2);
77
78    // Checking error in AddParticleType
79    Particle::AddParticleType("p+", 0.466, 2, 1.8);
80    Particle::AddParticleType("k+", 0.463, 1, 0.8);
81
82    Particle::PrintParticle();
83
84    // Checking 2 errors in SetIndex
85    p1.SetIndex(4);
86    p1.SetIndex("Not_There");
87
88    CHECK_FALSE(p1.GetIndex() == 4);
89
90    std::cout << s;
91
92    p1.PrintIndex();
93    p2.PrintIndex();
94    p3.PrintIndex();
95
96    SUBCASE("Testing Set/GetIndex") {
97      p1.SetIndex(1);
98      p1.PrintIndex();
99
```

```
100     CHECK(p1.GetIndex() == 1);
101     CHECK(p2.GetIndex() == 1);
102 
103     p1.SetIndex("p+");
104     CHECK(p1.GetIndex() == 1);
105     p1.SetIndex(0);
106   }
107 
108   CHECK(p1.GetIndex() == 0);
109 
110   // Testing GetP
111   CHECK(p1.GetPx() == doctest::Approx(2.3e3));
112   CHECK(p1.GetPy() == doctest::Approx(-3.43e3));
113   CHECK(p1.GetPz() == doctest::Approx(-6.32e3));
114   CHECK(p2.GetPx() == doctest::Approx(0.));
115   CHECK(p2.GetPy() == doctest::Approx(0.));
116   CHECK(p2.GetPz() == doctest::Approx(0.));
117   CHECK(p3.GetPx() == doctest::Approx(1e3));
118   CHECK(p3.GetPy() == doctest::Approx(-1e3));
119   CHECK(p3.GetPz() == doctest::Approx(2.1e3));
120 
121   // Testing GetMass
122   CHECK(p1.GetMass() == doctest::Approx(0.511));
123   CHECK(p2.GetMass() == doctest::Approx(10e3));
124   CHECK(p3.GetMass() == doctest::Approx(0.423));
125 
126   // Testing GetCharge
127   CHECK(p1.GetCharge() == -1);
128   CHECK(p2.GetCharge() == 1);
129   CHECK(p3.GetCharge() == -1);
130 
131   // Testing GetEnergy
132   CHECK(p1.GetEnergy() == doctest::Approx(7549.66).epsilon(1.));
133   CHECK(p2.GetEnergy() == doctest::Approx(10e3).epsilon(1.));
134   CHECK(p3.GetEnergy() == doctest::Approx(2531.8));
135 
136   // Testing GetInvMass
137   CHECK(p1.GetInvMass(p2) == doctest::Approx(15842).epsilon(1.));
138   CHECK(p2.GetInvMass(p3) == doctest::Approx(12273).epsilon(1.));
139   CHECK(p3.GetInvMass(p1) == doctest::Approx(7301).epsilon(1.));
140 
141   // Testing SetP
142   p2.SetP(1e3, -4e3, -5e3);
143 
144   CHECK(p2.GetPx() == doctest::Approx(1e3));
145   CHECK(p2.GetPy() == doctest::Approx(-4e3));
146   CHECK(p2.GetPz() == doctest::Approx(-5e3));
147 }
```

Code 8: File di implementazione della generazione degli eventi.

```
1 // To compile in SHELL: "g++ resonanceType.cpp particleType.cpp
```

```
   particle.cpp
// simulation.cpp `root-config --cflags --libs`"

#include <algorithm>
#include <cstdlib>

#include "TCanvas.h"
#include "TFile.h"
#include "TH1F.h"
#include "TH3F.h"
#include "TLatex.h"
#include "TMath.h"
#include "TROOT.h"
#include "TRandom.h"
#include "TStyle.h"
#include "particle.hpp"
#include "particleType.hpp"
#include "resonanceType.hpp"

// Cosmetics
void setStyle() {
  gROOT->SetStyle("Plain");
  gStyle->SetOptStat(1122);
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(0);
}

void simulation() {
  // Cosmetics
  setStyle();

  // To avoid reloading manually if .so is present
  R__LOAD_LIBRARY(particleType_cpp.so);
  R__LOAD_LIBRARY(resonanceType_cpp.so);
  R__LOAD_LIBRARY(particle_cpp.so);

  // Generating random generator seed
  gRandom->SetSeed();

  // Creating TFile
  TFile* file = new TFile("simulation_collisions.root", "RECREATE"
     );

  // \u03C0 is the unicode escape character for pion particles
  Particle::AddParticleType("\u03C0+", 0.13957, 1);
  Particle::AddParticleType("\u03C0-", 0.13957, -1);
  Particle::AddParticleType("K+", 0.49367, 1);
  Particle::AddParticleType("K-", 0.49367, -1);
  Particle::AddParticleType("p+", 0.93827, +1);
  Particle::AddParticleType("p-", 0.93827, -1);
  Particle::AddParticleType("K*", 0.89166, 0, 0.05);
```

```
51
52  // Printing particles'info
53  Particle::PrintParticle();
54
55  // Creating histograms
56
57  TH1F* h1 = new TH1F("h1", "Particle types", 7, 0, 7);
58
59  TH1F* h2 =
60      new TH1F("h2", "Azimuthal angle distribution", 1e3, 0, 2 *
           TMath::Pi());
61  TH1F* h3 = new TH1F("h3", "Polar angle distribution", 1e3, 0,
       TMath::Pi());
62  TH1F* h4 = new TH1F("h4", "Impulse distribution", 500, 0, 10);
63  TH1F* h5 = new TH1F("h5", "Transverse impulse distribution",
       500, 0, 10);
64  TH1F* h6 = new TH1F("h6", "Particle energy distribution", 500,
       0, 10);
65  TH1F* h7 = new TH1F(
66      "h7", "Invariant mass between discordant charge particles",
           80, 0, 3);
67  TH1F* h8 = new TH1F(
68      "h8", "Invariant mass between concordant charge particles",
           80, 0, 3);
69  TH1F* h9 = new TH1F(
70      "h9", "Invariant mass between pion+/kaon- and pion-/kaon+",
           150, 0, 3);
71  TH1F* h10 = new TH1F(
72      "h10", "Invariant mass between pion+/kaon+ and pion-/kaon-",
            150, 0, 3);
73  TH1F* h11 = new TH1F(
74      "h11", "Invariant mass between particles generated from
           decayment", 200,
75      0, 3);
76  TH3F* h12 = new TH3F("h12", "3D azimuthal and polar angles
       distribution", 100,
77                       -1, 1, 100, -1, 1, 100, -1, 1);
78
79  // Applying Sumw2() method on invariant mass histograms
80  h7->Sumw2();
81  h8->Sumw2();
82  h9->Sumw2();
83  h10->Sumw2();
84  h11->Sumw2();
85
86  // Respectively number of events and particles generated per
       event
87  constexpr double nGen{1e5};
88  constexpr double nPar{1e2};
89
90  // Creating array of 100+ generated particles
```

```
91  std::array<Particle, 120> eventParticles;
92
93  // Filling histograms
94
95  double phi{};
96  double theta{};
97  double p{};
98  double xRNDM{};
99  double yRNDM{};
100
101 for (int i{}; i != nGen; ++i) {
102   // Used to track eventParticles effective size every event
103   int effectiveSize{100};
104
105   for (int j{}; j != nPar; ++j) {
106     Particle particle;
107
108     phi = gRandom->Uniform(0, 2 * TMath::Pi());
109     theta = gRandom->Uniform(0, TMath::Pi());
110     p = gRandom->Exp(1);
111     xRNDM = gRandom->Rndm();
112
113     if (xRNDM < 0.4) {
114       particle.SetIndex("\u03C0+");
115     } else if (xRNDM < 0.8) {
116       particle.SetIndex("\u03C0-");
117     } else if (xRNDM < 0.85) {
118       particle.SetIndex("K+");
119     } else if (xRNDM < 0.90) {
120       particle.SetIndex("K-");
121     } else if (xRNDM < 0.945) {
122       particle.SetIndex("p+");
123     } else if (xRNDM < 0.99) {
124       particle.SetIndex("p-");
125     } else {
126       particle.SetIndex("K*");
127     }
128
129     particle.SetP(p * TMath::Sin(theta) * TMath::Cos(phi),
130                   p * TMath::Sin(theta) * TMath::Sin(phi),
131                   p * TMath::Cos(theta));
132
133     eventParticles[j] = particle;
134
135     h1->Fill(particle.GetIndex());
136     h2->Fill(phi);
137     h3->Fill(theta);
138     h12->Fill(TMath::Sin(theta) * TMath::Cos(phi),
139               TMath::Sin(theta) * TMath::Sin(phi), TMath::Cos(
                    theta));
140     h4->Fill(p);
```

```
141       h5->Fill(TMath::Sqrt(particle.GetPx() * particle.GetPx() +
142                            particle.GetPy() * particle.GetPy()));
143       h6->Fill(particle.GetEnergy());
144     }
145
146     std::for_each(eventParticles.begin(), eventParticles.end(),
147                   [&](Particle const& par) {
148                     if (par.GetIndex() == 6) {
149                       Particle dau1;
150                       Particle dau2;
151                       yRNDM = gRandom->Rndm();
152
153                       if (yRNDM < 0.5) {
154                         dau1.SetIndex("\u03C0+");
155                         dau2.SetIndex("K-");
156                         par.Decay2Body(dau1, dau2);
157                       } else {
158                         dau1.SetIndex("\u03C0-");
159                         dau2.SetIndex("K+");
160                         par.Decay2Body(dau1, dau2);
161                       }
162                       eventParticles[effectiveSize] = dau1;
163                       eventParticles[effectiveSize + 1] = dau2;
164                       effectiveSize += 2;
165
166                       h11->Fill(dau1.GetInvMass(dau2));
167                     }
168                   });
169
170     // Variable needed to find all combinations of the particles
       that are in the
171     // array, taken at groups of two
172     int n{};
173
174     std::for_each(
175         eventParticles.begin(), eventParticles.begin() +
              effectiveSize - 1,
176         [&](Particle const& par_i) {
177           auto nx = std::next((eventParticles.begin() + n));
178
179           std::for_each(
180               nx, eventParticles.begin() + effectiveSize - 1,
181               [&](Particle const& par_j) {
182                 if (par_i.GetCharge() * par_j.GetCharge() < 0) {
183                   h7->Fill(par_i.GetInvMass(par_j));
184                   if (((par_i.GetIndex() == 0 && par_j.GetIndex()
                           == 3) ||
185                        (par_i.GetIndex() == 3 && par_j.GetIndex()
                              == 0)) ||
186                       ((par_i.GetIndex() == 1 && par_j.GetIndex()
                              == 2) ||
```

```
                        (par_i.GetIndex() == 2 && par_j.GetIndex()
                           == 1))) {
                      h9->Fill(par_i.GetInvMass(par_j));
                    }
                  } else if (par_i.GetCharge() * par_j.GetCharge() >
                      0) {
                    h8->Fill(par_i.GetInvMass(par_j));
                    if (((par_i.GetIndex() == 0 && par_j.GetIndex()
                       == 2) ||
                         (par_i.GetIndex() == 2 && par_j.GetIndex()
                           == 0)) ||
                        ((par_i.GetIndex() == 1 && par_j.GetIndex()
                           == 3) ||
                         (par_i.GetIndex() == 3 && par_j.GetIndex()
                           == 1))) {
                      h10->Fill(par_i.GetInvMass(par_j));
                    }
                  }
                });
            ++n;
          });

    // Clearing the array at the end of every event
    std::fill(eventParticles.begin(), eventParticles.end(),
       Particle());
  }

  // Writing all on TFile
  file->cd();
  file->Write();
  file->Close();
}

// Add main in order to compile from SHELL
int main() {
  setStyle();

  simulation();

  return EXIT_SUCCESS;
}
```

Code 9: File di implementazione dell'analisi degli eventi.

```
// To compile in SHELL: "analysis.cpp `root-config --cflags --libs
   `"

#include <array>
#include <iomanip>
#include <iostream>

#include "TCanvas.h"
```

```
#include "TF1.h"
#include "TFile.h"
#include "TH1F.h"
#include "TH3F.h"
#include "TLatex.h"
#include "TLegend.h"
#include "TROOT.h"
#include "TStyle.h"

// Draw with Plain style in ROOT
void setStyle() {
  gROOT->SetStyle("Default");
  gStyle->SetOptStat(1122);
  gStyle->SetOptFit(1111);
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(0);
}

void analysis() {
  using namespace std;

  // Loading ROOT File
  TFile* file1 = new TFile("simulation_collisions.root", "READ");

  // Creating ROOT File
  TFile* file2 = new TFile("analysis_collisions.root", "RECREATE")
     ;

  // Reading histograms in ROOT File
  TH1F* h1 = (TH1F*)file1->Get("h1");
  TH1F* h2 = (TH1F*)file1->Get("h2");
  TH1F* h3 = (TH1F*)file1->Get("h3");
  TH1F* h4 = (TH1F*)file1->Get("h4");
  TH1F* h5 = (TH1F*)file1->Get("h5");
  TH1F* h6 = (TH1F*)file1->Get("h6");
  TH1F* h7 = (TH1F*)file1->Get("h7");
  TH1F* h8 = (TH1F*)file1->Get("h8");
  TH1F* h9 = (TH1F*)file1->Get("h9");
  TH1F* h10 = (TH1F*)file1->Get("h10");
  TH1F* h11 = (TH1F*)file1->Get("h11");
  TH3F* h12 = (TH3F*)file1->Get("h12");

  // Creating Canvas
  TCanvas* c1 = new TCanvas("c1", "Particle Distribution", 200,
     10, 900, 500);
  c1->Divide(2, 2);

  TCanvas* c2 = new TCanvas("c2", "Impulse", 200, 10, 900, 500);
  c2->Divide(1, 2);

  TCanvas* c3 = new TCanvas("c3", "Energy", 200, 10, 900, 500);
```

```
57  TCanvas* c4 = new TCanvas("c4", "Invariant Mass", 200, 10, 900,
        500);
58  c4->Divide(2, 2);
59
60  TCanvas* c5 =
61      new TCanvas("c5", "Invariant Mass Decayed Particles", 200,
            10, 900, 500);
62  TCanvas* c6 = new TCanvas("c6", "1st Difference", 200, 10, 900,
        500);
63  TCanvas* c7 = new TCanvas("c7", "2nd Difference", 200, 10, 900,
        500);
64
65  // Creating functions for fitting
66  TF1* f1 = new TF1("funiform1", "pol0", 0., 2 * TMath::Pi());
67  TF1* f2 = new TF1("funiform2", "pol0", 0., TMath::Pi());
68  TF1* f3 = new TF1("fexpo", "expo", 0., 10.);
69  TF1* f4 = new TF1("fgaus1", "gaus", 0.3, 1.5);
70  TF1* f5 = new TF1("fgaus2", "gaus", 0.6, 1.2);
71  TF1* f6 = new TF1("fgaus3", "gaus", 0.6, 1.2);
72
73  // Creating histograms of differences using copy constructor
74  TH1F* hDiff1 = new TH1F(*h7);
75  TH1F* hDiff2 = new TH1F(*h9);
76
77  hDiff1->SetTitle(
78      "Difference between particles with concordand/discordant
            charge");
79  hDiff2->SetTitle(
80      "Difference between concordant/discordant decayed particles"
            );
81
82  hDiff1->Add(h7, h8, 1, -1);
83  hDiff2->Add(h9, h10, 1, -1);
84
85  // Creating array that contains all TH1F histograms
86  array<TH1F*, 13> histos{h1, h2, h3,  h4,  h5,     h6,    h7,
87                          h8, h9, h10, h11, hDiff1, hDiff2};
88
89  // Setting parameters
90  double const kMass{0.89166};
91  double const kWidth{0.050};
92  f1->SetParameter(0, 1e4);
93  f2->SetParameter(0, 1e4);
94  f3->SetParameters(0, -1);
95  f4->SetParameter(1, kMass);
96  f4->SetParameter(2, kWidth);
97  f5->SetParameter(1, kMass);
98  f5->SetParameter(2, kWidth);
99  f6->SetParameter(1, kMass);
100 f6->SetParameter(2, kWidth);
101
```

```
102   // Setting parameters'names
103   f1->SetParNames("Constant");
104   f2->SetParNames("Constant");
105   f3->SetParNames("Constant", "Tau");
106   f4->SetParNames("Constant", "MassK*", "WidthK*");
107   f5->SetParNames("Constant", "MassK*", "WidthK*");
108   f6->SetParNames("Constant", "MassK*", "WidthK*");
109
110   // Cosmetics
111   f1->SetLineColor(kYellow);
112   f2->SetLineColor(kYellow);
113   f3->SetLineColor(kRed);
114   f4->SetLineColor(kRed);
115   f5->SetLineColor(kRed);
116   f6->SetLineColor(kRed);
117   f1->SetLineWidth(3);
118   f2->SetLineWidth(3);
119   f3->SetLineWidth(3);
120   f4->SetLineWidth(3);
121   f5->SetLineWidth(3);
122   f6->SetLineWidth(3);
123   f1->SetLineStyle(2);
124   f2->SetLineStyle(2);
125   f3->SetLineStyle(2);
126   f4->SetLineStyle(2);
127   f5->SetLineStyle(2);
128   f6->SetLineStyle(2);
129
130   // Fitting
131   h2->Fit("funiform1", "BQR");
132   h3->Fit("funiform2", "BQR");
133   h4->Fit("fexpo", "BQR");
134   h11->Fit("fgaus3", "BQR");
135   hDiff1->Fit("fgaus1", "BQR");
136   hDiff2->Fit("fgaus2", "BQR");
137
138   // Native array of particles'name
139   const char* label[7]{"\u03C0+", "\u03C0-", "K+", "K-", "p+", "p-
        ", "K*"};
140
141   // Drawing histograms on Canvas
142   for_each(histos.begin(), histos.end(), [&](TH1F* h) {
143     if (h == h1) {
144       c1->cd(1);
145       h->GetXaxis()->SetBinLabel(1, "#pi+");
146       h->GetXaxis()->SetBinLabel(2, "#pi-");
147       h->GetXaxis()->SetBinLabel(3, label[2]);
148       h->GetXaxis()->SetBinLabel(4, label[3]);
149       h->GetXaxis()->SetBinLabel(5, label[4]);
150       h->GetXaxis()->SetBinLabel(6, label[5]);
151       h->GetXaxis()->SetBinLabel(7, label[6]);
```

```
      h->GetXaxis()->SetTitle("Particles");
    } else if (h == h2) {
      c1->cd(2);
      h->GetXaxis()->SetTitle("#theta [rad]");
      h->SetMaximum(12000);
      h->SetMinimum(8000);
    } else if (h == h3) {
      c1->cd(3);
      h->GetXaxis()->SetTitle("#phi [rad]");
      h->SetMaximum(12000);
      h->SetMinimum(8000);
    } else if (h == h4) {
      c2->cd(1);
      h->GetXaxis()->SetTitle("Impulse [GeV/c]");
    } else if (h == h5) {
      c2->cd(2);
      h->GetXaxis()->SetTitle("Impulse [GeV/c]");
    } else if (h == h6) {
      c3->cd();
      h->GetXaxis()->SetTitle("Energy [GeV]");
      h->SetAxisRange(0., 6., "X");
    } else if (h == h7) {
      c4->cd(1);
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
    } else if (h == h8) {
      c4->cd(2);
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
    } else if (h == h9) {
      c4->cd(3);
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
    } else if (h == h10) {
      c4->cd(4);
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
    } else if (h == h11) {
      c5->cd();
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
      h->SetAxisRange(0.5, 1.5, "X");
    } else if (h == hDiff1) {
      c6->cd();
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
      h->SetAxisRange(0., 2.5, "X");
    } else if (h == hDiff2) {
      c7->cd();
      h->GetXaxis()->SetTitle("Mass [GeV/c^{2}]");
      h->SetAxisRange(0., 2.5, "X");
    }

    // Cosmetics
    h->SetMarkerStyle(20);
    h->SetMarkerSize(0.5);
    h->SetLineColor(kBlue + 4);
```

```
    h->SetFillColor(kBlue - 2);
    h->GetYaxis()->SetTitleOffset(1.2);
    h->GetXaxis()->SetTitleSize(0.04);
    h->GetYaxis()->SetTitleSize(0.04);
    h->GetYaxis()->SetTitle("Entries");
    gStyle->SetOptStat(1002200);
    gStyle->SetOptFit(1111);
    h->DrawCopy("H");
    h->DrawCopy("E,P,SAME");
  });

  // Drawing 3D angle distribution
  c1->cd(4);
  h12->SetMarkerColor(kBlue + 4);
  h12->GetXaxis()->SetTitleOffset(1.5);
  h12->GetYaxis()->SetTitleOffset(1.5);
  h12->GetZaxis()->SetTitleOffset(1.2);
  h12->GetXaxis()->SetTitleSize(0.04);
  h12->GetYaxis()->SetTitleSize(0.04);
  h12->GetZaxis()->SetTitleSize(0.04);
  h12->GetXaxis()->SetTitle("x-density");
  h12->GetYaxis()->SetTitle("y-density");
  h12->GetZaxis()->SetTitle("z-density");
  h12->DrawCopy();

  // Printing name and entries for all histograms
  for_each(histos.begin(), histos.end(), [&](TH1F* h) {
    cout << left << setw(10) << "\nName:" << h->GetTitle() << left
        << setw(10)
         << "\nEntries:" << h->Integral() << '\n';

    if (h == h1) {
      for (int i{1}; i != h->GetNbinsX() + 1; ++i) {
        cout << "\n\u0025 of " << label[i - 1] << left << setw(10)
             << " generated: " << (h->GetBinContent(i) / h->
                GetEntries()) * 100.
             << "\u0025\n"
             << label[i - 1] << " in " << i << left << setw(12)
             << " bin:" << h->GetBinContent(i) << " \u00b1 "
             << h->GetBinError(i) << '\n';
      }
    }
  });

  // Adding legend
  TLegend* leg1 = new TLegend(.1, .7, .3, .9);
  TLegend* leg2 = new TLegend(.1, .7, .3, .9);
  TLegend* leg3 = new TLegend(.7, .3458, .9, .536);
  TLegend* leg4 = new TLegend(.1, .7, .3, .9);
  TLegend* leg5 = new TLegend(.1, .7, .3, .9);
  TLegend* leg6 = new TLegend(.1, .7, .3, .9);
```

```
252
253     leg1->SetFillColor(0);
254     leg1->AddEntry(h2, "#theta distribution");
255     leg1->AddEntry(f1, "Uniform distribution");
256     c1->cd(2);
257     leg1->Draw("SAME");
258     leg2->SetFillColor(0);
259     leg2->AddEntry(h3, "#phi distribution");
260     leg2->AddEntry(f2, "Uniform distribution");
261     c1->cd(3);
262
263     leg2->Draw("SAME");
264     leg3->SetFillColor(0);
265     leg3->AddEntry(h4, "Impulse distribution");
266     leg3->AddEntry(f3, "Exponential distribution");
267     c2->cd(1);
268     leg3->Draw("SAME");
269     leg4->SetFillColor(0);
270     leg4->AddEntry(h11, "Benchmark distribution");
271     leg4->AddEntry(f4, "Gaussian distribution");
272     c5->cd();
273     leg4->Draw("SAME");
274     leg5->SetFillColor(0);
275     leg5->AddEntry(hDiff1, "Difference_1 distribution");
276     leg5->AddEntry(f5, "Gaussian distribution");
277     c6->cd();
278     leg5->Draw("SAME");
279     leg6->SetFillColor(0);
280     leg6->AddEntry(hDiff2, "Difference_2 distribution");
281     leg6->AddEntry(f6, "Gaussian distribution");
282     c7->cd();
283     leg6->Draw("SAME");
284
285     // Printing datas in SHELL
286
287     // Azimuthal angle
288     cout << "\nAzimuthal angle fit:" << '\n'
289          << f1->GetParName(0) << left << setw(29) << ':' << f1->
               GetParameter(0)
290          << " \u00b1 " << f1->GetParError(0) << left << setw(39)
291          << "\n\u03c7^2/NDF azimuthal angle fit:"
292          << f1->GetChisquare() / f1->GetNDF() << left << setw(39)
293          << "\n\u03c7^2 probability azimuthal angle fit:" << f1->
               GetProb()
294          << '\n';
295
296     // Polar angle
297     cout << "\nPolar angle fit:" << '\n'
298          << f2->GetParName(0) << left << setw(25) << ':' << f2->
               GetParameter(0)
299          << " \u00b1 " << f2->GetParError(0) << left << setw(35)
```

```
300         << "\n\u03c7^2/NDF polar angle fit:" << f2->GetChisquare()
               / f2->GetNDF()
301         << left << setw(35)
302         << "\n\u03c7^2 probability polar angle fit:" << f2->GetProb
               () << '\n';
303
304   // 3D impulse
305   cout << "\n3D impulse fit:" << '\n'
306         << f3->GetParName(0) << left << setw(24) << ':' << f3->
               GetParameter(0)
307         << " \u00b1 " << f3->GetParError(0) << '\n'
308         << f3->GetParName(1) << left << setw(29) << ':' << f3->
               GetParameter(1)
309         << " \u00b1 " << f3->GetParError(1) << left << setw(34)
310         << "\n\u03c7^2/NDF 3D impulse fit:" << f3->GetChisquare() /
                f3->GetNDF()
311         << left << setw(34)
312         << "\n\u03c7^2 probability 3d impulse fit:" << f3->GetProb
               () << '\n';
313
314   // K* 1st difference
315   cout << '\n'
316         << f4->GetParName(1) << left << setw(30) << " =" << f4->
               GetParameter(1)
317         << " \u00b1 " << f4->GetParError(1) << '\n'
318         << f4->GetParName(2) << left << setw(29) << " =" << f4->
               GetParameter(2)
319         << " \u00b1 " << f4->GetParError(2) << '\n'
320         << f4->GetParName(0) << left << setw(28) << " =" << f4->
               GetParameter(0)
321         << " \u00b1 " << f4->GetParError(0) << left << setw(38)
322         << "\n\u03c7^2/NDF 1st difference fit:"
323         << f4->GetChisquare() / f4->GetNDF() << left << setw(38)
324         << "\n\u03c7^2 probability 1st difference fit:" << f4->
               GetProb() << '\n';
325
326   // K* 2nd difference
327   cout << '\n'
328         << f5->GetParName(1) << left << setw(30) << " =" << f5->
               GetParameter(1)
329         << " \u00b1 " << f5->GetParError(1) << '\n'
330         << f5->GetParName(2) << left << setw(29) << " =" << f5->
               GetParameter(2)
331         << " \u00b1 " << f5->GetParError(2) << '\n'
332         << f5->GetParName(0) << left << setw(28) << " =" << f5->
               GetParameter(0)
333         << " \u00b1 " << f5->GetParError(0) << left << setw(38)
334         << "\n\u03c7^2/NDF 2st difference fit:"
335         << f5->GetChisquare() / f5->GetNDF() << left << setw(38)
336         << "\n\u03c7^2 probability 2nd difference fit:" << f5->
               GetProb()
```

```
337         << "\n\n";
338
339   // Writing on new TFile
340   file2->cd();
341   c1->Write();
342   c2->Write();
343   c3->Write();
344   c4->Write();
345   c5->Write();
346   c6->Write();
347   c7->Write();
348
349   // Saving Canvas in .pdf, .png and .jpg formats
350   c1->Print("particleDistribution.pdf");
351   c2->Print("impulse.pdf");
352   c3->Print("energy.pdf");
353   c4->Print("invariantMass.pdf");
354   c5->Print("invariantMassDecay.pdf");
355   c6->Print("decayParticleData1.pdf");
356   c7->Print("decayParticleData1.pdf");
357   c1->Print("particleDistribution.png");
358   c2->Print("impulse.png");
359   c3->Print("energy.png");
360   c4->Print("invariantMass.png");
361   c5->Print("invariantMassDecay.png");
362   c6->Print("decayParticleData1.png");
363   c7->Print("decayParticleData1.png");
364   c1->Print("particleDistribution.jpg");
365   c2->Print("impulse.jpg");
366   c3->Print("energy.jpg");
367   c4->Print("invariantMass.jpg");
368   c5->Print("invariantMassDecay.jpg");
369   c6->Print("decayParticleData1.jpg");
370   c7->Print("decayParticleData1.jpg");
371
372   file2->Close();
373   file1->Close();
374 }
375
376 // Add main in order to compile from SHELL
377 int main() {
378   analysis();
379
380   return EXIT_SUCCESS;
381 }
```