Simulazione di eventi di collisioni tra particelle elementari

Silvia Vicentini

2 gennaio 2024

1 Introduzione

Il programma è stato realizzato col fine di simulare ed analizzare eventi di collisione tra particelle elementari generate secondo proporzioni ben definite. Le particelle generate appartengono ai seguenti tipi: π^+, π^-, K^+, K^-, P^+, P^-, K^{0^*}. K^{0^*} è l'unica tipologia di particella instabile, le altre sono stabili. Ciascun tipo di particella è univocamente definito dal nome, la massa, la carica ed eventualmente anche dalla larghezza di risonanza Γ, che esprime la durata media della vita della particella. Ogni particella (appartenente ad una categoria specifica tra le precedenti) è in più caratterizzata da quantità di moto $\vec{p} = (p_x, p_y, p_z)$. Dai prodotti di decadimento di K^{0^*} si ricava la massa a riposo della stessa applicando le leggi della relatività ristretta.

Il codice è scritto in linguaggio C++, seguendo il modello di programmazione ad oggetti basato sull'ereditarietà virtuale e sul polimorfismo dinamico. Si è fatto uso della generazione Monte Carlo di ROOT per ottenere valori casuali da distribuzioni ben definite: per la generazione delle particelle secondo le definite proporzioni, per produrre a partire da una distribuzione uniforme gli angoli polare e azimutale necessari per stabilire la direzione dell'impulso di ciascuna particella, per estrarre da una distribuzione esponenziale il modulo della quantità di moto di queste. I risultati ottenuti sono stati rappresentati su istogrammi ROOT e si è svolta l'analisi dei dati tramite fit agli istogrammi.

2 Struttura del codice

Nel codice vengono definite tre diverse classi:

- ParticleType
- ResonanceType
- Particle

La classe ParticleType rappresenta le particelle di tipo stabile, definite da un nome, una massa e una carica. Questi sono attributi privati costanti della classe. Tra i metodi pubblici si trovano due costruttori, uno parametrico, l'altro di default, i metodi *getter* degli attributi e un metodo Print, che stampa in output le proprietà caratteristiche del tipo.

La classe ResonanceType rappresenta le particelle di tipo instabile. Essendo un tipo specializzato di ParticleType, eredita tutti gli attributi ed i metodi di ParticleType. A questi aggiunge l'attributo privato costante larghezza di risonanza ed il corrispondente metodo *getter*. Il metodo Print è stato modificato in maniera tale da stampare anche la larghezza. Per fare

ciò, l'attributo e le due funzioni appena citate sono state dichiarate in ParticleType come virtual.

La classe Particle rappresenta le singole particelle distinte a partire dalla quantità di moto che ciascuna possiede. Attributi privati di questa classe sono infatti il nome della particella e le componenti p_x, p_y, p_z. L'istanza viene legata ad una tipologia specifica di particella tramite composizione (più conveniente in termini di memoria rispetto all'ereditarietà virtuale perché risparmia di duplicare le proprietà di base per un ampio numero di oggetti): tra gli attributi privati è definito un vettore di puntatori a oggetti di tipo ParticleType, nominato fParticleType, static, comune a tutti gli oggetti di tipo Particle. AddParticleType consente di riempire il vettore coi tipi desiderati. fIndex è invece un intero che assume il valore della posizione del tipo corrispondente alla particella in esame all'interno del vettore grazie al metodo privato FindParticle che si basa sul confronto dei nomi di Particle e della corrispondente ParticleType.

Tra i metodi pubblici vi sono i *getter*, che consentono di restituire i valori delle proprietà di base e cinematiche della particella ed i *print* che stampano tali valori. I *setter* permettono di modificare il valore della quantità di moto e dell'indice. Energy e InvMass calcolano l'energia (1) e la massa invariante (2) di una particella decaduta. Decay2Body e Boost eseguono il decadimento delle particelle instabili seguendo le leggi relativistiche.

$$E = \sqrt{m^2 + \|\vec{p}\|^2} \tag{1}$$

$$M_{1,2} = \sqrt{(E_1 + E_2)^2 - \|\vec{p}_1 + \vec{p}_2\|^2} \tag{2}$$

Il corretto funzionamento di ciascun metodo definito nelle tre classi è verificato all'interno del file test.cpp tramite il modello di unit testing DOCTEST.

3 Generazione

Nel file simulation.cpp si esegue la generazione degli eventi risultanti dalle collisioni di particelle elementari. Inizialmente si inseriscono all'interno del vettore fParticleType i tipi di particelle di nostro interesse. Si generano due cicli for annidati, quello esterno su 10^5 eventi, quello interno su 100 particelle, per un totale di 10^7 istanze. Ad ogni iterazione del ciclo interno viene generata una nuova particella tra le seguenti secondo le definite proporzioni:

- 40% π^+
- 40% π^-
- 5% K^+
- 5% K^-
- 4.5% P^+
- 4.5% P^-
- 1% K^{0^*}

Per fare ciò si è fatto ricorso al metodo Monte Carlo della Classe TRandom di ROOT. La particella così generata viene immagazzinata nel vettore EventParticle. Si estrae da una distribuzione esponenziale con media -1 il valore del modulo della quantità di moto $\|\vec{p}\|$ della particella. Si generano gli angoli polare ϕ e azimutale θ estraendoli da distribuzioni uniformi con dominio

rispettivamente $[0, \pi]$ e $[0, 2\pi]$. Si passa dunque in coordinate cartesiane[1] e si ricava il valore dell'impulso trasverso cioè la proiezione di $\vec{p}$ sul piano xy.[2] Sulla base della legge (1) si calcola l'energia della particella. In questa fase vengono riempiti col metodo Fill di ROOT i seguenti istogrammi:

- *Paricle types*: distribuzione dei tipi di particelle secondo definite proporzioni.
- *Impulse modulus*: distribuzione con andamento esponenziale del modulo dell'impulso $\|\vec{p}\|$.
- *Transverse impulse modulus*: distribuzione del modulo dell'impulso trasverso.
- *Particel energy*: distribuzione dell'energia.
- *Azimuthal angle*: distribuzione uniforme dell'angolo azimutale $\theta \in [0, 2\pi]$.
- *Polar angle*: distribuzione uniforme dell'angolo polare $\phi \in [0, \pi]$.
- *3D azimuthal and polar angles*: distribuzione degli angoli azimutale e polare in un istogramma bidimensionale.

Se la particella generata è del tipo K^{0^*}, questa viene fatta decadere secondo i due possibili decadimenti equiprobabili:

$$K^{0^*} \to \pi^+ K^- \tag{3}$$

$$K^{0^*} \to \pi^- K^+ \tag{4}$$

Dunque viene rimossa da EventParticle la particella decaduta e vengono aggiunte le due nuove particelle prodotte. Viene poi riempito l'istogramma *Invariant mass between particels from decayment*: distribuzione della massa invariante generata a partire dalle particelle generate dal decadimento di una stessa K^{0^*}.

Chiuso il ciclo interno si riempiono i seguenti istogrammi:

- *Invariant mass (discordant charges)*: distribuzione della massa invariante generata da due particelle di carica discorde.
- *Invariant mass (concordant charges)*: distribuzione della massa invariante generata da due particelle di carica concorde.
- *Invariant mass (π^+/K^- and π^-/K^+)*: distribuzione della massa invariante generata da due particelle appartenenti alla coppia π^+/K^- o π^-/K^+.
- *Invariant mass (π^+/K^+ and π^-/K^-)*: distribuzione della massa invariante generata da due particelle appartenenti alla coppia π^-/K^+ o π^+/K^-.
- *Invariant mass (all particles)*: distribuzione della massa invariante generata da due particelle di un evento.

Tutti gli istogrammi vengono salvati su un file root, intitolato simulation.root, così da poterli analizzare in seguito.

[1] Passaggio da coordinate polari a coordinate sferiche:

$$\begin{cases} p_x = \|\vec{p}\| sin(\theta) cos(\phi) \\ p_y = \|\vec{p}\| sin(\theta) sin(\phi) \\ p_z = \|\vec{p}\| cos(\theta) \end{cases}$$

[2] $p_\perp = \|\vec{p}\| \sin(\theta)$

4 Analisi

L'analisi degli istogrammi è implementata all'interno del file analysis.cpp. Si aprono in modalità lettura gli istogrammi salvati nel file simulation.root.
Dall'istogramma *Particle types* (Figura 1) si ricavano le abbondanze di ciascuna particella e si verifica che la loro percentuale sia consistente con quella prevista (Tabella 1). I valori ottenuti risultano tutti compatibili con quelli attesi.

Specie	Occorrenza osservata	Occorrenza attesa
π^+	$(3.999 \pm 0.002) \cdot 10^6$	$4 \cdot 10^6$
π^-	$(4.000 \pm 0.002) \cdot 10^6$	$4 \cdot 10^6$
K^+	$(5.008 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
K^-	$(5.001 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
P^+	$(4.496 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
P^-	$(4.511 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
K^{0^*}	$(9.98 \pm 0.03) \cdot 10^4$	10^5

Tabella 1: Abbondanze dei tipi di particelle generate e valori previsti su un totale di 10^7 particelle.

Si passa poi all'analisi degli istogrammi *Azimuthal angle*, *Polar angle* e *Impulse modulus* (Figura 1). Si creano tre funzioni di classe TF1 per eseguire i fit alle distribuzioni:

- per l'angolo azimutale $Entries = Par0$
- per l'angolo polare $Entries = Par0$
- per il modulo dell'impulso $Entries = e^{Par0 + Par1 \cdot p}$

Si ricavano dunque i valori di tali parametri (Tabella 2) e del chi quadro ridotto. Quest'ultimo, poiché compatibile con il valore 1, dimostra che le distribuzioni sono coerenti con l'andamento previsto. Si osserva anche che il valore di *Par1*, cioè il reciproco della media τ del grafico esponenziale, è compatibile con -1, come atteso.

Distribuzione	Parametri del Fit	χ^2	DOF	$\frac{\chi^2}{DOF}$
Fit a distribuzione angolo θ (pol0)	$Par0 = (9.999 \pm 0.003) \cdot 10^3$	1051	999	1.0
Fit a distribuzione angolo ϕ (pol0)	$Par0 = (9.999 \pm 0.003) \cdot 10^3$	999	999	1.0
Fit a distribuzione modulo impulso (expo)	$Par0$ =(12.2063±0.0004) $Par1$ =-(1.0003±0.0003) (c/GeV)	514	498	1.0

Tabella 2: Analisi dei parametri dati dai fit alle distribuzioni degli angoli polari, azimutali e del modulo dell'impulso. Si ricavano i parametri del fit e il valore del chi quadro.

Infine si studia il segnale di risonanza di K^{0^*}. Per fare ciò si eseguono le due seguenti operazioni servendosi del metodo Add di ROOT:

- Si sottrae all'istogramma *Invariant mass (discordant charges)* l'istogramma *Invariant mass (concordant charges)*
- Si sottrae all'istogramma *Invariant mass (π^+/K^- and π^-/K^+)* l'istogramma *Invariant mass (π^+/K^+ and π^-/K^-)*

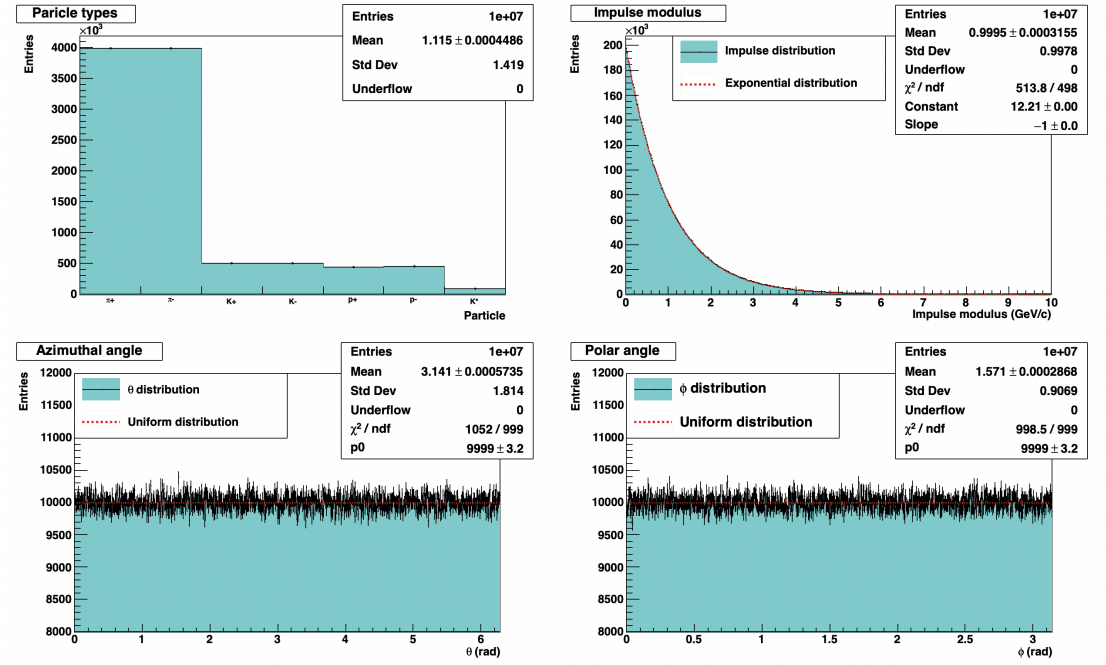

Figura 1: In alto a sinistra è rappresentata la distribuzione dei tipi di particelle su un totale di 10^7 particelle generate. In alto a destra è rappresentata la distrivuzione del modulo dell'impulso delle particelle generate e il fit ad una funzione esponenziale con media -1. In basso sono rappresentate le distribuzioni degli angoli rispettivamente azimutale e polare e il fit a funzioni uniformi.

Si ottengono così i grafici illustrati in Figura 2. Vengono confrontati con l'istogramma di massa invariante dato dalle particelle figlie del decadimento di K^{0^*} (Figura 3). Per ciascun istogramma si esegue un fit ad una gaussiana del tipo:

$$Entries = A \cdot e^{-(\frac{x-\mu}{2\cdot\sigma})^2} \quad (5)$$

dove A rappresenta l'ampiezza della gaussiana, μ la media e σ la deviazione standard. Si ricavano i valori dei parametri delle funzioni così ottenute e del χ^2/DOF (Tabella 3). Si osserva che le tre medie μ sono compatibili, così come le deviazioni standard σ. Il valore del chi quadro, essendo prossimo all'unità, dimostra la coerenza della distribuzione con l'andamento previsto.

Distribuzione e Fit	Media (μ) (GeV/c^2)	Sigma (σ) (GeV/c^2)	Ampiezza A	χ^2/DOF
Massa invariante vere K^{0^*} (fit gauss)	0.89173±0.00016	0.049920±0.00011	$(3.144 \pm 0.012) \cdot 10^3$	0.92
Massa Invariante ottenuta da differenza delle combinazioni di carica discorde e concorde (fit gauss)	0.886±0.005	0.050±0.005	$(1.39 \pm 0.13) \cdot 10^4$	0.88
Massa Invariante ottenuta da differenza delle combinazioni πK di carica discorde e concorde (fit gauss)	0.889±0.003	0.045±0.003	$(1.39 \pm 0.07) \cdot 10^4$	0.81

Tabella 3: Parametri dei fit a gaussiane degli istogrammi del segnale di risonanza di K^{0^*} e relativo valore del chi quadro.

Si salvano gli istogrammi così ottenuti su un nuovo file intitolato "analysis.root".

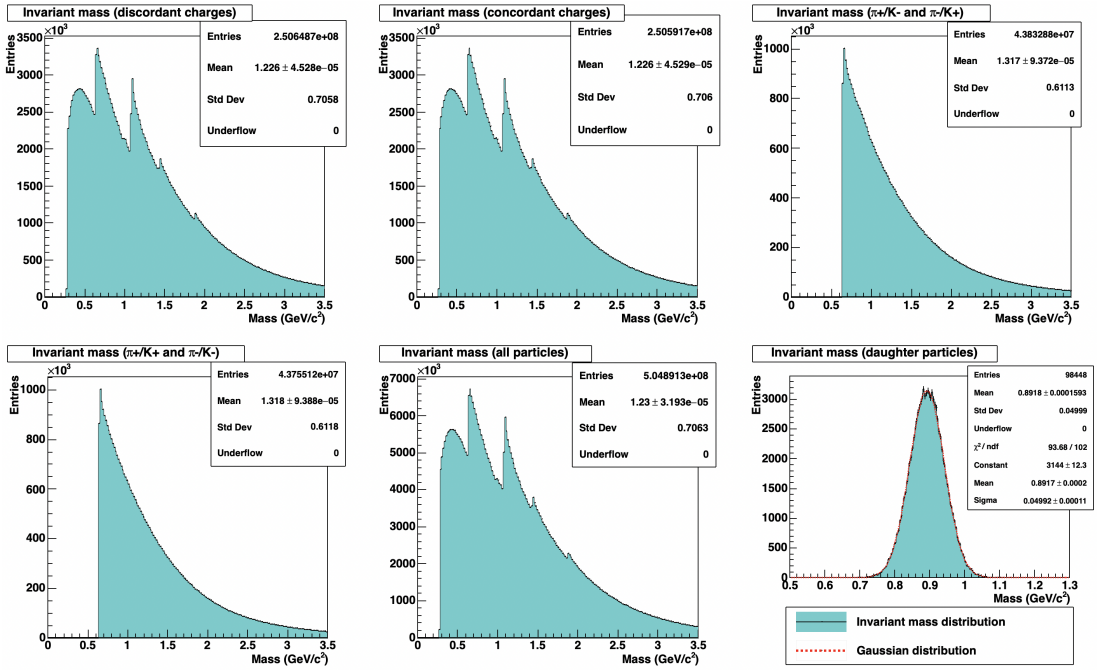

Figura 2: Partendo da in alto a destra procedendo verso sinistra: distribuzione di massa invariante data dalla combinazione delle particelle di carica discorde, distribuzione della massa invariante data dalle particelle di carica concorde, distribuzione della massa invariante data dalla coppia πK discorde, distribuzione della massa invariante data dalla coppia πK di carica concorde, distribuzione della massa invariante ottenuta dalla combinazione di tutte le particelle generate in un evento, distribuzione della massa invariante data dai prodotti del decadimento di K^{0^*}. Quest'ultimo grafico è sovrapposto al fit ad una gaussiana.

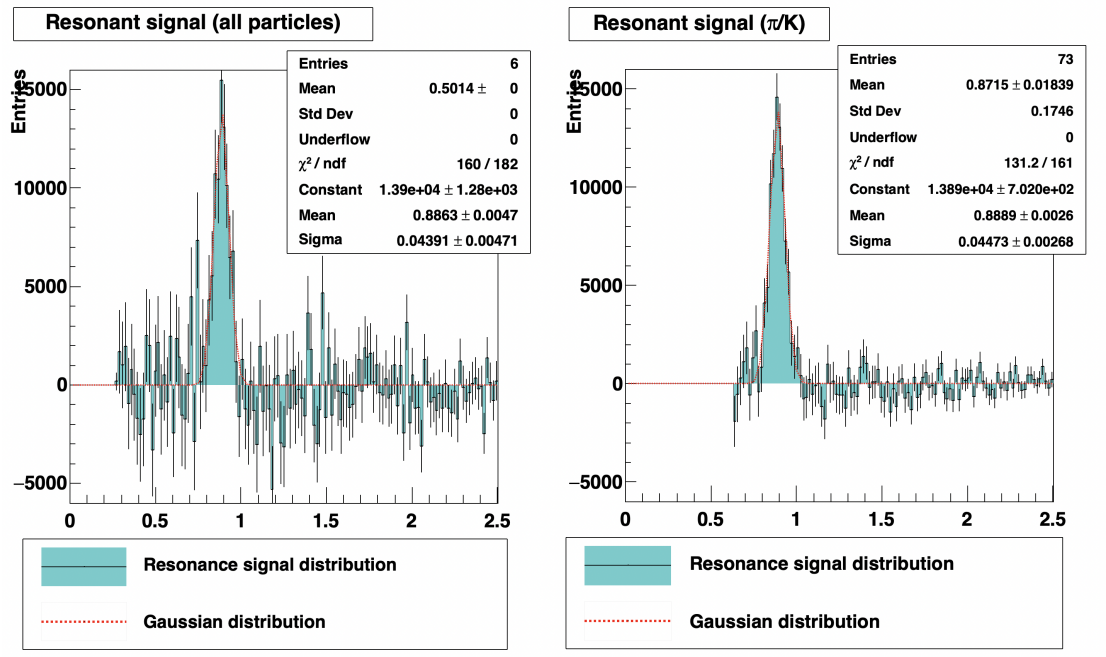

Figura 3: A sinistra è rappresentato il segnale di risonanza di K^{0^*} dato dalla differenza degli istogrammi di massa invariante tra particelle di carica discorde e concorde. Questo è sovrapposto al fit ad una gaussiana. A destra è rappresentato il grafico dato dalla differenza degli istogrammi di massa invariante dati dalle combinazioni concorde e discorde di πK. Questo è sovrapposto al fit ad una gaussiana.

5 Appendice

Viene riportato di seguito il codice del programma, suddiviso in header e source files.

Listing 1: particletype.hpp

```
#ifndef PARTICLETYPE_HPP
#define PARTICLETYPE_HPP
#include <iostream>
#include <string>

class ParticleType
{
private:
    const char *fName;
    const double fMass;
    const int fCharge;

public:
    // Constructors
    ParticleType(const char *, const double, const int);
    ParticleType();

    // Getter methods
    const char *GetName() const;
    double GetMass() const;
    int GetCharge() const;
    virtual double GetWidth() const;

    // Printer method
    virtual void Print() const;
};
#endif
```

Listing 2: particletype.cpp

```
#include "particletype.hpp"

// constructor
ParticleType::ParticleType() : fMass(0), fCharge(0) {}

ParticleType::ParticleType(const char *name,
                           const double mass,
                           const int charge) : fName(name), fMass
                              (mass), fCharge(charge)
{
}
const char *ParticleType::GetName() const
{
    return fName;
}
```

```
15 double ParticleType::GetMass() const
16 {
17     return fMass;
18 }
19 int ParticleType::GetCharge() const
20 {
21     return fCharge;
22 }
23 double ParticleType::GetWidth() const { return 0; }
24
25 void ParticleType::Print() const
26 {
27     std::cout << "-----------------------------------\n";
28     std::cout << "Particle name = " << fName << '\n';
29     std::cout << "Particle mass = " << fMass << '\n';
30     if (fCharge > 0)
31     {
32         std::cout << "Particle charge = +" << fCharge << '\n';
33     }
34     else
35     {
36         std::cout << "Particle charge = " << fCharge << '\n';
37     }
38 }
```

Listing 3: resonancetype.hpp

```
1  #ifndef RESONANCETYPE_HPP
2  #define RESONANCETYPE_HPP
3  #include "particletype.hpp"
4
5  class ResonanceType : public ParticleType
6  {
7  private:
8      const double fWidth;
9
10 public:
11     // Constructor
12     ResonanceType(const char *, const double, const int, const
           double);
13
14     // Getter method
15     double GetWidth() const override;
16
17     // Printer method
18     void Print() const override;
19 };
20
21 #endif
```

Listing 4: resonancetype.cpp

```
#include "resonancetype.hpp"

ResonanceType::ResonanceType(const char *name, const double mass,
    const int charge, const double width) : ParticleType(name,
   mass, charge), fWidth(width)
{
}

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

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

Listing 5: particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP
#include "resonancetype.hpp"
#include <vector>

class Particle
{
private:
    // Container of the differet types of particles generated
    static std::vector<ParticleType *> fParticleType;

    static const int fMaxNumParticleType = 10;
    int fIndex;
    const char *fName;
    double fPx;
    double fPy;
    double fPz;

    // Funtion to define the corresponding type of the particle
       above those incuded in fParticleType
    static int FindParticle(const char *);

    // Function used in Decay2Body
    void Boost(double, double, double);

public:
    // Constructor
```

```
27      Particle(const char *, double Px = 0, double Py = 0, double
            Pz = 0);
28
29      // Default constructor
30      Particle();
31
32      // Getter methods
33      double GetPx() const;
34      double GetPy() const;
35      double GetPz() const;
36      int GetIndex() const;
37      double GetMass() const;
38      int GetCharge() const;
39      static int GetSize();
40
41      // Function to calculate the energy of a particle
42      double Energy() const;
43
44      // Function to calculate the invariant mass of a particle
            from the products of decayment
45      double InvMass(Particle &) const;
46
47      // Function to add a type of particle to fParticleType
48      static void AddParticleType(const char *, const double, const
            int, double width = 0);
49
50      // Setter methods
51      void SetIndex(const int);
52      void SetIndex(const char *);
53      void SetP(double, double, double);
54
55      // Printer methods
56      static void PrintfParticleType();
57      void PrintParticle() const;
58
59      // Function to set the products of decayment
60      int Decay2Body(Particle &, Particle &) const;
61  };
62
63  #endif
```

Listing 6: particle.cpp

```
1   #include "particle.hpp"
2
3   #include <iomanip>
4   #include <cmath>
5   #include <cassert>
6
7   Particle::Particle()
```

```
{
    fIndex = -1;
    fPx = 0;
    fPy = 0;
    fPz = 0;
}

Particle::Particle(const char *name, double Px, double Py, double
    Pz) : fName{name}, fPx{Px}, fPy{Py}, fPz{Pz}
{
    fIndex = FindParticle(name);
    if (fIndex == -1)
    {
        std::cerr << '\n'
                  << "Error : " << name << " is not one of the
                     types considered.\n";
    }
}

std::vector<ParticleType *> Particle::fParticleType{};

int Particle::FindParticle(const char *name)
{
    for (int i = 0; i < Particle::GetSize(); ++i)
    {
        if (name == fParticleType[i]->GetName())
        {
            return i;
        }
    }
    return -1;
};

double Particle::GetPx() const { return fPx; };
double Particle::GetPy() const { return fPy; };
double Particle::GetPz() const { return fPz; };
int Particle::GetIndex() const { return fIndex; };
double Particle::GetMass() const { return fParticleType[fIndex]->
   GetMass(); };
int Particle::GetCharge() const { return fParticleType[fIndex]->
   GetCharge(); };
int Particle::GetSize() { return fParticleType.size(); }

double Particle::InvMass(Particle &p) const
{
    double p_sum_norm2 = pow(p.GetPx() + GetPx(), 2) + pow(p.
       GetPy() + GetPy(), 2) + pow(p.GetPz() + GetPz(), 2);
    return sqrt(pow((Energy() + p.Energy()), 2) - p_sum_norm2);
}
```

```
52
53 double Particle::Energy() const
54 {
55     double norm2 = fPx * fPx + fPy * fPy + fPz * fPz;
56     return sqrt(GetMass() * GetMass() + norm2);
57 }
58
59 void Particle::AddParticleType(const char *name, const double
       mass, const int charge, double width)
60 {
61     if (GetSize() < fMaxNumParticleType)
62     {
63         if (FindParticle(name) != -1)
64         {
65             std::cerr << '\n'
66                       << name << " is alseady included.\n";
67         }
68         else
69         {
70             if (width == 0)
71             {
72                 ParticleType *particle = new ParticleType(name,
                       mass, charge);
73                 fParticleType.push_back(particle);
74             }
75             else
76             {
77                 ResonanceType *particle = new ResonanceType(name,
                        mass, charge, width);
78                 fParticleType.push_back(particle);
79             }
80         }
81     }
82     else
83     {
84         std::cerr << "Can't include more than " <<
               fMaxNumParticleType << " types.\n";
85     }
86 }
87
88 void Particle::SetIndex(const int index)
89 {
90     if (index >= 0 && index <= GetSize())
91     {
92         fIndex = index;
93         fName = fParticleType[fIndex]->GetName();
94     }
95     else
96     {
```

```
97          std::cerr << "Particle can't be one of the possible types
                .\n";
98      }
99  };
100
101 void Particle::SetIndex(const char *name)
102 {
103     SetIndex(FindParticle(name));
104 };
105
106 void Particle::PrintfParticleType()
107 {
108     std::cout << "Particle Types\n";
109     std::cout << std::setw(12) << std::left << "Name" << std::
            setw(12)
110               << std::left << "Mass" << std::setw(12) << std::
                    left
111               << "Charge" << std::setw(6) << std::left << "Width"
                     << '\n';
112     for (ParticleType *particle : fParticleType)
113     {
114         std::cout << std::setw(12) << std::left << particle->
            GetName() << std::setw(12)
115                   << std::left << particle->GetMass() << std::
                      setw(12) << std::left
116                   << particle->GetCharge() << std::setw(6) << std
                      ::left << particle->GetWidth() << '\n';
117     }
118 };
119
120 void Particle::PrintParticle() const
121 {
122     if (fIndex != -1)
123     {
124         std::cout << "-----------------------------------\n";
125         std::cout << "Particle " << fIndex << " = " << fName << "
            \n";
126         std::cout << "Px = " << GetPx() << "\n";
127         std::cout << "Py = " << GetPy() << "\n";
128         std::cout << "Pz = " << GetPz() << "\n";
129     }
130     else
131     {
132         std::cout << "-----------------------------------\n";
133         std::cout << "Can't return particle index because it's
            not one of the types considered.\n";
134         std::cout << "Particle name = " << fName << "\n";
135         std::cout << "Px = " << GetPx() << "\n";
136         std::cout << "Py = " << GetPy() << "\n";
```

```
137         std::cout << "Pz = " << GetPz() << "\n";
138     }
139 }
140 
141 void Particle::SetP(double Px, double Py, double Pz)
142 {
143     fPx = Px;
144     fPy = Py;
145     fPz = Pz;
146 }
147 
148 void Particle::Boost(double bx, double by, double bz)
149 {
150     double energy = Energy();
151 
152     // Boost this Lorentz vector
153     double b2 = bx * bx + by * by + bz * bz;
154     double gamma = 1.0 / sqrt(1.0 - b2);
155     double bp = bx * fPx + by * fPy + bz * fPz;
156     double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;
157 
158     fPx += gamma2 * bp * bx + gamma * bx * energy;
159     fPy += gamma2 * bp * by + gamma * by * energy;
160     fPz += gamma2 * bp * bz + gamma * bz * energy;
161 }
162 
163 int Particle::Decay2Body(Particle &dau1, Particle &dau2) const
164 {
165     if (GetMass() == 0.0)
166     {
167         printf("Decayment cannot be preformed if mass is zero\n")
                ;
168         return 1;
169     }
170 
171     double massMot = GetMass();
172     double massDau1 = dau1.GetMass();
173     double massDau2 = dau2.GetMass();
174 
175     if (fIndex > -1)
176     { // add width effect
177 
178         // gaussian random numbers
179 
180         float x1, x2, w, y1;
181 
182         double invnum = 1. / RAND_MAX;
183         do
184         {
```

```
            x1 = 2.0 * rand() * invnum - 1.0;
            x2 = 2.0 * rand() * invnum - 1.0;
            w = x1 * x1 + x2 * x2;
        } while (w >= 1.0);

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

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

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

    double pout = sqrt((massMot * massMot - (massDau1 + massDau2)
        * (massDau1 + massDau2)) * (massMot * massMot - (massDau1
        - massDau2) * (massDau1 - massDau2))) / massMot * 0.5;

    double norm = 2 * M_PI / RAND_MAX;

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

    double energy = sqrt(fPx * fPx + fPy * fPy + fPz * fPz +
       massMot * massMot);

    double bx = fPx / energy;
    double by = fPy / energy;
    double bz = fPz / energy;

    dau1.Boost(bx, by, bz);
    dau2.Boost(bx, by, bz);

    return 0;
}
```

Listing 7: test.cpp

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN

#include "particle.hpp"

```

```
#include "doctest.h"

#include <vector>

TEST_CASE("Testing ParticleType and ResonanceType's methods")
{
    SUBCASE("ParticleType")
    {
        ParticleType particle1("e", 0.511, -1);
        CHECK(particle1.GetName() == "e");
        CHECK(particle1.GetMass() == doctest::Approx(0.511));
        CHECK(particle1.GetCharge() == -1);

        particle1.Print();
    }

    SUBCASE("ResonanceType")
    {
        ResonanceType particle2("K*", 497.648, 0, 0.6);
        CHECK(particle2.GetName() == "K*");
        CHECK(particle2.GetMass() == doctest::Approx(497.648));
        CHECK(particle2.GetCharge() == 0);
        CHECK(particle2.GetWidth() == 0.6);

        particle2.Print();
    }

    SUBCASE("Testing the Print method")
    {
        ParticleType particle3("p", 938.272, +1);
        ResonanceType particle4("K+", 493.677, +1, 0.9);
        std::vector<ParticleType *> test;
        test.push_back(&particle3);
        test.push_back(&particle4);
        for (ParticleType *particle : test)
        {
            particle->Print();
        }
    }

    SUBCASE("Testing the const methods for ParticleType")
    {
        const ParticleType particle5("\u03C0+", 139.6, +1);
        CHECK(particle5.GetName() == "\u03C0+");
        CHECK(particle5.GetMass() == doctest::Approx(139.6));
        CHECK(particle5.GetCharge() == 1);
        particle5.Print();
    }

```

```
54      SUBCASE("Testing the const methods for ResonanceType")
55      {
56          const ResonanceType particle6("\u03C0-", 139.6, -1, 0.3);
57          CHECK(particle6.GetName() == "\u03C0-");
58          CHECK(particle6.GetMass() == doctest::Approx(139.6));
59          CHECK(particle6.GetCharge() == -1);
60          CHECK(particle6.GetWidth() == 0.3);
61          particle6.Print();
62      }
63  }
64
65  TEST_CASE("Testing Particle's methods")
66  {
67      // m is expressed in GeV/c
68      Particle::AddParticleType("e", 0.511, -1);
69      Particle::AddParticleType("K*", 497.648, 0, 0.6);
70      Particle::AddParticleType("p", 938.272, +1);
71      Particle::AddParticleType("\u03C0-", 139.6, -1, 0.3);
72
73      // p is expressed in GeV
74      Particle particle1("e", 0.4599, 0.44457, 0.36792);
75      Particle particle2("K*");
76      Particle particle3("p", -938.272);
77      const Particle particle4("\u03C0-", 110.284, -115.868,
            114.472);
78      Particle particle5("p-");
79      Particle particle6("p", 798.272);
80
81      SUBCASE("Testing FindParticle function")
82      {
83          CHECK(particle1.GetIndex() == 0);
84          CHECK(particle2.GetIndex() == 1);
85          CHECK(particle3.GetIndex() == 2);
86          CHECK(particle4.GetIndex() == 3);
87          CHECK(particle5.GetIndex() == -1);
88          CHECK(particle6.GetIndex() == 2);
89      }
90
91      SUBCASE("Testing getter methods")
92      {
93          CHECK(Particle::GetSize() == 4);
94
95          CHECK(particle1.GetPx() == doctest::Approx(0.4599));
96          CHECK(particle1.GetPy() == doctest::Approx(0.44457));
97          CHECK(particle1.GetPz() == doctest::Approx(0.36792));
98          CHECK(particle2.GetPx() == 0);
99          CHECK(particle2.GetPy() == 0);
100         CHECK(particle2.GetPz() == 0);
101         CHECK(particle3.GetPx() == doctest::Approx(-938.272));
```

```
        CHECK(particle3.GetPy() == 0);

        CHECK(particle1.GetMass() == 0.511);
        CHECK(particle1.GetCharge() == -1);
    }

    SUBCASE("Testing Energy and InvMass methods")
    {

        CHECK(particle1.Energy() == doctest::Approx(0.897572627))
            ;
        CHECK(particle2.Energy() == doctest::Approx(497.648));

        CHECK(particle1.InvMass(particle2) == doctest::Approx
            (498.5450265));
    }

    Particle::PrintfParticleType();

    particle1.PrintParticle();
    particle2.PrintParticle();
    particle3.PrintParticle();
    particle4.PrintParticle();
    particle5.PrintParticle();

    SUBCASE("Testing setter methods")
    {
        Particle particle7;
        particle7.SetIndex(3);
        CHECK(particle7.GetIndex() == 3);
        CHECK(particle7.GetPx() == 0);
        CHECK(particle7.GetPy() == 0);
        CHECK(particle7.GetPz() == 0);

        particle7.SetIndex("e");
        CHECK(particle7.GetIndex() == 0);

        particle7.SetP(323.47, 353.33, 467.80);
        CHECK(particle7.GetPx() == 323.47);
        CHECK(particle7.GetPy() == 353.33);
        CHECK(particle7.GetPz() == 467.80);
    }
}
```

Listing 8: simulation.cpp

```
#include "particle.hpp"

#include "TFile.h"
#include "TH1F.h"
```

```
#include "TH3F.h"
#include "TMath.h"
#include "TROOT.h"
#include "TRandom.h"
#include "TStyle.h"

// Cosmetics
void setStyle()
{
    gROOT->SetStyle("Plain");
    gStyle->SetPalette(57);
    gStyle->SetOptTitle(0);
    gStyle->SetOptStat(1111);
}

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

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

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

    // Setting all particle types
    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, 0.050);

    // Creating histograms
    TH1F *h0 = new TH1F("h0", "Paricle types", 7, 0, 7);
    TH1F *h1 = new TH1F("h1", "Impulse modulus", 500, 0, 10);
    TH1F *h2 = new TH1F("h2", "Transverse impulse modulus", 500,
       0, 10);
    TH1F *h3 = new TH1F("h3", "Particle energy", 500, 0, 10);
    TH1F *h4 = new TH1F("h4", "Azimuthal angle", 1E3, 0, 2 *
       TMath::Pi());
    TH1F *h5 = new TH1F("h5", "Polar angle", 1E3, 0, TMath::Pi())
       ;
    TH3F *h12 = new TH3F("h12", "3D azimuthal and polar angles",
       100, -1, 1, 100, -1, 1, 100, -1, 1);
    TH1F *h6 = new TH1F("h6", "Invariant mass (discordant charges
       )", 200, 0, 3.5);
```

```
49      TH1F *h7 = new TH1F("h7", "Invariant mass (concordant charges
            )", 200, 0, 3.5);
50      TH1F *h8 = new TH1F("h8", "Invariant mass (#pi+/K- and #pi-/K
            +)", 200, 0, 3.5);
51      TH1F *h9 = new TH1F("h9", "Invariant mass (#pi+/K+ and #pi-/K
            -)", 200, 0, 3.5);
52      TH1F *h10 = new TH1F("h10", "Invariant mass (all particles)",
             200, 0, 3.5);
53      TH1F *h11 = new TH1F("h11", "Invariant mass (daughter
            particles)", 200, 0.5, 1.3);
54
55      // To guarantee a correct use of errors
56      h6->Sumw2();
57      h7->Sumw2();
58      h8->Sumw2();
59      h9->Sumw2();
60      h10->Sumw2();
61      h11->Sumw2();
62
63      std::vector<Particle> EventParticle;
64
65      const int nGen = 1E5;
66      const int nParticles = 100;
67      double phi{};
68      double theta{};
69      double p{};
70      double x{};
71      double y{};
72
73      for (int j{}; j < nGen; ++j)
74      {
75          for (int i{}; i < nParticles; ++i)
76          {
77              Particle particle;
78              x = gRandom->Rndm();
79
80              // Setting particle type
81              if (x < 0.4)
82              {
83                  particle.SetIndex(0);
84              }
85              else if (x < 0.8)
86              {
87                  particle.SetIndex(1);
88              }
89              else if (x < 0.85)
90              {
91                  particle.SetIndex(2);
92              }
```

```
93              else if (x < 0.9)
94              {
95                  particle.SetIndex(3);
96              }
97              else if (x < 0.945)
98              {
99                  particle.SetIndex(4);
100             }
101             else if (x < 0.99)
102             {
103                 particle.SetIndex(5);
104             }
105             else
106             {
107                 particle.SetIndex(6);
108             }
109             h0->Fill(particle.GetIndex());
110
111             // Setting phi, theta and p
112             phi = gRandom->Rndm() * TMath::Pi();
113             theta = gRandom->Rndm() * 2 * TMath::Pi();
114             p = gRandom->Exp(1);
115             particle.SetP(p * sin(theta) * cos(phi), p * sin(
                    theta) * sin(phi), p * cos(theta));
116             h12->Fill(sin(theta) * cos(phi), sin(theta) * sin(phi
                    ), cos(theta));
117             h1->Fill(p);
118             h2->Fill(p * sin(theta));
119             h3->Fill(particle.Energy());
120             h4->Fill(theta);
121             h5->Fill(phi);
122
123             EventParticle.push_back(particle);
124
125             // Decayment
126             if (EventParticle[i].GetIndex() == 6)
127             {
128                 y = gRandom->Rndm();
129                 if (y < 0.5)
130                 {
131                     Particle dau1("\u03C0-");
132                     Particle dau2("K+");
133                     EventParticle[i].Decay2Body(dau1, dau2);
134                     EventParticle.pop_back();
135                     EventParticle.push_back(dau1);
136                     EventParticle.push_back(dau2);
137                     h11->Fill(dau1.InvMass(dau2));
138                 }
139                 else
```

```
                {
                    Particle dau1("\u03C0+");
                    Particle dau2("K-");
                    EventParticle[i].Decay2Body(dau1, dau2);
                    EventParticle.pop_back();
                    EventParticle.push_back(dau1);
                    EventParticle.push_back(dau2);
                    h11->Fill(dau1.InvMass(dau2));
                }
            }
        }

        int size = EventParticle.size();

        // Filling invariant mass histograms
        for (int a = 0; a < size; ++a)
        {
            for (int b = a + 1; b < size; ++b)
            {
                h10->Fill(EventParticle[a].InvMass(EventParticle[
                    b]));
                if (EventParticle[a].GetCharge() * EventParticle[
                    b].GetCharge() < 0)
                {
                    h6->Fill(EventParticle[a].InvMass(
                        EventParticle[b]));
                    if ((EventParticle[a].GetIndex() == 0 &&
                        EventParticle[b].GetIndex() == 3) || (
                        EventParticle[a].GetIndex() == 3 &&
                        EventParticle[b].GetIndex() == 0) || (
                        EventParticle[a].GetIndex() == 1 &&
                        EventParticle[b].GetIndex() == 2) || (
                        EventParticle[a].GetIndex() == 2 &&
                        EventParticle[b].GetIndex() == 1))
                    {
                        h8->Fill(EventParticle[a].InvMass(
                            EventParticle[b]));
                    }
                }
                else if (EventParticle[a].GetCharge() *
                    EventParticle[b].GetCharge() > 0)
                {
                    h7->Fill(EventParticle[a].InvMass(
                        EventParticle[b]));

                    if ((EventParticle[a].GetIndex() == 0 &&
                        EventParticle[b].GetIndex() == 2) || (
                        EventParticle[a].GetIndex() == 2 &&
                        EventParticle[b].GetIndex() == 0) || (
```

```
                        EventParticle[a].GetIndex() == 1 &&
                        EventParticle[b].GetIndex() == 3) || (
                        EventParticle[a].GetIndex() == 3 &&
                        EventParticle[b].GetIndex() == 1))
173                 {
174                     h9->Fill(EventParticle[a].InvMass(
                            EventParticle[b]));
175                 }
176             }
177         }
178     }
179
180     // Clearing EventParticle
181     for (int i{}; i < size; ++i)
182     {
183         EventParticle.clear();
184     }
185 }
186
187 // Writing histograms on file
188 file->cd();
189 file->Write();
190 file->Close();
191 }
```

Listing 9: analysis.cpp

```
1  #include "TCanvas.h"
2  #include "TFile.h"
3  #include "TH1F.h"
4  #include "TH3F.h"
5  #include "TROOT.h"
6  #include "TF1.h"
7  #include "TMath.h"
8  #include "TStyle.h"
9  #include "TLatex.h"
10 #include "TLegend.h"
11 #include <iomanip>
12 #include <vector>
13 #include <iostream>
14
15 // Cosmetics
16 void set_style()
17 {
18     gROOT->SetStyle("Plain");
19     gStyle->SetPalette(57);
20     gStyle->SetOptStat(11220);
21     gStyle->SetOptFit(111);
22 }
23
```

```
// Histogram cosmetic
void histo_cosmetic(TH1F *h)
{
    int fillColor = 426;
    h->SetFillColor(fillColor);
    h->SetLineWidth(1);
    h->SetLineColor(kBlack);
    h->GetYaxis()->SetTitle("Entries");
    h->GetYaxis()->SetTitleOffset(1.5);
    h->DrawCopy("H");
    h->DrawCopy("E,SAME");
}

// Function cosmetic
void function_cosmetic(TF1 *f)
{
    f->SetLineStyle(2);
    f->SetLineWidth(3);
    f->SetLineColor(kRed);
}

void analysis()
{
    // Cosmetics
    set_style();

    // Reading from simulation.root and creating analysis.root
    TFile *file1 = new TFile("simulation.root", "READ");
    TFile *file2 = new TFile("analysis.root", "RECREATE");
    file1->ls();

    // Copying histograms
    TH1F *h0 = (TH1F *)file1->Get("h0");
    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 *c = new TCanvas("c", "Various histograms", 200, 10,
       950, 700);
```

```
72      c->Divide(2, 2);
73      TCanvas *cEnergy = new TCanvas("cEnergy", "Particle energy",
            200, 10, 950, 700);
74      TCanvas *cImpulse = new TCanvas("cImpulse", "Impulse and
            transverse impulse", 200, 10, 950, 700);
75      cImpulse->Divide(2, 1);
76      TCanvas *c3DAngles = new TCanvas("c3DAngles", "3D angles
            distribution", 200, 10, 950, 700);
77      TCanvas *cInvMass = new TCanvas("cInvMass", "Invariant mass",
             200, 10, 950, 700);
78      cInvMass->Divide(3, 2);
79      TCanvas *cResonance = new TCanvas("cResonance", "Resonance
            signal", 200, 10, 950, 700);
80      cResonance->Divide(2, 1);
81      TCanvas *cvuota = new TCanvas("cvuota", "cvuota signal", 20,
            10, 90, 50);
82
83      // Creating functions for fitting
84      TF1 *f1 = new TF1("f1", "expo", 0, 10);
85      TF1 *f4 = new TF1("f4", "pol0", 0, 2 * TMath::Pi());
86      TF1 *f5 = new TF1("f5", "pol0", 0, TMath::Pi());
87      TF1 *f11 = new TF1("f11", "gaus", 0.5, 1.3);
88
89      // Creating histograms for resonance signal
90      TH1F *hDiff1 = new TH1F(*h6);
91      TH1F *hDiff2 = new TH1F(*h8);
92
93      hDiff1->Add(h6, h7, 1, -1);
94      hDiff2->Add(h8, h9, 1, -1);
95
96      // Creating functions for resonance signal histograms fitting
97      TF1 *fDiff1 = new TF1("fDiff1", "gaus", 0, 3.5);
98      TF1 *fDiff2 = new TF1("fDiff2", "gaus", 0, 3.5);
99
100     // Setting functions parameters
101     double k_mass = 0.89166;
102     double k_width = 0.050;
103     f1->SetParameter(0, -1);
104     f4->SetParameter(0, 1E5);
105     f5->SetParameter(0, 1E5);
106     f11->SetParameter(k_mass, k_width);
107     fDiff1->SetParameters(k_mass, k_width);
108     fDiff2->SetParameters(k_mass, k_width);
109
110     // Setting functions cosmetic
111     function_cosmetic(f1);
112     function_cosmetic(f4);
113     function_cosmetic(f5);
114     function_cosmetic(f11);
```

```
115     function_cosmetic(fDiff1);
116     function_cosmetic(fDiff2);
117 
118     // Fitting histograms
119     h1->Fit("f1");
120     h4->Fit("f4");
121     h5->Fit("f5");
122     h11->Fit("f11");
123     hDiff1->Fit("fDiff1");
124     hDiff2->Fit("fDiff2");
125 
126     vector<TH1F *> histo{h0, h1, h4, h5, h3, h2, h6, h7, h8, h9,
           h10, h11, hDiff1, hDiff2};
127     vector<const char *> bin_names{"#pi+", "#pi-", "K+", "K-", "p
           +", "p-", "K*"};
128 
129     //  Drawing on c
130     for (int i = 0; i < 4; ++i)
131     {
132         c->cd(i + 1);
133         if (i == 0)
134         {
135             histo[i]->GetXaxis()->SetTitle("Particle");
136             for (int j = 1; j < 8; ++j)
137             {
138                 histo[i]->GetXaxis()->SetBinLabel(j, bin_names[j
                       - 1]);
139             }
140             histo[i]->SetMarkerStyle(20);
141             histo[i]->SetMarkerSize(0.3);
142         }
143         if (i == 1)
144         {
145             histo[i]->GetXaxis()->SetTitle("Impulse modulus (GeV/
                   c)");
146         }
147         if (i == 2)
148         {
149             histo[i]->GetXaxis()->SetTitle("#theta (rad)");
150             histo[i]->SetMinimum(8000);
151             histo[i]->SetMaximum(12000);
152         }
153         if (i == 3)
154         {
155             histo[i]->GetXaxis()->SetTitle("#phi (rad)");
156             histo[i]->SetMinimum(8000);
157             histo[i]->SetMaximum(12000);
158         }
159         histo_cosmetic(histo[i]);
```

```
160     }
161
162     // Drawing on cEnergy
163     cEnergy->cd();
164     histo[4]->GetXaxis()->SetTitle("Energy (GeV)");
165     histo_cosmetic(histo[4]);
166
167     // Drawing on cImpulse
168     cImpulse->cd(1);
169     histo[1]->GetXaxis()->SetTitle("Impulse modulus (GeV/c)");
170     histo_cosmetic(histo[1]);
171
172     cImpulse->cd(2);
173     histo[5]->GetXaxis()->SetTitle("Transverse impulse modulus (
            GeV/c)");
174     histo_cosmetic(histo[5]);
175
176     // Drawing on c3DAngles
177     c3DAngles->cd();
178     h12->GetXaxis()->SetTitle("sin(#theta)*cos(#phi)");
179     h12->GetYaxis()->SetTitle("sin(#theta)*sin(#phi)");
180     h12->GetZaxis()->SetTitle("cos(#theta)");
181     h12->GetXaxis()->SetTitleOffset(2);
182     h12->GetYaxis()->SetTitleOffset(2);
183     h12->GetZaxis()->SetTitleOffset(1);
184     int fillColor = 426;
185     h12->SetMarkerColor(fillColor);
186     h12->SetLineWidth(2);
187     h12->DrawCopy("H");
188     h12->DrawCopy("E, SAME");
189
190     // Drawing on cInvMass
191     for (int i = 6; i < 12; ++i)
192     {
193         cInvMass->cd(i - 5);
194         histo[i]->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
195         histo_cosmetic(histo[i]);
196     }
197
198     // Drawing on cResonance
199     for (int i = 12; i < 14; ++i)
200     {
201         cResonance->cd(i - 11);
202         histo[i]->SetMinimum(-6000);
203         histo[i]->SetMaximum(16000);
204         histo[i]->SetAxisRange(0., 2.5, "X");
205         histo[i]->GetYaxis()->SetTitleOffset(2.5);
206         if (i == 12)
207         {
```

```
208             histo[i]->SetTitle("Resonant signal (all particles)")
                    ;
209         }
210         if (i == 13)
211         {
212             histo[i]->SetTitle("Resonant signal (#pi/K)");
213         }
214         histo_cosmetic(histo[i]);
215     }
216 
217     // Creating legends
218     TLegend *leg1 = new TLegend(.2, .7, .6, .9);
219     TLegend *leg4 = new TLegend(.1, .7, .5, .9);
220     TLegend *leg5 = new TLegend(.1, .7, .5, .9);
221     TLegend *leg11 = new TLegend(.1, .7, .4, .9);
222     TLegend *legDiff1 = new TLegend(.1, .7, .3, .9);
223     TLegend *legDiff2 = new TLegend(.1, .7, .3, .9);
224 
225     // Drawing legends
226     leg1->SetFillColor(0);
227     leg1->AddEntry(h1, "Impulse distribution");
228     leg1->AddEntry(f1, "Exponential distribution");
229     c->cd(2);
230     leg1->Draw("SAME");
231 
232     leg4->SetFillColor(0);
233     leg4->AddEntry(h4, "#theta distribution");
234     leg4->AddEntry(f4, "Uniform distribution");
235     c->cd(3);
236     leg4->Draw("SAME");
237 
238     leg5->SetFillColor(0);
239     leg5->AddEntry(h5, "#phi distribution");
240     leg5->AddEntry(f5, "Uniform distribution");
241     c->cd(4);
242     leg5->Draw("SAME");
243 
244     leg11->SetFillColor(0);
245     leg11->AddEntry(h11, "Invariant mass distribution");
246     leg11->AddEntry(f11, "Gaussian distribution");
247     cInvMass->cd(6);
248     leg11->Draw("SAME");
249 
250     legDiff1->SetFillColor(0);
251     legDiff1->AddEntry(h1, "Resonance signal distribution");
252     legDiff1->AddEntry(f1, "Gaussian distribution");
253     cResonance->cd(1);
254     legDiff1->Draw("SAME");
255 
```

```
256     legDiff2->SetFillColor(0);
257     legDiff2->AddEntry(h1, "Resonance signal distribution");
258     legDiff2->AddEntry(f1, "Gaussian distribution");
259     cResonance->cd(2);
260     legDiff2->Draw("SAME");
261 
262     // Writing statistics on shell
263     std::cout << "-----------------------------------\n";
264     std::cout << "Number of \u03C0+ generated: " << left << setw
           (10) << h0->GetBinContent(1) << "    " << h0->GetBinError
           (1)
265               << " percentage: " << (h0->GetBinContent(1) / h0->
                  GetEntries()) * 100 << "%\n";
266     std::cout << "Number of \u03C0- generated: " << left << setw
           (10) << h0->GetBinContent(2) << "    " << h0->GetBinError
           (2)
267               << " percentage: " << (h0->GetBinContent(2) / h0->
                  GetEntries()) * 100 << "%\n";
268     for (int i{3}; i < 8; ++i)
269     {
270         std::cout << "Number of " << bin_names[i - 1] << left <<
               setw(10)
271                   << " generated: " << h0->GetBinContent(i) << "
                         " << h0->GetBinError(i)
272                   << " percentage: " << (h0->GetBinContent(i) /
                      h0->GetEntries()) * 100 << "%\n";
273     }
274     std::cout << "-----------------------------------\n";
275     std::cout << "Azimuthal angle distribution statistics:\n"
276               << "Parameter = " << f4->GetParameter(0) << "    "
                  << f4->GetParError(0) << '\n'
277               << "Chisquare = " << f4->GetChisquare() << '\n'
278               << "DOF = " << f4->GetNDF() << '\n'
279               << "Chisquare/DOF = " << f4->GetChisquare() / f4->
                  GetNDF() << '\n'
280               << "Probability = " << f4->GetProb() << '\n';
281     std::cout << "-----------------------------------\n";
282     std::cout << "Polar angle distribution statistics:\n"
283               << "Parameter = " << f5->GetParameter(0) << "    "
                  << f5->GetParError(0) << '\n'
284               << "Chisquare = " << f5->GetChisquare() << '\n'
285               << "DOF = " << f5->GetNDF() << '\n'
286               << "Chisquare/DOF = " << f5->GetChisquare() / f5->
                  GetNDF() << '\n'
287               << "Probability = " << f5->GetProb() << '\n';
288     std::cout << "-----------------------------------\n";
289     std::cout << "Impulse modulus distribution statistics:\n"
290               << "First parameter = " << f1->GetParameter(0) << "
                      " << f1->GetParError(0) << '\n'
```

```
291                 << "Second parameter = " << f1->GetParameter(1) <<
                        "    " << f1->GetParError(1) << '\n'
292                 << "Chisquare = " << f1->GetChisquare() << '\n'
293                 << "DOF = " << f1->GetNDF() << '\n'
294                 << "Chisquare/DOF = " << f1->GetChisquare() / f1->
                        GetNDF() << '\n'
295                 << "Probability = " << f1->GetProb() << '\n';
296     std::cout << "-----------------------------------\n";
297     std::cout << "Invariant mass from daugther particles
                    statistics:\n"
298                 << "Mean = " << f11->GetParameter(1) << "    " <<
                        f11->GetParError(1) << '\n'
299                 << "Sigma = " << f11->GetParameter(2) << "    " <<
                        f11->GetParError(2) << '\n'
300                 << "Amplitude = " << f11->GetParameter(0) << "    "
                         << f11->GetParError(0) << '\n'
301                 << "Chisquare/DOF = " << f11->GetChisquare() / f11
                        ->GetNDF() << '\n'
302                 << "Probability = " << f11->GetProb() << '\n';
303     std::cout << "-----------------------------------\n";
304     std::cout << "Resonance signal (all particles) statistics:\n"
305                 << "Mean = " << fDiff1->GetParameter(1) << "    "
                        << fDiff1->GetParError(1) << '\n'
306                 << "Sigma = " << fDiff1->GetParameter(2) << "    "
                        << fDiff1->GetParError(2) << '\n'
307                 << "Amplitude = " << fDiff1->GetParameter(0) << "
                           " << fDiff1->GetParError(0) << '\n' // Non so
                         cosa sia!!!!!
308                 << "Chisquare/DOF = " << fDiff1->GetChisquare() /
                        fDiff1->GetNDF() << '\n'
309                 << "Probability = " << fDiff1->GetProb() << '\n';
310     std::cout << "-----------------------------------\n";
311     std::cout << "Resonance signal (\u03C0/K) statistics:\n"
312                 << "Mean = " << fDiff2->GetParameter(1) << "    "
                        << fDiff2->GetParError(1) << '\n'
313                 << "Sigma = " << fDiff2->GetParameter(2) << "    "
                        << fDiff2->GetParError(2) << '\n'
314                 << "Amplitude = " << fDiff2->GetParameter(0) << "
                           " << fDiff2->GetParError(0) << '\n'
315                 << "Chisquare/DOF = " << fDiff2->GetChisquare() /
                        fDiff2->GetNDF() << '\n'
316                 << "Probability = " << fDiff2->GetProb() << '\n';
317     std::cout << "-----------------------------------\n";
318
319     // Drawing histograms on file2
320     file2->cd();
321     c->Write();
322     cEnergy->Write();
323     cImpulse->Write();
```

```
    c3DAngles->Write();
    cInvMass->Write();
    cResonance->Write();

    // Saving Canvas as .pdf, .C, .root and .jpeg
    c->Print("VariousDistributions.pdf");
    c->Print("VariousDistributions.C");
    c->Print("VariousDistributions.root");
    c->Print("VariousDistributions.jpeg");
    cEnergy->Print("EnergyDistribution.pdf");
    cEnergy->Print("EnergyDistribution.C");
    cEnergy->Print("EnergyDistribution.root");
    cEnergy->Print("EnergyDistribution.jpeg");
    cImpulse->Print("Impulse.pdf");
    cImpulse->Print("Impulse.C");
    cImpulse->Print("Impulse.root");
    cImpulse->Print("Impulse.jpeg");
    c3DAngles->Print("3DAngles.pdf");
    c3DAngles->Print("3DAngles.C");
    c3DAngles->Print("3DAngles.root");
    c3DAngles->Print("3DAngles.jpeg");
    cInvMass->Print("InvMass.pdf");
    cInvMass->Print("InvMass.C");
    cInvMass->Print("InvMass.root");
    cInvMass->Print("InvMass.jpeg");
    cResonance->Print("ResonanceSignal.pdf");
    cResonance->Print("ResonanceSignal.C");
    cResonance->Print("ResonanceSignal.root");
    cResonance->Print("ResonanceSignal.jpeg");

    file2->Close();
    file1->Close();
}
```