Simulazione delle collisioni di particelle elementari

Marchetti Tommaso, Mazzarelli Elena

1 Introduzione

L'obbiettvo del programma realizzato è quello di simulare la collisione di particelle elementari e di analizzare e rappresentare i risultati di tali eventi. In particolare le particelle e le rispettive anti-particelle prese in considerazione sono state i pioni (π^+, π^-), i kaoni (K^+, K^-), i protoni (p^+, p^-) e la particella di risonanza k*. Quest'ultima, differentemente dalle altre, è una particella instabile e di conseguenza decade molto velocemente in una coppia $\pi^+ k^-$ o $\pi^- k^+$ con una proporzionalità del 50%. In particolare dalle distribuzioni di massa invariante dei prodotti del decadimento di k* è stato possibile osservare il segnale di decadimento della particella stessa.
Il programma è stato realizzato impiegando il linguaggio di programmazione c++ e la programmazione ad oggetti. Si è inoltre fatto uso dell'ereditarietà per descrivere il rapporto fra particelle e particelle di risonanza. Queste ultime sono infatti caratterizzate non solo da un nome, una massa e una carica, proprietà condivise da tutte le particelle, ma anche da una larghezza, parametro che ne descrive la vita media. Nello specifico per l'analisi e la rappresentazione degli eventi osservati si è impiegato il framework di ROOT in modalità compilata. In particolare il codice genera 10^5 eventi, ognuno contenente 100 particelle dei tipi predefiniti utilizzando il metodo di generazione Monte Carlo. I grafici prodotti riguardano le seguenti caratteristiche: tipi di particelle, distribuzioni degli angoli, distribzuioni dell'impulso, distribuzione dell'energia e distribuzioni delle masse invarianti.

2 Struttura del codice

Il programma è composto da tre file .cpp (particleType, resonanceType, particle), dai rispettivi header e da due macro (main.cpp e analysis.cpp).

2.1 particleType e resonanceType

Nei file particleType e resonanceType vengono dichiarate e definite le classi omonime impiegando l'ereditarietà pubblica. In particolare la classe ParticleType costituisce la classe base e ha come membri privati le proprietà principali comuni a tutte le particelle (nome, massa e carica), accessibili tramite dei Getter pubblici e impostabili tramite il costruttore. Questa classe possiede anche due metodi virtuali, Print e GetWidth. Il primo stampa a schermo le proprietà della particella, mentre il secondo ritorna zero per tutte le particelle diverse da k*, in quanto una particella stabile non possiede l'attributo Width. La classe derivata ResonanceType possiede come unico attributo privato la vita media. Questo è accessibile tramite GetWidth, il quale è ridefinito per restituire il valore corretto nel caso di k*, come il metodo Print.

2.2 particle

La classe Particle è divisa fra il file particle.hpp, che contiene le dichiarazoni, e il file particle.cpp, che contiene le definizioni.
Il costruttore della classe Particle crea un oggetto con un nome ed un impulso, inoltre se il nome dato non coincide con nessuno di quelli presenti nell'array fParticleType da un messaggio di errore. Sotto è presente il costruttore di default.
Nella sezione privata sono dichiarati i membri statici fMaxNumParticleType, fParticleType e fNParticleType che gestiscono un array di puntatori a oggetti di tipo ParticleType e il metodo FindParticle(), che restitisce l'indice del tipo di particella nell'array fParticleType dato il suo nome.
Altri membri privati sono l'indice della particella in fParticleType (fIndex) e le tre componenti dell'impulso (fPx, fPy, fPz). Questi possono essere sia settati, tramite dei setter, sia letti, tramite dei getter. Il setter di fIndex in particolare può prendere in input sia un int sia una stringa. Il metodo privato Boost() calcola le tre componeni dell'impulso delle particelle figlie del decadimento di k*.
Tra i membri pubblici i getter della massa e della carica della particella considerata, che utilizzano i

getter della classe ParticleType.
I metodi Energy() e InvMass() calcolano i valori dell'energia della particella e della massa invariante fra le particella considerata ed un'altra data in Input.
Ci sono poi tre metodi che permettono di aggiungere un oggetto all'array fParticleType e di stampare a schermo le caratteristiche degli elementi nell'array o le caratteristiche della particella considerata. I primi due metodi sono inoltre dichiarati statici.
L'ultimo metodo pubblico è Decay2body() che calcola le caratteristiche delle particelle create a seguito del decadimento di una particella k*.

3 Generazione

La parte di generazione delle particelle è contenuta nel file main.cpp. Sono stati impiegati il metodo di generazione Montecarlo e la classe di Root TRandom per generare 10^5 eventi ciascuno con 100 particelle.
All'interno della funzione Generate() è stato utilizzato il metodo AddParticle per aggiungere i tipi di particella presi in considerazione. Sono poi stati creati l'array EventParticles contenente le particelle generate in un ciclo e i vari istogrammi. L'array ha dimensione 120 per tenere conto anche delle figlie della particella k*.
La generazione delle particelle è stata gestita tramite un doppio loop for. Nel primo loop si pone la variabile n uguale a zero per assicurarsi che ad ogni loop si aggiungano le figlie della k* a partire dalla posizione cento dell'array EventParticles.
Nel secondo loop sono generati l'angolo azimutale θ e l'angolo polare ϕ secondo due distribuzioni uniformi, rispettivamente negli intervalli [0,2π] e [0,π] , e la quantità di moto secondo una distribuzione esponenziale negativa con media 1. Dopodichè il metodo SetP() imposta le tre componenti dell'impulso della particella generata usando le coordinate sferiche. In seguito viene generata secondo una distribuzione uniforme in [0,1] la variabile x. Questa è usata per definire le proporzioni secondo cui sono state generate le particelle. Le proporzioni sono le seguenti: pioni(+-)=80%, kaoni(+-)=10%, protoni(+-)=9% e k*=1%. Dal momento che la particella k* decade nelle coppie π^+k^- o π^-k^+ secondo una proporzionalità del 50%, viene chiamato il metodo Decay2Body() per calcolare e impostare le caratteristiche delle figlie.
Gli istogrammi sono stati riempiti rispettivamente con gli indici delle particelle, con l'angolo theta, con l'angolo phi, con l'impulso p, con l'impulso trasverso, con l'energia delle particelle (ottenuta con il metodo Energy()), con la massa invariante fra le figlie (calcolata con il metodo InvMass()) e con le masse invarianti tra le seguenti particelle:

- particelle di carica concorde;
- particelle di carica discorde;
- kaoni e pioni concordi;
- kaoni e pioni discordi;.

Infine vengono disegnati i grafici e salvati su un file chiamato generate.root.

4 Analisi

Il file analysis.cpp legge i dati scritti su generate.root e ricrea i grafici.
Sono state definite le funzioni con cui fare il fit sugli istogrammi degli angoli, dell'impulso, delle masse invarianti delle figlie di k* e delle differenze fra i seguenti grafici:

- Il grafico della massa invariante fra le particelle di carica discorde e quello fra le particelle di carica concorde;
- Il grafico della massa invariante fra pioni e kaoni di carica discorde e quello con la massa invariante di kaoni e pioni di carica concorde.

In particolare i due angoli hanno un fit uniforme rispettivamente in [0,2π] e [0,π], l'impulso ha un fit esponenziale con media 1 e la massa invariante delle figlie di k* e le due differenze hanno un fit gaussiano con media la massa di k* e sigma la sua vita media.

Per prima cosa, sono state confrontate le occorrenze osservate (figura 1) con quelle attese. Il risultato di tale confronto (Tabella 1) è evidenziaziato dalla parte finale del codice che stampa a schermo le percentuali di particelle effettivamente generate in un ciclo e le confronta con le percentuali aspettate, calcolando anche di quanti σ il valore reale dista da quello voluto. I risultati mostrano una corrispondenza entro gli errori.
Per quanto riguarda i fit relativi agli angoli e all'impulso (figura 1), il chi quadro ridotto (tabella 2), essendo prossimo all'unità, mostra un accordo tra i dati del fit e quelli della generazione.
Similmente avviene per i fit relativi alle masse invarianti (figura 2; Chi quadro ridotto in tabella 3). In particolare, sottraendo gli istogrammi, si è separato il segnale della risonanza di K* dal fondo di altre particelle. Nello specifico, la media della gaussiana tende alla massa della particella che decade, mentre la deviazione standard alla vita media della particella.

Specie	Occorrenze osservate	Occorrenze attese
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+	$(4.998 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
k-	$(5.002 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
p+	$(4.509 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
p-	$(4.501 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
k*	$(9.947 \pm 0.003) \cdot 10^4$	$1 \cdot 10^5$

Tabella 1: numero di particelle osservate e generate per ciascuna tipologia

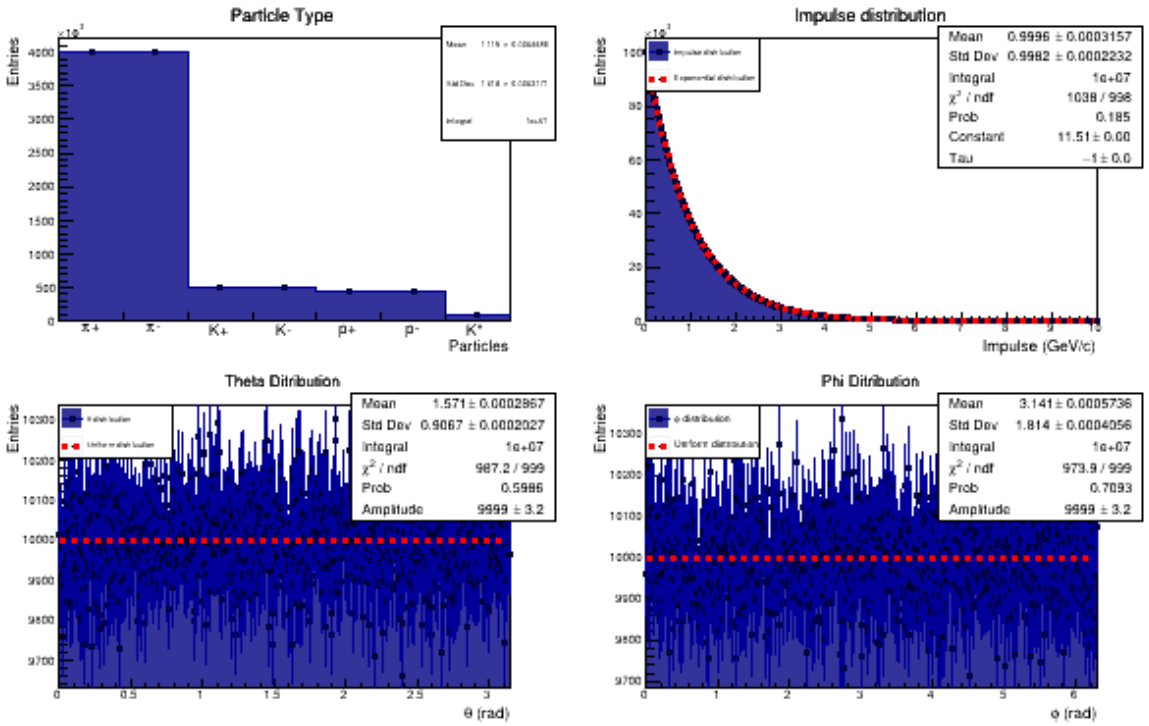

Figura 1: Da sinistra verso destra - in alto le occorrenze osservate per ciascun tipo di particella a fine generazione e la distribuzione dell'impulso, in basso le distribuzioni relative agli angoli theta e phi

Distribuzione	Parametri del Fit	χ^2	DOF	χ^2/DOF
Fit a distribuzione angolo θ (pol0)	9999 ± 3	987	999	0,99
Fit a distribuzione angolo ϕ (pol0)	9999 ± 3	974	999	0,99
Fit a distribuzione modulo impulso (expo)	11.5132 ± 0.0004	1038	998	1,04

Tabella 2: Risultati dei fit dei grafici relativi alle distribuzioni degli angoli e del modulo dell'impulso

Distribuzione e Fit	Media	Sigma	Ampiezza	χ^2/DOF
Massa Invariante vere K* (fit gauss)	0.8916 ± 0.0001	0.0498 ± 0.0001	$(7.95 \pm 0.03) \cdot 10^3$	0.90
Massa Invariante ottenuta da differenza delle combinazioni di carica discorde e concorde (fit gauss)	0.894 ± 0.005	0.049 ± 0.006	$(7.6 \pm 0.7) \cdot 10^3$	1.05
Massa Invariante ottenuta da differenza delle combinazioni πK di carica discorde e concorde (fit gauss)	0.894 ± 0.003	0.051± 0.003	$(8.2 \pm 0.4) \cdot 10^3$	0.90

Tabella 3: Risultati dei fit dei grafici relativi alle masse invarianti

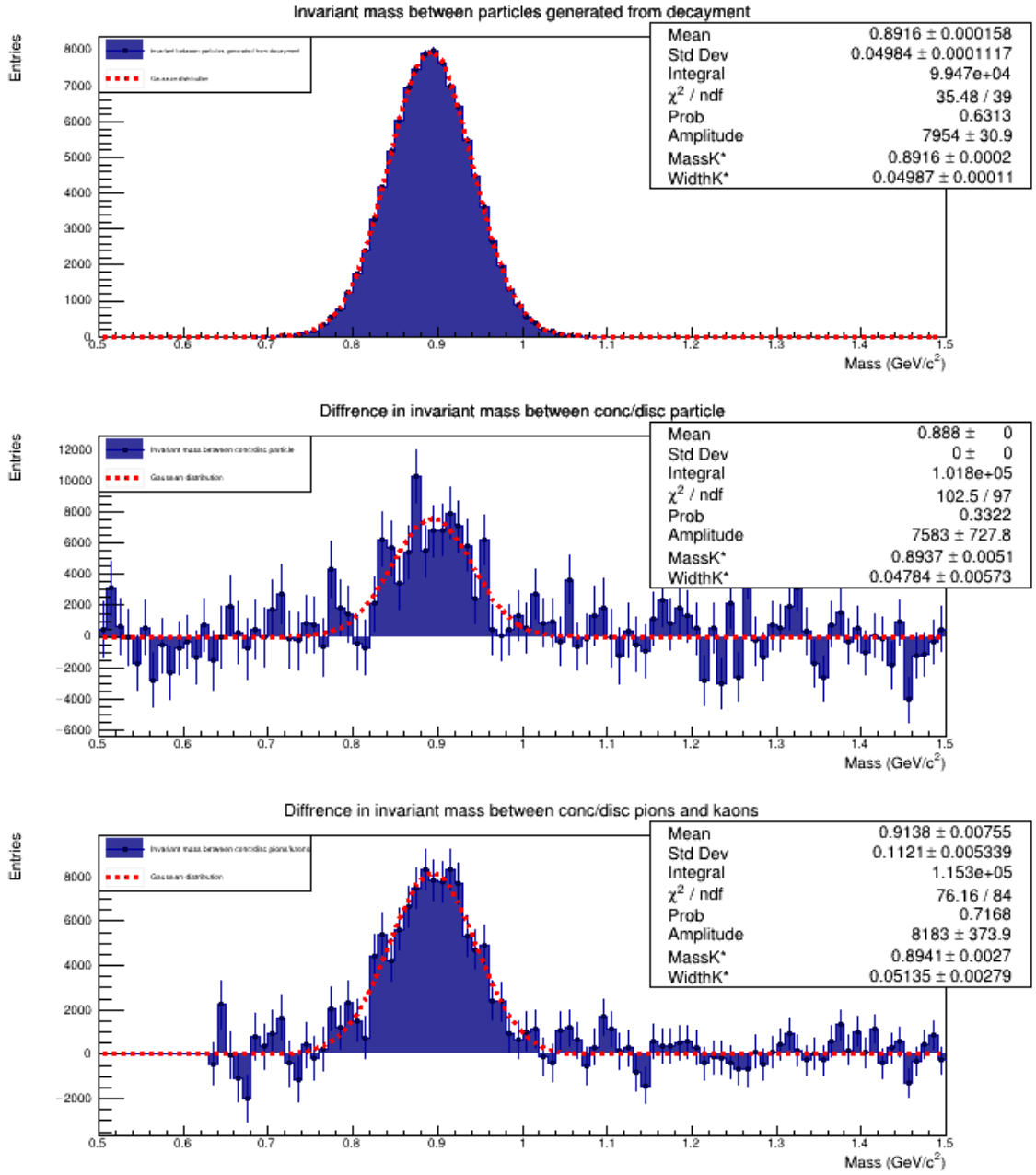

Figura 2: Dall'alto verso il basso istogrammi e relativi fit delle: masse invarianti delle figlie di k*, differenze di massa invariante fra le particelle con carica concorde e discorde, differenze di massa invariante fra πk con carica concorde e discorde, gaussiano

5 Appendice (codice)

5.1 particleType.hpp

```
#ifndef PARTICLETYPE_HPP
#define PARTICLETYPE_HPP

#include <iostream>

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

    std::string GetName() const;
    double GetMass() const;
    int GetCharge() const;
    virtual double GetWidth() const;

    virtual void Print() const;

private:
    const std::string fName{};
    const double fMass{};
    const int fCharge{};
};

#endif
```

5.2 particleType.cpp

```
#include "particleType.hpp"

ParticleType::ParticleType(const std::string name,const double mass,const int charge) : fName(name)
,fMass(mass), fCharge(charge) {};

std::string ParticleType::GetName() const {return fName; };

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

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

double ParticleType::GetWidth() const {return 0;};

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

5.3 resonanceType.hpp

```
#ifndef RESONANCETYPE_HPP
#define RESONANCETYPE_HPP

#include "particleType.hpp"

class ResonanceType : public ParticleType
{
public:
    ResonanceType(const std::string name, const double mass, const int charge, const double width);
    ResonanceType()=default;
    double GetWidth() const;
    void Print() const;

private:
    const double fWidth{};
```

```
};

#endif
```

5.4 resonanceType.cpp

```
#include "resonanceType.hpp"

ResonanceType::ResonanceType(const std::string 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 << "Widht: " << fWidth << "/n";
}
```

5.5 particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

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

class Particle
{
public:
    Particle(std::string name, const double px, const double py, const double pz);
    Particle() = default;
    int GetIndex() const;
    double GetPx() const;
    double GetPy() const;
    double GetPz() const;

    double GetMass() const;
    int GetCharge() const;
    double Energy() const;
    double InvMass(Particle &p) const;

    void SetIndex(const int index);
    void SetIndex(std::string name);
    void SetP(const double px, const double py, const double pz);

    static void AddParticleType(std::string name, const double mass, const int charge, const double
    width);
    static void PrintArray();
    void PrintParticle();

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

private:
    static const int fMaxNumParticleType = 10;
    static ParticleType* fParticleType[fMaxNumParticleType];
    static int fNParticleType;
    int fIndex{};
    double fPx{};
    double fPy{};
    double fPz{};

    static int FindParticle(std::string name);
    void Boost(double bx, double by, double bz);
};

#endif
```

5.6 particle.cpp

```
#include <cmath>
#include <cstdlib>
#include "particle.hpp"

ParticleType *Particle::fParticleType[Particle::fMaxNumParticleType];
int Particle::fNParticleType = 0;
```

```
Particle::Particle(const std::string name, const double px, const double py, const double pz) :
fIndex(FindParticle(name)), fPx(px), fPy(py), fPz(pz)
{
    if (FindParticle(name) == -1)
    {
        std::cout << "No correspondence found \n";
    };
};
int Particle::GetIndex() const { return fIndex; };
double Particle::GetPx() const { return fPx; };
double Particle::GetPy() const { return fPy; };
double Particle::GetPz() const { return fPz; };

double Particle::GetMass() const { return fParticleType[fIndex]->GetMass(); };
int Particle::GetCharge() const { return fParticleType[fIndex]->GetCharge(); }
double Particle::Energy() const
{
    double impulse_2 = fPx * fPx + fPy * fPy + fPz * fPz;
    double mass_ = fParticleType[fIndex]->GetMass();
    double energy = sqrt(mass_ * mass_ + impulse_2);
    return energy;
};

double Particle::InvMass(Particle &p) const
{
    double inv_mass = sqrt((Energy() + p.Energy()) * (Energy() + p.Energy()) -
                           ((fPx + p.fPx) * (fPx + p.fPx) + (fPy + p.fPy) * (fPy + p.fPy) +
                            (fPz + p.fPz) * (fPz + p.fPz)));
    return inv_mass;
};

void Particle::SetIndex(const int index) { fIndex = index; };
void Particle::SetIndex(std::string name) { fIndex = FindParticle(name); };
void Particle::SetP(const double px, const double py, const double pz)
{
    fPx = px;
    fPy = py;
    fPz = pz;
};

void Particle::AddParticleType(const std::string name, const double mass, const int charge, const
double width=0)
{
    if (fNParticleType < fMaxNumParticleType)
    {
        if (FindParticle(name) == -1)
        {
            if (width == 0)
            {
                ParticleType *p = new ParticleType(name, mass, charge);
                fParticleType[fNParticleType] = p;
                ++fNParticleType;
            }
            else
            {
                ResonanceType *p = new ResonanceType(name, mass, charge, width);
                fParticleType[fNParticleType] = p;
                ++fNParticleType;
            }
        }
    }
}

void Particle::PrintArray()
{

    for (int i = 0; i < fNParticleType; ++i)
    {
        fParticleType[i]->Print();
    };
};
void Particle::PrintParticle()
{
    std::cout << "Index: " << fIndex << "\n";
    std::cout << "Name: " << fParticleType[fIndex]->GetName() << "\n";
    std::cout << "Px: " << fPx << "\n";
```

```
    std::cout << "Py: " << fPy << "\n";
    std::cout << "Pz: " << fPz << "\n";
};

int Particle::FindParticle(std::string name)
{
    for (int i = 0; i < fNParticleType; ++i)
    {
        std::string nameP = fParticleType[i]->GetName();
        if (name == nameP)
            return i;
    };
    return -1;
};

int Particle::Decay2body(Particle &dau1, Particle &dau2) const
{
    if (GetMass() == 0.0)
    {
        printf("Decayment cannot be preformed if mass is zero\n");
        return 1;
    }

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

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

        // gaussian random numbers

        float x1, x2, w, y1;

        double invnum = 1. / RAND_MAX;
        do
        {
            x1 = 2.0 * 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;
}
void Particle::Boost(double bx, double by, double bz)
```

```
{

    double energy = Energy();

    // Boost this Lorentz vector
    double b2 = bx * bx + by * by + bz * bz;
    double gamma = 1.0 / sqrt(1.0 - b2);
    double bp = bx * fPx + by * fPy + bz * fPz;
    double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

    fPx += gamma2 * bp * bx + gamma * bx * energy;
    fPy += gamma2 * bp * by + gamma * by * energy;
    fPz += gamma2 * bp * bz + gamma * bz * energy;
}
```

5.7 main.cpp

```
#include "particleType.hpp"
#include "resonanceType.hpp"
#include "particle.hpp"
#include "TRandom.h"
#include "TH1.h"
#include "TH2.h"
#include "TCanvas.h"
#include "TFile.h"
#include "TBenchmark.h"

#include <cmath>

void Generate()
{
    gBenchmark->Start("With the generation");

    gRandom->SetSeed();
    Particle::AddParticleType("+", 0.13957, 1, 0);
    Particle::AddParticleType("-", 0.13957, -1, 0);
    Particle::AddParticleType("k+", 0.49367, 1, 0);
    Particle::AddParticleType("k-", 0.49367, -1, 0);
    Particle::AddParticleType("p+", 0.93827, 1, 0);
    Particle::AddParticleType("p-", 0.93827, -1, 0);
    Particle::AddParticleType("k*", 0.89166, 0, 0.050);

    Particle EventParticles[120];
    Double_t phi, theta, p, x, y, transverse_p = 0;
    Int_t n = 0;

    TH1F *h1 = new TH1F("h1", "Particle Type", 7, 0, 7);
    TH1F *h2 = new TH1F("h2", "Theta Ditribution", 1000, 0, M_PI);
    TH1F *h3 = new TH1F("h3", "Phi Ditribution", 1000, 0, 2 * M_PI);
    TH1F *h4 = new TH1F("h4", "Impulse distribution", 1000, 0, 10);
    TH1F *h5 = new TH1F("h5", "Transverse Impulse", 1000, 0, 10);
    TH1F *h6 = new TH1F("h6", "Energy", 1000, 0, 10);
    TH1F *h7 = new TH1F("h7", "Invariant mass between concordant charge particles", 100, 0.5, 1.5);
    TH1F *h8 = new TH1F("h8", "Invariant mass between discordant charge particles", 100, 0.5, 1.5);
    TH1F *h9 = new TH1F("h9", "Invariant mass between pion+/kaon- and pion-/kaon+", 100, 0.5, 1.5);
    TH1F *h10 = new TH1F("h10", "Invariant mass between pion+/kaon+ and pion-/kaon-", 100, 0.5, 1.5);
    TH1F *h11 = new TH1F("h11", "Invariant mass between particles generated from decayment", 100, 0.5,
    1.5);

    h6->Sumw2();
    h7->Sumw2();
    h8->Sumw2();
    h9->Sumw2();
    h10->Sumw2();

    for (Int_t i = 0; i < 1E5; ++i)
    {
        n = 0;

        for (Int_t j = 0; j < 100; ++j)
        {
            phi = gRandom->Uniform(0, 2 * M_PI);
            theta = gRandom->Uniform(0, M_PI);
            p = gRandom->Exp(1);

            x = gRandom->Rndm();
```

```
    EventParticles[j].SetP(p * sin(theta) * cos(phi), p * sin(theta) * sin(phi), p *
    cos(theta));
    if (x < 0.4)
    {
        EventParticles[j].SetIndex("+");
    }
    else if (x < 0.8)
    {
        EventParticles[j].SetIndex("-");
    }
    else if (x < 0.85)
    {
        EventParticles[j].SetIndex("k+");
    }
    else if (x < 0.90)
    {
        EventParticles[j].SetIndex("k-");
    }
    else if (x < 0.945)
    {
        EventParticles[j].SetIndex("p+");
    }
    else if (x < 0.99)
    {
        EventParticles[j].SetIndex("p-");
    }
    else
    {
        EventParticles[j].SetIndex("k*");
        y = gRandom->Rndm();
        if (y < 0.50)
        {
            EventParticles[100 + n].SetIndex("+");
            EventParticles[101 + n].SetIndex("k-");
        }
        else
        {
            EventParticles[100 + n].SetIndex("-");
            EventParticles[101 + n].SetIndex("k+");
        }

        EventParticles[j].Decay2body(EventParticles[100 + n], EventParticles[101 + n]);
        Double_t Invariant_Decay = EventParticles[100 + n].InvMass(EventParticles[101 + n]);
        h11->Fill(Invariant_Decay);
        n += 2;
    }
    transverse_p = sqrt((p * sin(theta) * cos(phi)) * (p * sin(theta) * cos(phi)) + (p *
    sin(theta) * sin(phi)) * (p * sin(theta) * sin(phi)));
    h1->Fill(EventParticles[j].GetIndex());
    h2->Fill(theta);
    h3->Fill(phi);
    h4->Fill(p);
    h5->Fill(transverse_p);
    h6->Fill(EventParticles[j].Energy());
}

for (Int_t f = 0; f < 100+n; ++f)
{
    if (EventParticles[f].GetMass() != 0 && EventParticles[f].GetIndex() != 7)
    {
        for (Int_t g = f + 1; g < 100+n; ++g)
        {
            if (EventParticles[f].GetCharge() * EventParticles[g].GetCharge() == 1)
            {
                Double_t Concordant_Invariant = EventParticles[f].InvMass(EventParticles[g]);
                h7->Fill(Concordant_Invariant);
            }
            else if (EventParticles[f].GetCharge() * EventParticles[g].GetCharge() == -1)
            {
                Double_t Discordant_Invariant = EventParticles[f].InvMass(EventParticles[g]);
                h8->Fill(Discordant_Invariant);
            }
        }

            for (Int_t g = f + 1; g < 100+n; ++g)
            {
```

```
                        if ((EventParticles[f].GetIndex() == 0 && EventParticles[g].GetIndex() == 3) ||
                        (EventParticles[f].GetIndex() == 3 && EventParticles[g].GetIndex() == 0) ||
                        (EventParticles[f].GetIndex() == 1 && EventParticles[g].GetIndex() == 2) ||
                        (EventParticles[f].GetIndex() == 2 && EventParticles[g].GetIndex() == 1))
                        {
                            Double_t Discordant_pk = EventParticles[f].InvMass(EventParticles[g]);
                            h9->Fill(Discordant_pk);
                        }
                        else if ((EventParticles[f].GetIndex() == 0 && EventParticles[g].GetIndex() ==
                        2) || (EventParticles[f].GetIndex() == 2 && EventParticles[g].GetIndex() == 0)
                        || (EventParticles[f].GetIndex() == 1 && EventParticles[g].GetIndex() == 3) ||
                        (EventParticles[f].GetIndex() == 3 && EventParticles[g].GetIndex() == 1))
                        {
                            Double_t Concordant_pk = EventParticles[f].InvMass(EventParticles[g]);
                            h10->Fill(Concordant_pk);
                        }
                    }

            }
        }
    }
    TCanvas *c1 = new TCanvas();
    h1->Draw();
    TCanvas *c2 = new TCanvas();
    h2->Draw();
    TCanvas *c3 = new TCanvas();
    h3->Draw();
    TCanvas *c4 = new TCanvas();
    h4->Draw();
    TCanvas *c5 = new TCanvas();
    h5->Draw();
    TCanvas *c6 = new TCanvas();
    h6->Draw();
    TCanvas *c7 = new TCanvas();
    h7->Draw();
    TCanvas *c8 = new TCanvas();
    h8->Draw();
    TCanvas *c9 = new TCanvas();
    h9->Draw();
    TCanvas *c10 = new TCanvas();
    h10->Draw();
    TCanvas *c11 = new TCanvas();
    h11->Draw();

    TFile *file = new TFile("generate.root", "recreate");

    h1->Write("h1");
    h2->Write("h2");
    h3->Write("h3");
    h4->Write("h4");
    h5->Write("h5");
    h6->Write("h6");
    h7->Write("h7");
    h8->Write("h8");
    h9->Write("h9");
    h10->Write("h10");
    h11->Write("h11");

    file->Close();

    gBenchmark->Show("With the generation");

}
```

5.8 analysis.cpp

```
#include "TFile.h"
#include "TH1.h"
#include "TStyle.h"
#include "TLegend.h"
#include "TBenchmark.h"

#include <string>
#include <array>
```

```
void Analysis()
{
    gBenchmark->Start("With the analysis");

    TH1::AddDirectory(kFALSE);
    TFile *file1 = new TFile("generate.root", "READ");

    // Creating ROOT File
    TFile *file2 = new TFile("analysis.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");

    TH1F *hDiff1 = new TH1F(*h7);
    hDiff1->Add(h8, h7, 1, -1);
    hDiff1->SetTitle("Diffrence in invariant mass between conc/disc particle");

    TH1F *hDiff2 = new TH1F(*h9);
    hDiff2->Add(h9, h10, 1, -1);
    hDiff2->SetTitle("Diffrence in invariant mass between conc/disc pions and kaons");

    std::array<TH1F *, 13> histos{h1, h2, h3, h4, h5, h6, h7,
                                  h8, h9, h10, h11, hDiff1, hDiff2};

    for (TH1F *h : histos)
    {
        h->SetMarkerStyle(25);
        h->SetMarkerSize(0.25);
        h->SetMarkerColor(kBlue + 4);
        h->SetLineColor(kBlue + 2);
        h->SetFillColor(kBlue - 2);
        h->GetYaxis()->SetTitleOffset(1.1);
        h->GetXaxis()->SetTitleSize(0.045);
        h->GetYaxis()->SetTitleSize(0.045);
        h->GetYaxis()->SetTitle("Entries");
        gStyle->SetOptStat(1002200);
        gStyle->SetOptFit(1111);
    }

    TF1 *f1 = new TF1("theta", "pol0", 0., TMath::Pi());
    TF1 *f2 = new TF1("phi", "pol0", 0., 2 * TMath::Pi());
    TF1 *f3 = new TF1("f3", "expo", 0, 10);
    TF1 *f4 = new TF1("f4", "gaus", 0, 2);
    TF1 *f5 = new TF1("f5", "gaus", 0, 2);
    TF1 *f6 = new TF1("f6", "gaus", 0, 2);

    std::array<TF1 *, 6> fits{f1, f2, f3, f4, f5, f6};
    for (TF1 *f : fits)
    {
        f->SetLineColor(kRed);
        f->SetLineWidth(3);
        f->SetLineStyle(2);
    }

    f4->SetParameter(1, 0.89166);
    f4->SetParameter(2, 0.050);
    f5->SetParameter(1, 0.89166);
    f5->SetParameter(2, 0.050);
    f6->SetParameter(1, 0.89166);
    f6->SetParameter(2, 0.050);

    f1->SetParNames("Amplitude");
    f2->SetParNames("Amplitude");
    f3->SetParNames("Constant", "Tau");
    f4->SetParNames("Amplitude", "MassK*", "WidthK*");
    f5->SetParNames("Amplitude", "MassK*", "WidthK*");
```

```
f6->SetParNames("Amplitude", "MassK*", "WidthK*");

h2->Fit(f1);
h3->Fit(f2);
h4->Fit(f3);
hDiff1->Fit(f4);
hDiff2->Fit(f5);
h11->Fit(f6);

TLegend *leg1 = new TLegend(0.1, 0.75, 0.3, 0.9);
TLegend *leg2 = new TLegend(0.1, 0.75, 0.3, 0.9);
TLegend *leg3 = new TLegend(0.1, 0.75, 0.3, 0.9);
TLegend *leg4 = new TLegend(0.1, 0.75, 0.3, 0.9);
TLegend *leg5 = new TLegend(0.1, 0.75, 0.3, 0.9);

leg1->AddEntry(h2, "#theta distribution");
leg1->AddEntry(f1, "Uniform distribution");

leg2->AddEntry(h3, "#phi distribution");
leg2->AddEntry(f2, "Uniform distribution");

leg3->AddEntry(h4, "Impulse distribution");
leg3->AddEntry(f3, "Exponential distribution");

leg4->AddEntry(hDiff1, "Invariant mass between conc/disc particle");
leg4->AddEntry(f4, "Gaussian distribution");

leg5->AddEntry(hDiff2, "Invariant mass between conc/disc pions/kaons");
leg5->AddEntry(f5, "Gaussian distribution");

TCanvas *c1 = new TCanvas("c1", "Particles, angles, impulse distributions", 200, 10, 600, 400);
c1->Divide(2, 2);

c1->cd(1);
h1->GetXaxis()->SetBinLabel(1, "#pi+");
h1->GetXaxis()->SetBinLabel(2, "#pi-");
h1->GetXaxis()->SetBinLabel(3, "K+");
h1->GetXaxis()->SetBinLabel(4, "K-");
h1->GetXaxis()->SetBinLabel(5, "p+");
h1->GetXaxis()->SetBinLabel(6, "p-");
h1->GetXaxis()->SetBinLabel(7, "K*");
h1->GetXaxis()->SetLabelSize(0.07);
h1->GetXaxis()->SetTitle("Particles");
h1->Draw("H");
h1->Draw("E,P,SAME");

c1->cd(2);

h4->GetXaxis()->SetTitle("Impulse (GeV/c)");
h4->Draw("H");
h4->Draw("E,P,SAME");
leg3->Draw("SAME");

c1->cd(3);
h2->GetXaxis()->SetTitle("#theta (rad)");
h2->Draw("H");
h2->Draw("E,P,SAME");
leg1->Draw("SAME");

c1->cd(4);
h3->GetXaxis()->SetTitle("#phi (rad)");
h3->Draw("H");
h3->Draw("E,P,SAME");
leg2->Draw("SAME");

TCanvas *c2 = new TCanvas("c2", "Transverse Impulse", 200, 10, 600, 400);
h5->GetXaxis()->SetTitle("Transverse Impulse (GeV/c)");
h5->Draw("H");
h5->Draw("E,P,SAME");

TCanvas *c3 = new TCanvas("c3", "Energy", 200, 10, 600, 400);
h6->GetXaxis()->SetTitle("Energy (GeV)");
h6->Draw("H");
h6->Draw("E,P,SAME");

TCanvas *c4 = new TCanvas("c4", "Invariant mass between concordant charge particles", 200, 10, 600,
400);
```

```
h7->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
h7->Draw("H");
h7->Draw("E,P,SAME");

TCanvas *c5 = new TCanvas("c5", "Invariant mass between discordant charge particles", 200, 10, 600,
400);
h8->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
h8->Draw("H");
h8->Draw("E,P,SAME");

TCanvas *c6 = new TCanvas("c6", "Invariant mass between pion+/kaon- and pion-/kaon+", 200, 10, 600,
400);
h9->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
h9->Draw("H");
h9->Draw("E,P,SAME");

TCanvas *c7 = new TCanvas("c7", "Invariant mass between pion+/kaon+ and pion-/kaon-", 200, 10, 600,
400);
h10->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
h10->Draw("H");
h10->Draw("E,P,SAME");

TCanvas *c8 = new TCanvas("c8", "Invariant mass", 200, 10, 700, 800);
c8->Divide(1, 3);

c8->cd(1);

h11->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
h11->Draw("H");
h11->Draw("E,P,SAME");

c8->cd(2);

hDiff1->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
hDiff1->Draw("H");
hDiff1->Draw("E,P,SAME");
leg4->Draw("SAME");

c8->cd(3);

hDiff2->GetXaxis()->SetTitle("Mass (GeV/c^{2})");
hDiff2->Draw("H");
hDiff2->Draw("E,P,SAME");
leg5->Draw("SAME");

c1->Print("particles.png");
c8->Print("invmass.png");

h1->Write("h1");
h2->Write("h2");
h3->Write("h3");
h4->Write("h4");
h5->Write("h5");
h6->Write("h6");
h7->Write("h7");
h8->Write("h8");
h9->Write("h9");
h10->Write("h10");
h11->Write("h11");
hDiff1->Write("hDiff1");
hDiff2->Write("hDiff2");

file2->Close();

std::cout << "\nTheta distribution: \n";
std::cout << "Parameter: " << f1->GetParameter(0) << " +- " << f1->GetParError(0) << "\n";
std::cout << "Chi square: " << f1->GetChisquare() << " NDF: " << f1->GetNDF() << "\n";
std::cout << "Probability: " << f1->GetProb() << "\n";

std::cout << "\nPhi distribution: \n";
std::cout << "Parameter: " << f2->GetParameter(0) << " +- " << f2->GetParError(0) << "\n";
std::cout << "Chi square: " << f2->GetChisquare() << " NDF: " << f2->GetNDF() << "\n";
std::cout << "Probability: " << f2->GetProb() << "\n";

std::cout << "\nImpulse distribution: \n";
std::cout << "Parameter: " << f3->GetParameter(0) << " +- " << f3->GetParError(0) << "\n";
```

```
std::cout << "Chi square: " << f3->GetChisquare() << " NDF: " << f3->GetNDF() << "\n";
std::cout << "Probability: " << f3->GetProb() << "\n";

std::cout << "\nGaus distribution of invariant mass between particles generated from decayment:
\n";
std::cout << "Parameter : " << f6->GetParameter(0) << " +- " << f6->GetParError(0) << "\n";
std::cout << "Mean: " << f6->GetParameter(1) << " +- " << f6->GetParError(1) << "\n";
std::cout << "Std dev : " << f6->GetParameter(2) << " +- " << f6->GetParError(2) << "\n";
std::cout << "Chi square: " << f6->GetChisquare() << " NDF: " << f6->GetNDF() << "\n";
std::cout << "Probability: " << f6->GetProb() << "\n";

std::cout << "\nGaus distribution of invariant mass between conc and disc: \n";
std::cout << "Parameter : " << f4->GetParameter(0) << " +- " << f4->GetParError(0) << "\n";
std::cout << "Mean: " << f4->GetParameter(1) << " +- " << f4->GetParError(1) << "\n";
std::cout << "Std dev : " << f4->GetParameter(2) << " +- " << f4->GetParError(2) << "\n";
std::cout << "Chi square: " << f4->GetChisquare() << " NDF: " << f4->GetNDF() << "\n";
std::cout << "Probability: " << f4->GetProb() << "\n";

std::cout << "\nGaus distribution of invariant mass between pion and kaon: \n";
std::cout << "Parameter : " << f5->GetParameter(0) << " +- " << f5->GetParError(0) << "\n";
std::cout << "Mean: " << f5->GetParameter(1) << " +- " << f5->GetParError(1) << "\n";
std::cout << "Std dev : " << f5->GetParameter(2) << " +- " << f5->GetParError(2) << "\n";
std::cout << "Chi square: " << f5->GetChisquare() << " NDF: " << f5->GetNDF() << "\n";
std::cout << "Probability: " << f5->GetProb() << "\n";

std::cout << "\nParticles distribution: \n";
std::cout << "\nNumber of particles: " << h1->GetEntries() << ";\n\n";
for (int i = 1; i < 8; ++i)
{
    std::string type;
    double percentage;
    if (i == 1)
    {
        type = "+";
        percentage = 40;
    }
    else if (i == 2)
    {
        type = "-";
        percentage = 40;
    }
    else if (i == 3)
    {
        type = "k+";
        percentage = 5;
    }
    else if (i == 4)
    {
        type = "k-";
        percentage = 5;
    }
    else if (i == 5)
    {
        type = "p+";
        percentage = 4.5;
    }
    else if (i == 6)
    {
        type = "p-";
        percentage = 4.5;
    }
    else if (i == 7)
    {
        type = "k*";
        percentage = 1;
    }
    std::cout << "Number of " << type << ": " << h1->GetBinContent(i) << "; Error: " <<
    h1->GetBinError(i) << ";\n";
    double quantity = ((h1->GetBinContent(i)) / (h1->GetEntries())) * 100;
    double error = ((h1->GetBinError(i)) / (h1->GetEntries())) * 100;
    double k = (quantity - percentage) / error;
    std::cout << "Percentage of " << type << ": " << quantity << " +- " << error;
    if (quantity - error < percentage && percentage < quantity + error)
    {
        std::cout << " as expected;\n";
    }
```

```
        else
        {
            std::cout << " not as expected (Real value is distant from the expected value by " << k <<
            "\n";
        }
    }

    gBenchmark->Show("With the analysis");
}
```