Relazione laboratorio 2

Modulo C++/Root

Chiara Vece - a.a. 2018/2019

1 Introduzione

Lo scopo dell'esperienza è quello di simulare, attraverso Root, gli eventi fisici legati al decadimento delle particelle K* e alla collisione di particelle elementari quali protoni, pioni e kaoni.

2 Struttura del codice

Il codice è organizzato in tre classi principali e un Main Module. Nella prima classe, denominata ParticleType, sono implementate le proprietà fisiche delle particelle elementari quali massa, carica e nome. Nella seconda classe, ResonanceType, erede di ParticleType, è presente Width ovvero la risonanza. L'ultima classe, denominata Particle, non eredita da ParticleType e ResonanceType. Attraverso il reimpiego di codice per composizione e grazie a un array di puntatori statico a ParticleType che funge da tabella per descrivere le proprietà caratteristiche del tipo di particella la classe Particle include degli oggetti di tipo ParticleType e ResonanceType. La scelta è ricaduta sulla composizione piuttosto che sull'ereditarietà perchè è più dinamica, ogni oggetto può essere rimpiazzato al run-time con un altro dello stesso tipo, evita la costruzione di classi di difficile gestione e permette il risparmio di memoria. La classe Particle inoltre aggiunge delle proprietà cinematiche come le componenti dell'impulso della particella, assunte nell'istante successivo all'urto. In Particle è implementato il metodo statico AddParticle, con cui vengono inserite le grandezze fondamentali dichiarate nelle classi precedenti nell'array di puntatori. Nella medesima classe vi è il metodo FindParticle anch'esso statico che, ricevuto il nome della particella, la cerca nell'array e se questa è presente ne returna l'indice, altrimenti returna -1. Sono implementati anche un metodo per il calcolo dell'energia e un altro per il calcolo della massa invariante. Infine sono presenti metodi utili quali Getter e Setter per nome, massa, carica, indice e componenti dell'impulso.

3 Generazione

Nel Main Module tramite due cicli for annidati sono stati creati 10^5 eventi e in ogni evento vengono generate 100 particelle secondo proporzioni e caratteristiche elencate nella tabella sottostante:

Nome	Massa (GeV/c^2)	Carica	Risonanza(GeV/c^2)	Quantità generata
Pione(+)	0.13957	+1	-	0.400
Pione(-)	0.13957	-1	-	0.400
Kaone(+)	0.49367	+1	-	0.050
Kaone(-)	0.49367	-1	-	0.050
Protone(+)	0.93827	+1	-	0.045
Protone(-)	0.93827	-1	-	0.045
K*	0.89166	0	0.050	0.010

Per la generazione sono stati utilizzati i metodi Monte Carlo di ROOT come TRandom::Uniform(),TRandom::Exp(double mean). Per le particelle si è estratto un numero compreso tra 0 e 1 da una distribuzione uniforme, assegnando con degli "if - else if - else" un indice a ogni particella generata entro i range già elencati. Per quanto riguarda le proprietà cinematiche, l'impulso è stato generato mediante una distribuzione esponenziale, mentre gli angoli polare e azimutale con una distribuzione uniforme.

4 Analisi

Di seguito la tabella con i risultati attesi e quelli osservati per la generazione delle particelle secondo definite proporzioni. Nella prima colonna le specie di particelle generate, nella seconda le occorrenze osservate con i relativi errori ottenuti grazie ai metodi per gli istogrammi *GetBinContent()* e *GetBinError()*. Dall'analisi di questi dati possiamo osservare che gli eventi attesi e quelli osservati sono compatibili entro i limiti dell'errore statistico.

Specie	**Occorrenze osservate**	**Occorrenze attese**
π+	$(4000 \pm 2) \cdot 10^3$	$40 \cdot 10^5$
π-	$(39996 \pm 20) \cdot 10^2$	$40 \cdot 10^5$
K+	$(5008 \pm 7) \cdot 10^2$	$5 \cdot 10^5$
K-	$(4985 \pm 7) \cdot 10^2$	$5 \cdot 10^5$
P+	$(4509 \pm 7) \cdot 10^2$	$4.5 \cdot 10^5$
P-	$(4498 \pm 7) \cdot 10^2$	$4.5 \cdot 10^5$
K*	$(998 \pm 3) \cdot 10^2$	10^5

Gli angoli θ e ϕ sono stati generati con delle distribuzioni uniformi e attraverso un fit se ne verifica la consistenza. Si riportano di seguito il "*par[0]*" del fit, e il χ^2 ridotto, dal quale si evince la consistenza del fit. Per l'impulso, invece, verifichiamo la consistenza con un fit esponenziale e ne riportiamo la media ovvero "*par[1]*".

Distribuzione	**Parametri del fit**	χ^2	**DOF**	χ^2/DOF
Fit angolo θ	9999.0 $\pm$ 3.16	1020.94	999	1.02
Fit angolo ϕ	9999.0 $\pm$ 3.16	1030.69	999	1.03
Fit modulo impulso	$(9900 \pm 3) \cdot 10^{-4}$	1168.79	997	1.17

Nella tabella che segue sono riportati i valori ottenuti in seguito alla combinazione dei vari istogrammi di massa invariante. Nella prima riga vi sono i risultati dell'istogramma contenente solo i decadimenti della k*. La posizione del picco della gaussiana è legata alla massa della particella che decade, come si può notare dalla compatibilità tra la media della gaussiana e la massa della k* entro il limite dell'errore statistico. Nelle ultime righe sono riportati rispettivamente i risultati inerenti ai grafici sottrazione *"πk discordi - πk concordi"* e *"carica discorde - carica concorde"*, fittati entrambi con la funzione "gaus" di ROOT. Dal risultato di queste combinazioni possiamo osservare un picco in corrispondenza della risonanza k*, entro il limite d'errore.

Distribuzione	**Media**	**Sigma**	**Ampiezza**	χ^2/DOF
K*	0.8919 ± 0.0001	0.0500 ± 0.0001	955 ± 4	0.94
Subtract	0.78 ± 0.03	0.36 ± 0.02	$(1.9 \pm 0.9) \cdot 10^3$	1.1
Subtract2	0.471 ± 0.303	0.90 ± 0.20	$(4.8 \pm 0.4) \cdot 10^3$	1.15

4.1 Grafici

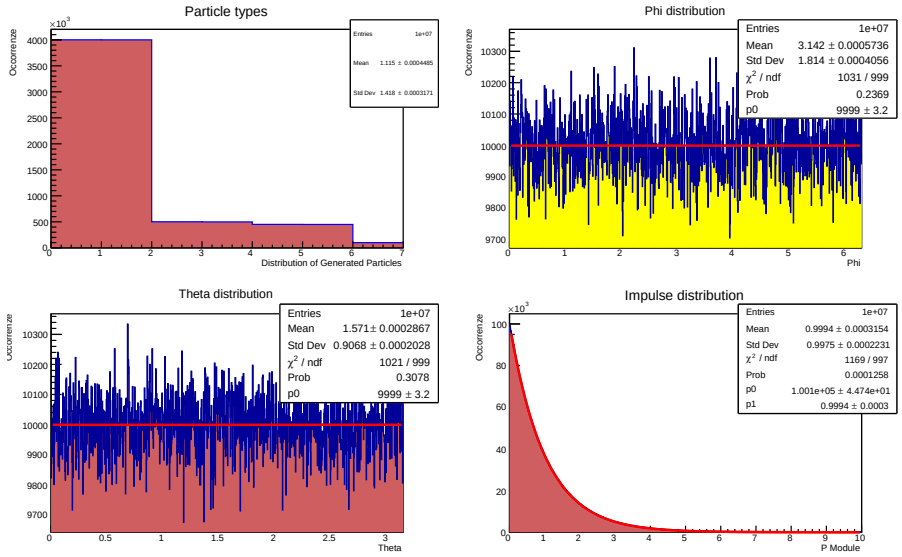

Figure 1: Particle types, phi, theta, impulse

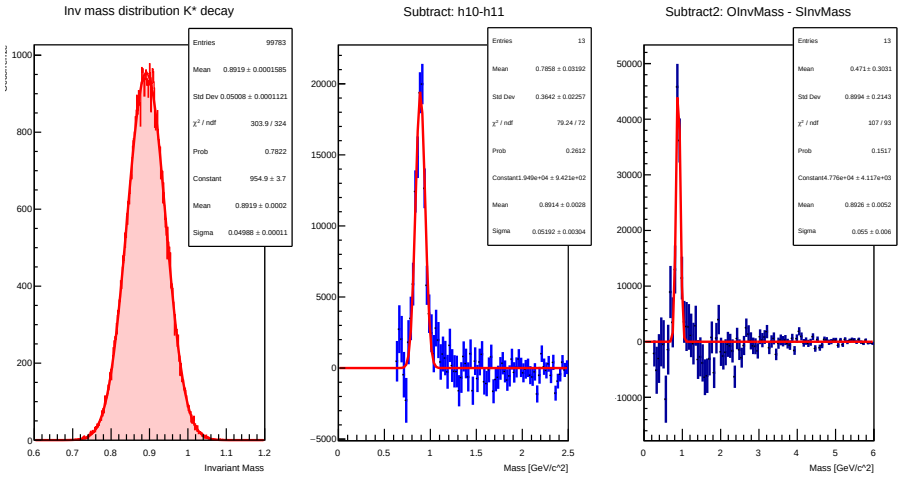

Figure 2: k*, subtract, subtract2

5 Appendice

"ParticleType.h"

```
#ifndef PARTICLETYPE_H
#define PARTICLETYPE_H

class ParticleType {
public:
ParticleType (const char* name, double mass, int charge);
//Dichiarazione e implementazione dei "getters"
const char* GetName () const { return fName; }
double GetMass () const { return fMass; }
int GetCharge () const { return fCharge; }
virtual void print (bool endline=true);
virtual double GetWidth () const { return 0; };
private:
const char* fName;
const double fMass;
const int fCharge;
};

#endif
```

"ParticleType.cpp"

```
#include <iostream>
#include "ParticleType.h"
using namespace std;

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

void ParticleType::print (bool endline) {
cout << "< Object 'ParticleType': name=\"" << fName << "\" mass=" << fMass <<
 " charge=" << fCharge << ">";
if (endline)
cout << endl;
}
```

"ResonanceType.h"

```
#ifndef RESONANCETYPE_H
#define RESONANCETYPE_H
#include "ParticleType.h"

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

```
double GetWidth () const { return fWidth; }
void print (bool endline=true);
private:
const double fWidth;
};

#endif
```

"ResonanceType.cpp"

```
#include <iostream>
#include "ResonanceType.h"
using namespace std;

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

ResonanceType::ResonanceType (ParticleType particle, double width) :
ParticleType(particle.GetName(), particle.GetMass(), particle.GetCharge()),
fWidth(width) {}

void ResonanceType::print (bool endline) {
cout << "<Object 'ResonanceType': name=\"" << GetName() << "\" mass=" << GetMass() <<
" charge=" << GetCharge() << " width=" << fWidth << ">";
if (endline)
cout << endl;
}
```

"Particle.h"

```
#ifndef PARTICLE_H
#define PARTICLE_H
#include "ParticleType.h"
#include "ResonanceType.h"

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

int GetIParticle () const { return fIParticle; }
void SetIParticle (int index);
void SetIParticle (const char* name);

int GetPx () const { return fPx; }
int GetPy () const { return fPy; }
int GetPz () const { return fPz; }
void SetP (double px, double py, double pz);

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

void print () const;
const char* GetName () const;
double GetMass () const;
int GetCharge () const;
double GetEnergy () const;
double InvMass (Particle& p) const;

int decay2body(Particle &dau1, Particle &dau2) const;
void boost(double bx, double by, double bz);
private:
static const int fMaxNumParticleType = 10;
static ParticleType* fParticleType[fMaxNumParticleType];
static int fNParticleType;
int fIParticle;
double fPx, fPy, fPz;
static int FindParticle (const char* name);
};

#endif
```

"Particle.cpp"

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include "Particle.h"
```

```
using namespace std;

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

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


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

//costruttori

Particle::Particle () : fIParticle(-1), fPx(0), fPy(0), fPz(0) {}
Particle::Particle (const char* name, double px, double py, double pz) : fPx(px),
fPy(py), fPz(pz) {
int index = FindParticle(name);
if (index == -1)
cout << "No particle found with this name." << endl;
else
fIParticle = index;
}

void Particle::SetIParticle (int index) {
if (index < fNParticleType)
fIParticle = index;
else
cout << "Particle ID not valid." << endl;
}

void Particle::SetIParticle (const char* name) {
int index = FindParticle(name);
if (index == -1)
cout << "Partice type name not valid." << endl;
else
fIParticle = index;
}
```

```
void Particle::printParticleTypes () {
cout << "index\tobject" << endl;
for (int i = 0; i < fNParticleType; i ++) {
cout << i << '\t';
fParticleType[i]->print();
}
}

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

void Particle::print () const {
cout << "Index: " << fIParticle << ", name: " << fParticleType[fIParticle]->GetName()
 <<", P = (" << fPx << ", " << fPy << ", " << fPz << ")" << endl;
}
//Getters
const char* Particle::GetName() const {
return fParticleType[fIParticle]->GetName();
}

double Particle::GetMass () const {
return fParticleType[fIParticle]->GetMass();
}

int Particle::GetCharge () const {
return fParticleType[fIParticle]->GetCharge();
}

double Particle::GetEnergy () const {
return sqrt(pow(GetMass(), 2) + fPx*fPx + fPy*fPy + fPz*fPz);
}

//metodo per il calcolo della massa invariante
double Particle::InvMass (Particle& particle) const {
double e = pow(GetEnergy() + particle.GetEnergy(), 2);
double p = pow(fPx + particle.fPx, 2) + pow(fPy + particle.fPy, 2) +
pow(fPz + particle.fPz, 2);
return sqrt(e - p);
}

int Particle::decay2body (Particle &dau1, Particle &dau2) const {
if (GetMass() == 0.0) {
cout << "Decayment cannot be preformed if mass is zero." << endl;
return 1;
}
```

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

if (fIParticle > -1) {
float x1, x2, w, y1, y2;
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;
y2 = x2 * w;

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

if(massMot < massDau1 + massDau2){
cout << "Decayment cannot be preformed because mass is too low in this channel." << endl;
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 = GetEnergy();
```

```
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;
}
```

"Main.cpp"

```
#include "ParticleType.h"
#include "ResonanceType.h"
#include "Particle.h"
#include "TRandom.h"
#include "TFile.h"
#include "TH1.h"
#include "TH2.h"
#include "TF1.h"
#include "TStyle.h"
#include "TCanvas.h"
#include "TMath.h"
#include "TStyle.h"
#include "TLegend.h"
#include <iostream>

using namespace std;
void generate () {


Particle::AddParticleType("pione+",0.13957,1.0);
Particle::AddParticleType("pione-",0.13957,-1.0);
Particle::AddParticleType("kaone+",0.49367,1.0);
Particle::AddParticleType("kaone-",0.49367,-1.0);
Particle::AddParticleType("protone+",0.93827,1.0);
Particle::AddParticleType("protone-",0.93827,-1.0);
Particle::AddParticleType("k*",0.89166,0,0.050);



TFile* file = new TFile("histograms.root", "recreate");
//creazione degli istogrammi e impostazioni di cosmetica
//Particle types

TH1F* TypeParticles = new TH1F("Particle types", "Particle types", 7, 0, 7);
TypeParticles -> SetFillColor(46);
TypeParticles->SetLineColor(kBlue);
TypeParticles->SetMarkerStyle(20);
TypeParticles-> GetXaxis()-> SetTitle("Distribution of Generated Particles");
```

```
TypeParticles-> GetYaxis()-> SetTitle("Occorrenze");

//Phi Distribution

TH1F* PhiDistribution = new TH1F("Phi distribution",
 "Phi distribution",1000, 0, 2*M_PI);
PhiDistribution -> SetFillColor(5);
PhiDistribution -> GetXaxis()-> SetTitle("Phi");
PhiDistribution -> GetYaxis()-> SetTitle("Occorrenze");

//Theta Distribution

TH1F* ThetaDistribution = new TH1F("Theta distribution", "Theta distribution",
1000, 0, M_PI);
ThetaDistribution -> SetFillColor(46);
ThetaDistribution -> GetXaxis() -> SetTitle("Theta");
ThetaDistribution -> GetYaxis() -> SetTitle("Occorrenze");

//Impulse Distribution

TH1F* PDistribution = new TH1F("Impulse distribution",
"Impulse distribution", 1000, 0,10);
PDistribution->SetFillColor(46);
PDistribution -> GetXaxis()-> SetTitle("P Module");
PDistribution -> GetYaxis()-> SetTitle("Occorrenze");

//Transverse P

TH1F* TransverseP = new TH1F("Transverse impulse distribution",
"Transverse impulse distribution", 1000, 0, 5);
TransverseP -> SetFillColor(46);
TransverseP -> GetXaxis() -> SetTitle("Transverse P");
TransverseP -> GetYaxis() -> SetTitle("Occorrenze");

//Energy

TH1F* Energy = new TH1F("Energy distribution", "Energy distribution",1000, 0, 6);
Energy -> SetFillColor(46);
Energy -> GetXaxis() -> SetTitle("Energia delle particelle generate");
Energy -> GetYaxis() -> SetTitle("Occorrenze");

//Inv Mass

TH1F* InvMass = new TH1F("Inv mass distribution", "Inv mass distribution",80, 0, 2.5);
InvMass -> SetFillColor(46);
InvMass -> GetXaxis() -> SetTitle("Mass [GeV/c^2]");
InvMass -> GetYaxis() -> SetTitle("Occorrenze");

//Inv Mass discordant charge
```

```
TH1F* OInvMass = new TH1F("Inv mass discordant charge ",
"Inv mass discordant charge",100, 0,6 );
OInvMass -> SetFillColor(46);
OInvMass -> GetXaxis() -> SetTitle("Invariant Mass ");
OInvMass -> GetYaxis() -> SetTitle("Occorrenze");

//Inv Mass same charge

TH1F* SInvMass = new TH1F("Inv mass concordant charge ",
"Inv mass concordant charge",100, 0,6 );
SInvMass -> SetFillColor(46);
SInvMass -> GetXaxis() -> SetTitle("Invariant Mass");
SInvMass -> GetYaxis() -> SetTitle("Occorrenze");

//Invariant Mass for pi+/k- and pi-/K+ Particles

TH1F* h10 = new TH1F("Inv mass distribution (Pione-Kaone discordant) ",
"Inv mass (Pione-Kaone discordant) ",100, 0,2.5);
h10 -> SetLineColor(kRed);
h10 -> GetXaxis() -> SetTitle("Invariant Mass");
h10 -> GetYaxis() -> SetTitle("Occorrenze");

//Invariant Mass for pi+/k+ and pi-/K- Particles

TH1F* h11 = new TH1F("Inv mass distribution (Pione-Kaone concordant) ",
"Inv mass distribution (Pione-Kaone concordant)",100, 0, 2.5 );
h11 -> SetLineColor(kRed);
h11 -> GetXaxis() -> SetTitle("Invariant Mass");
h11 -> GetYaxis() -> SetTitle ("Occorrenze");

//Invariant Mass for Decay

TH1F* HDecay = new TH1F("Inv mass distribution K* decay",
 "Inv mass distribution K* decay",500, 0.6, 1.2);
HDecay -> SetLineColor(kRed);
HDecay -> GetXaxis()-> SetTitle("Invariant Mass");
HDecay -> GetYaxis()-> SetTitle("Occorrenze");


//Fine creazione grafici e cosmetica




Particle particles [120];
int k, decaycounter;
//1° for per la creazione di 10^5 eventi e 2° per la creazione delle 100 particelle
for (int i= 0; i < 100000; i++){
k=99; //posizione delle particelle decadute nell'array
```

```
decaycounter=0; //contatore per le figlie delle k*decadute
for(int j = 0; j < 100; j++) {
double phi = gRandom -> Uniform (0, 2*M_PI);
PhiDistribution->Fill(phi);
double theta = gRandom -> Uniform (0, M_PI);
ThetaDistribution->Fill(theta);
double P = gRandom -> Exp(1);

//grazie alle coordinate cilindriche trasformo p in px, py, pz
double Px = P*sin(theta)*cos(phi);
double Py = P*sin(theta)*sin(phi);
double Pz = P*cos(theta);
PDistribution->Fill(sqrt(pow(Px,2)+pow(Py,2)+pow(Pz,2)));
TransverseP->Fill(sqrt(Px*Px + Py*Py));
particles[j].SetP(Px,Py,Pz);

//genero le particelle secondo definite proporzioni

double num_random = gRandom -> Uniform(0,1);
if (num_random < 0.4)  particles[j].SetIParticle(0);
else if (num_random < 0.8) particles[j].SetIParticle(1);
else if (num_random< 0.85) particles[j].SetIParticle(2);
else if (num_random<0.9) particles[j].SetIParticle(3);
else if (num_random<0.945) particles[j].SetIParticle(4);
else if (num_random< 0.99) particles[j].SetIParticle(5);
else if (num_random<0.995) {

particles[j].SetIParticle(6);
++k;
particles[k].SetIParticle(0);
particles[k+1].SetIParticle(3);
particles[j].decay2body(particles[k], particles[k+1]);
HDecay->Fill(particles[k].InvMass(particles[k+1]));
++k;
decaycounter += 2;
}

else {

particles[j].SetIParticle(6);
++k;
particles[k].SetIParticle(1);
particles[k+1].SetIParticle(2);
particles[j].decay2body(particles[k], particles[k+1]);
HDecay->Fill(particles[k].InvMass(particles[k+1]));
++k;
decaycounter += 2;
}
```

```
//Riempio gli istogrammi relativi all'energia e al tipo di particelle generate
TypeParticles -> Fill(particles[j].GetIParticle());
Energy -> Fill(particles[j].GetEnergy());
}

//Grazie a questo for riempio gli istogrammi delle masse invarianti
for (int m = 0; m < 100 + decaycounter; m ++) {
for (int n = 0; n < m; n ++) {
InvMass->Fill(particles[m].InvMass(particles[n]));

//controllo se le cariche sono discordi
if((particles[m].GetCharge()*particles[n].GetCharge())<0){
//riempio istogramma massa invariante carica discorde
OInvMass->Fill(particles[m].InvMass(particles[n]));
}

//controllo se le cariche sono concordi
if((particles[m]).GetCharge()*particles[n].GetCharge()>0){
//riempio istogramma massa invariante carica concorde
SInvMass->Fill(particles[m].InvMass(particles[n]));
}
//riempio l'istogramma della massa invariante pione- kaone discordi
if(((particles[m].GetIParticle()==0) && (particles[n].GetIParticle()==3)) ||
((particles[m].GetIParticle()==1)&& (particles[n].GetIParticle()==2)) ||
((particles[m].GetIParticle()==3)&& (particles[n].GetIParticle()==0)) ||
((particles[m].GetIParticle()==2) && (particles[n].GetIParticle()==1))){
h10->Fill(particles[m].InvMass(particles[n]));
}
//riempio l'istogramma della massa invariante pione - kaone concordi
if(((particles[m].GetIParticle()==0) && (particles[n].GetIParticle()==2)) ||
((particles[m].GetIParticle()==1) && (particles[n].GetIParticle()==3))||
((particles[m].GetIParticle()==2) && (particles[n].GetIParticle()==0)) ||
((particles[m].GetIParticle()==3) && (particles[n].GetIParticle()==1))){
h11->Fill(particles[m].InvMass(particles[n]));
}

}

}

}




//Creo la canvas e inizio l'analisi degli istogrammi
```

```
TCanvas *c = new TCanvas("c","Particle,Boost, Theta and Phi Distribution",
200,10,600,400);
c->Divide(2,2);

gStyle->SetOptFit(1111); //necessario per visualizzare info sul fit
gStyle->SetOptStat(2210);

//operazioni su TypeParticles
c->cd(1);
int confronto[7]={4000000,4000000,500000,500000,450000,450000,100000};
double chi_square=0, chi_reduced=0;
for(int i=1; i<8; i++){
double binContent= TypeParticles->GetBinContent(i);
double error= TypeParticles->GetBinError(i);
cout<<i<<'\n'<< "Content:\t"<<binContent<<'\n'<<
"Correct number:\t"<<confronto[i-1]<<'\n'<<'\n'<<"Difference:\t"<<
binContent-confronto[i-1]<<'\n'<<"Error:\t"<<error<<'\n';
chi_square+=pow(binContent-confronto[i-1],2)/confronto[i-1];
}
TypeParticles->Draw();
//c->Update();


//Operazioni su PhiDistribution
c->cd(2);
PhiDistribution->Draw();
//fit
TF1 *f1 = new TF1("f1","[0]",0,2*M_PI);
f1->SetParameter(0,1000);
f1->SetLineColor(kRed);
PhiDistribution-> Fit("f1");
PhiDistribution-> Draw("SAME");
cout << '\n';
cout << "Risultati del fit di Phi"<< endl;
cout << "Chi quadro" << f1->GetChisquare() << endl;
cout << "DOF:" << f1-> GetNDF() << endl;
cout << "Chi quadro ridotto" << f1 -> GetChisquare()/ f1->GetNDF()<< endl;
cout << "Par0" << f1 -> GetParameter(0) << "+/-" << f1 -> GetParError(0) << endl;


c->cd(3);
ThetaDistribution->Draw();
//fit
TF1 *f2 = new TF1("f2", "[0]", 0, M_PI);
f2 -> SetParameter(0,1000);
f2 -> SetLineColor(kRed);
ThetaDistribution -> Fit("f2");
ThetaDistribution -> Draw("SAME");
cout << '\n';
cout << "Risultati del fit di Theta"<< endl;
```

```
cout << "Chi quadro" << f2->GetChisquare() << endl;
cout << "DOF:" << f2-> GetNDF() << endl;
cout << "Chi quadro ridotto" << f2 -> GetChisquare()/ f2->GetNDF()<< endl;
cout << "Par0" << f2 -> GetParameter(0) << "+/-" << f2 -> GetParError(0) << endl;

c->cd(4);
PDistribution-> Draw();
//fit
TF1 *f3 = new TF1("f3", "[0]*exp(-x/[1])", 0, 10);
f3->SetParameter(0,10000);
f3->SetParameter(1,1);
f3->SetLineColor(kRed);
PDistribution->Fit("f3");
PDistribution->Draw("SAME");
cout << '\n';
cout << "Risultati del fit di P"<< endl;
cout << "Chi quadro" << f3->GetChisquare() << endl;
cout << "DOF:" << f3-> GetNDF() << endl;
cout << "Chi quadro ridotto" << f3 -> GetChisquare()/ f3->GetNDF()<< endl;
cout << "Par1" << f3 -> GetParameter(1) << "+/-" << f3 -> GetParError(1) << endl;

c-> Print("output/c.pdf");

TCanvas *c3 = new TCanvas ("K*, Subtract, Subtract2", "K*,
Subtract, Subtract2", 200, 10000, 600, 400);

c3 -> Divide(3,1);
c3 -> cd(1);

//operazioni sul grafico della massa invariante delle k*
HDecay -> Draw();
//fit
HDecay->Fit("gaus");
TF1 * f4 = HDecay->GetFunction("gaus");
cout << '\n';
cout << "Risultati del fit della massa invariante delle k*"<< endl;
cout<< "K* media: "<< HDecay -> GetMean() <<endl;
cout << "Sigma:" << HDecay -> GetRMS() << endl;
cout<<"chi quadro ridotto"<< f4-> GetChisquare()/ f4->GetNDF() <<endl;


c3 -> cd(2);

//Sottrazione Istogrammi
TH1F * Subtract= new TH1F (*h10);
Subtract -> Sumw2();
Subtract -> Add(h10, h11, 1.0, -1.0);
Subtract->GetXaxis()-> SetTitle("Mass [GeV/c^2]");
Subtract->GetYaxis()-> SetTitle("Entries");
Subtract->Draw();
```

```
Subtract -> SetTitle("Subtract: h10-h11");
Subtract->Fit("gaus");


c3 -> cd(3);

//secondo istogramma di sottrazione
TH1F * Subtract2 = new TH1F (*OInvMass);
Subtract2->GetXaxis()-> SetTitle("Mass [GeV/c^2]");
Subtract2->GetYaxis()-> SetTitle("Occorrenze");
Subtract2 -> Sumw2();
Subtract2 -> Add(OInvMass, SInvMass, 1.0, -1.0);
Subtract2->Draw();
Subtract2 -> SetTitle("Subtract2: OInvMass - SInvMass");
Subtract2->Fit("gaus");

//Legenda
c3->cd(1);
TLegend *legDecayedK = new TLegend{.1,.7,.3,.9,"Decayed K* Legend"};
legDecayedK->SetFillColor(0);
legDecayedK->AddEntry(InvMass,"Inv mass generation");
legDecayedK->AddEntry("gaus","Fit gaussiano");


c3->cd(2);
TLegend *legPKDifference = new TLegend{.1,.7,.3,.9,"Pione-Kaone Difference Legend"};
legPKDifference->SetFillColor(0);
legPKDifference->AddEntry(Subtract,"Inv Mass generation");
legPKDifference->AddEntry("gaus","Fit gaussiano");


c3->cd(3);
TLegend *legChargeDiff = new TLegend{.1,.7,.3,.9,"Charge Difference Legend"};
legChargeDiff->SetFillColor(0);
legChargeDiff->AddEntry(Subtract2,"Inv Mass generation");
legChargeDiff->AddEntry("gaus","Fit gaussiano");


c3 -> Print("output/InvMass.pdf");

file->Write();


}
```