Relazione Finale Modulo Root
Anno accademico 2020-2021

Marco Caporale
Matricola 0000915327

Dicembre 2020

1 Introduzione

La presente relazione ha lo scopo di illustrare il funzionamento del programma del modulo di Root per l'esame di Laboratorio di Elettromagnetismo e Ottica.

Il programma simula la generazione di 10^5 eventi composti da 100 particelle ciascuno. Ogni particella è generata da distribuzioni di probabilità, sia per il tipo di particella che per i parametri della stessa.
I dati vengono quindi utilizzati per riempire degli istogrammi dai quali verificheremo che la generazione è avvenuta correttamente secondo i parametri forniti fittando le distribuzioni.
Utilizzeremo infine i metodi di fit di Root per l'analisi degli istogrammi per estrarre i parametri delle risonanze Kstar.

2 Funzionamento codice

Il codice è diviso logicamente in due parti, gli header utilizzati (e le rispettive implementazioni) ed il main dove viene eseguita la simulazione, vengono costruiti gli istogrammi e vengono fatte le opportune operazioni di fit.

2.1 Header

Per il programma sono stati utilizzati 3 file di intestazione. Il primo di essi è **particletype.h**. In particletype è definita la omonima classe che descrive un oggetto composto da un nome (di tipo std::string), una massa (double) ed un carica (int). Gli attributi privati sono accompagnati dai relativi getter, dal costruttore. Infine per completare la classe sono presenti due funzioni virtuali, una che restituisce il valore della risonanza (spiegata in resonancetype.h) e da una funzione di print che stampa a schermo i valori degli attributi.

Nell'header **resonancetype.h** costruiamo una classe che eredita pubblicamente gli attributi della particletype e inoltre aggiunge un ulteriore attributo ovvero la larghezza della risonanza (tipo double). Onde evitare undefined behaviour durante l'esecuzione del programma l'aver definito il getter della risonanza virtuale permette di chiamarlo anche su oggetti privi di risonanza restituendo il

valore 0. Ovviamente è eseguito l'overload sulla funzione print per aggiungere a schermo anche il valore della risonanza.

L'ultimo header è **particle.h** che contiene la classe particle. Nella suddetta espandiamo il concetto di particella basandoci sulle classi precedentemente descritte. La struttura di base è composta da un array statico di particelle che contiene i tipi di particletype che verranno aggiunti al programma. Questo array ha capienza limitata perchè il numero di particelle usate nella simulazione non sarà elevato. Ad ogni particella che verrà creata verrà associato un indice che la associa alla corrispettiva particletype dell'array statico. Quando una particella viene aggiunta all'array tramite apposita funzione un messaggio di stampa a schermo lo comunica all'utente indicandogli inoltre quanti posti sono ancora disponibili. Vi è una funzione di controllo che assicura che non venga aggiunta più volte la stessa particella. Inoltre le particelle sono dotate di attributi double per la quantità di moto lungo gli assi principali. Sono implementati poi i rispettivi getter e setter. Inoltre sono presenti le funzioni per ottenere energia, massa e massa invariante di una particella. Infine è presente una funzione per il decadimento della particella.

2.2 Main

Nel Main si svolge il resto del programma. In un primo momento vengono aggiunti all'array statico i tipi di particelle che andremo ad utilizzare durante il programma.
Sono quindi creati gli istogrammi che andremo a riempire durante il corso del programma, alcuni sono stati creati in copia multipla con differente binnaggio per meglio apprezzarne il riempimento.
A questo punto il programma entra nel loop principale che di fatto è la sezione più consistente in quanto a carico operativo per la macchina. Qui vengono eseguiti i 10^5 cicli di creazione delle 100 particelle. Le generazioni avvengono tramite generazione casuale di 4 valori. I primi due sono gli angoli di uscita θ e ϕ, rispettivamente generati uniformemente su π e su 2π. Il primo angolo indica la direzione rispetto all'asse z, il secondo invece l'angolo spazzato sul piano xy. Viene poi generata la quantità di moto da distribuzione esponenziale, associando poi i rispettivi valori sugli assi moltiplicando il modulo per le opportune funzioni trigonometriche degli angolo prima generati. Viene infine generato un valore da distribuzione uniforme fra 0 ed 1. A seconda del valore viene associato il tipo di particletype ai dati precedentemente generati, di fatto ottenendo le particelle desiderate secondo le distribuzioni percentuali volute.
Terminata la generazione delle 100 particelle si procede alle operazioni di analisi per il riempimento degli istogrammi.
Il ripetersi del ciclo per 10^5 volte può risultare tedioso e lungo per l'utente specie se la macchina su cui è eseguito il programma non è eccessivamente performante. Per questo motivo è stata aggiunta una stampa a terminale di una barra di riempimento che indicativamente mostra lo stato di avanzamento del programma (figura 1).

Al termine del ciclo sono creati gli istogrammi e le canvas di Root. Sono eseguite le operazioni di sottrazione fra istogrammi ed eseguiti i fit lineari per le distribuzioni angolari onde verificare che la generazione sia effettivamente uniforme come atteso, esponenziale sulla quantità di moto ed infine i fit guassiani

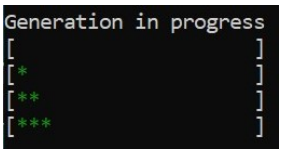

Figura 1: Barra di riempimento

per ottenere i dati dei decadimenti delle risonanze Kstar.
Una stampa a schermo mostra i parametri dei fit e le distribuzioni delle particelle generate (di fianco al loro valore atteso).
Gli oggetti di Root sono quindi salvati su un Root file.
Infine il terminale emette un suono di notifica per avvisare l'utente della fine del programma (sempre per le considerazioni prima fatte sul tempo impiegato dall'esecuzione) e viene stampato a schermo il tempo totale di esecuzione.

2.3 Macro.c

Per la lettura del file .root e la ricreazione delle Canvas (che non vengono salvate sul file Root) è stata aggiunta la macro `macro.c`.

2.4 Compile Options

L'ambiente Root su cui è stato sviluppato il programma supporta esclusivamente la compilazione esterna da Bash, sono pertanto riportate le opzioni di compilazione utilizzate dal terminale

```
g++ particle.cpp particletype.cpp resonancetype.cpp main.cpp `root-config
-cflags -libs` -O3 -Wall -Wextra
```

3 Generazione

Il programma genera 7 tipi differenti di particletype con rispettiva probabilità

- Pioni+, 40%
- Pioni-, 40%
- Kaoni+, 5%
- Kaoni-, 5%
- Protoni+, 4.5%
- Protoni-, 4.5%
- Risonanze K*, 1%

In ogni evento vengono generate 100 particelle.
In ogni esecuzione vengono generati 10^5 eventi. Le proprietà cinematiche sono generate tramite la generazione casuale secondo distribuzione esponenziale del modulo dell'impulso e distribuzioni uniformi per gli angoli θ e ϕ. Con opportuno

cambio di coordinate possiamo assegnare quindi i moduli degli impulsi nei 3 assi cartesiani.

4 Analisi Dati

Dai dati generati abbiamo differenti aspetti da analizzare:

4.1 Percentuali di popolazione

Il primo dato da verificare è che effettivamente le particelle siano state generate secondo le distribuzioni attese, possiamo verificare che effettivamente ciò avviene correttamente sia dalla stampa a schermo che dall'istogramma di Root.

```
Generated particles:
Pion+   are 0.399843 ± 0.000199961,          0.40 expected
Pion-   are 0.400316 ± 0.000200079,          0.40 expected
Kaon+   are 0.0499808 ± 7.06971e-05,          0.05 expected
Kaon-   are 0.0499832 ± 7.06988e-05,          0.05 expected
Proton+ are 0.044976 ± 6.70641e-05,          0.045 expected
Proton- are 0.0448812 ± 6.69934e-05,          0.045 expected
K*      are 0.0100196 ± 3.16538e-05,          0.01 expected
```

Figura 2: Stampa a schermo delle popolazioni relative e rispettivo dato atteso

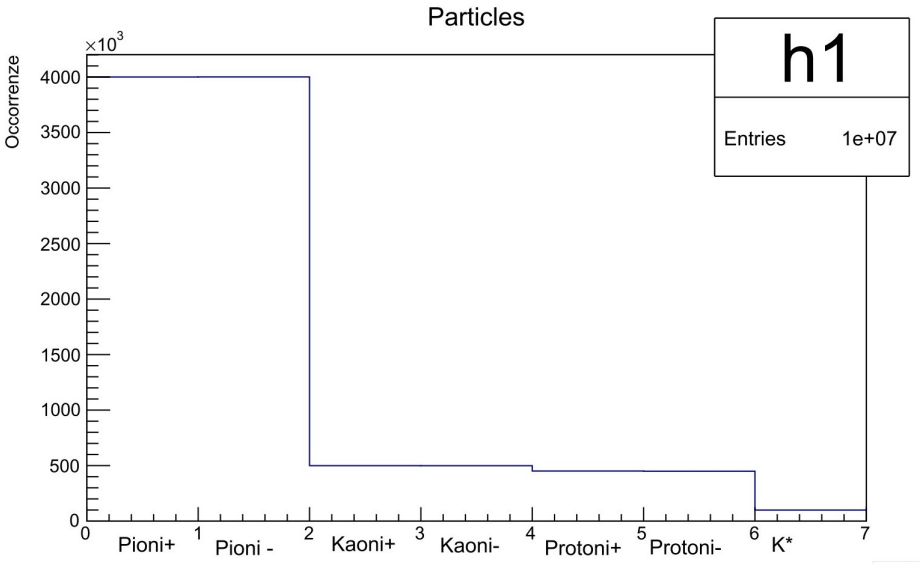

Figura 3: Grafico Root delle particelle generate

4.2 Angoli di emissione

A questo punto passiamo ad analizzare gli angoli di emissione delle particelle.

```
AZIMUTHAL AND POLAR ANGLES FITS

****************************************
Minimizer is Linear / Migrad
Chi2                      =      491.156
NDf                       =          499
p0                        =        19999   +/-   6.3244

****************************************
Minimizer is Linear / Migrad
Chi2                      =      233.248
NDf                       =          249
p0                        =      39999.1   +/-   12.649
```

Figura 4: Stampa a schermo del fit dell'angolo

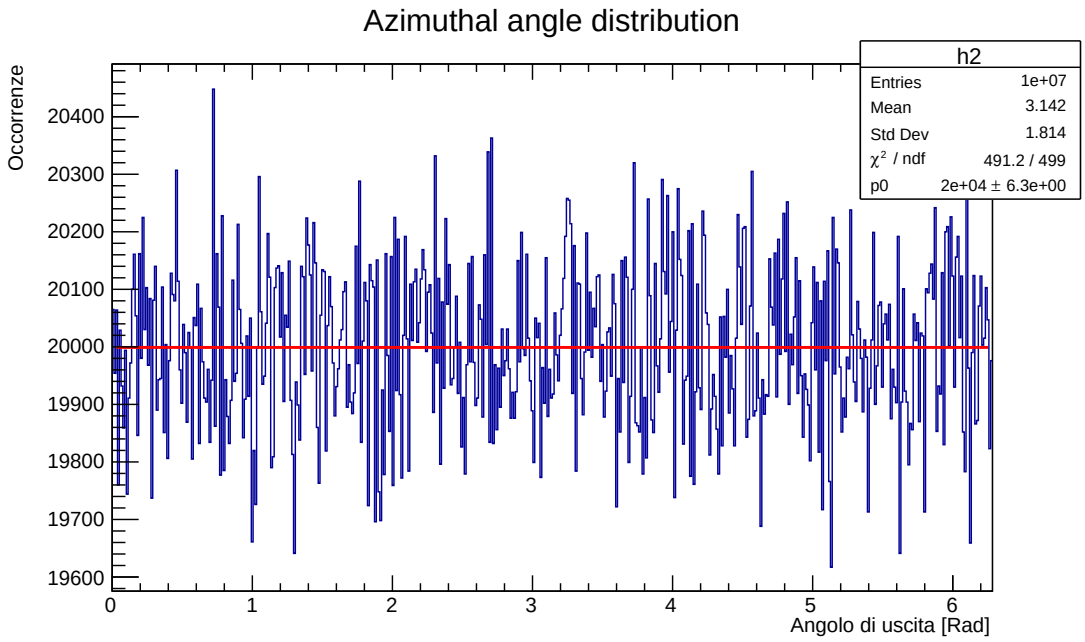

Figura 5: Grafico Root angoli azimutali

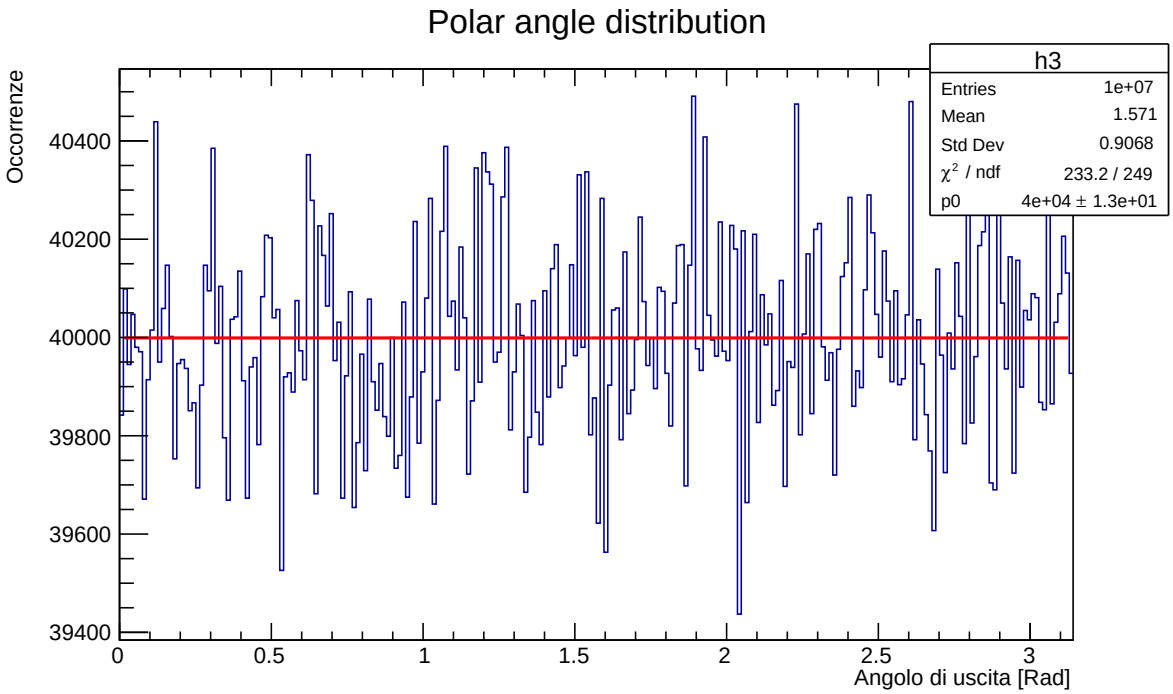

Figura 6: Grafico Root angoli polari

4.3 Modulo dell'impulso

Analizziamo ora il modulo dell'impulso

```
IMPULSE EXPONENTIAL FIT
 FCN=726.895 FROM MIGRAD    STATUS=CONVERGED      46 CALLS          47 TOTAL
                     EDM=2.48883e-10    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                   STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  Constant     1.11389e+01   4.58262e-04   5.31146e-06  -4.86900e-02
   2  Slope       -1.00080e+00   3.38916e-04   3.08745e-06  -9.51074e-02
```

Figura 7: Stampa a schermo del fit esponenziale

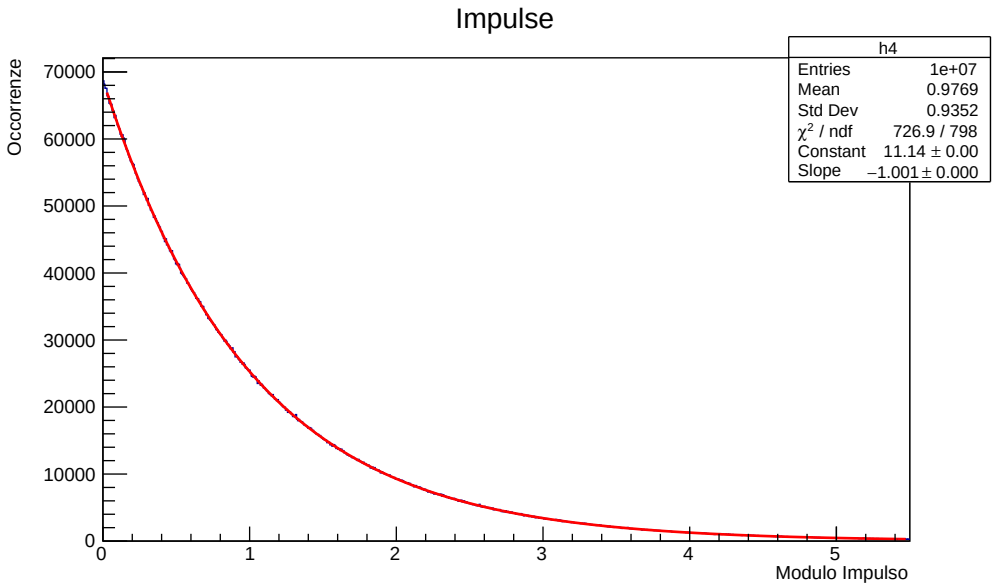

Figura 8: Grafico modulo impulso particelle fittato

4.4 Risonanze K*

Per ultimo procediamo all'analisi delle risonanze Kstar estratte dalle differenze degli istogrammi. Assumere GeV/c^2 come unità di misura della massa invariante.

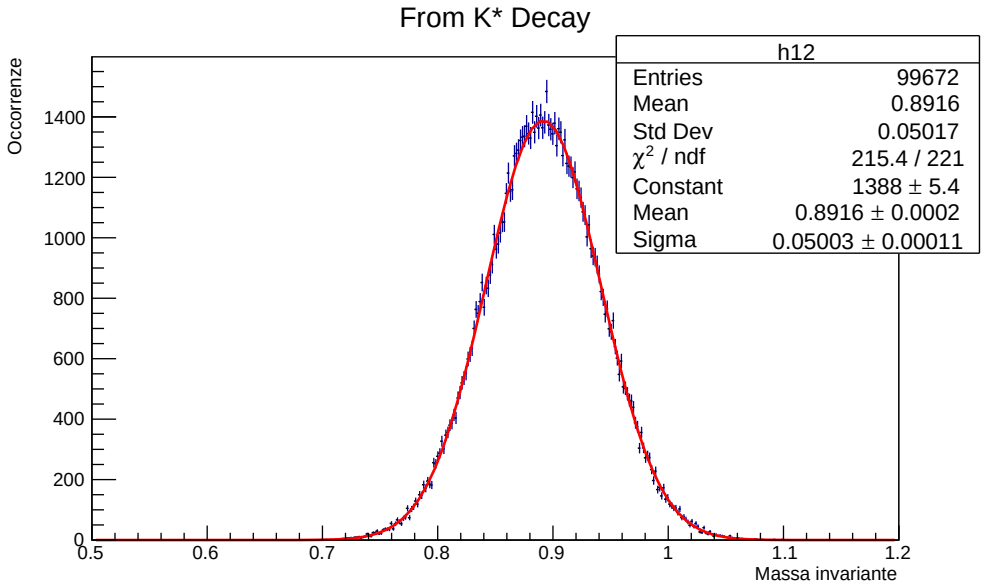

Figura 9: Decadimento della K*

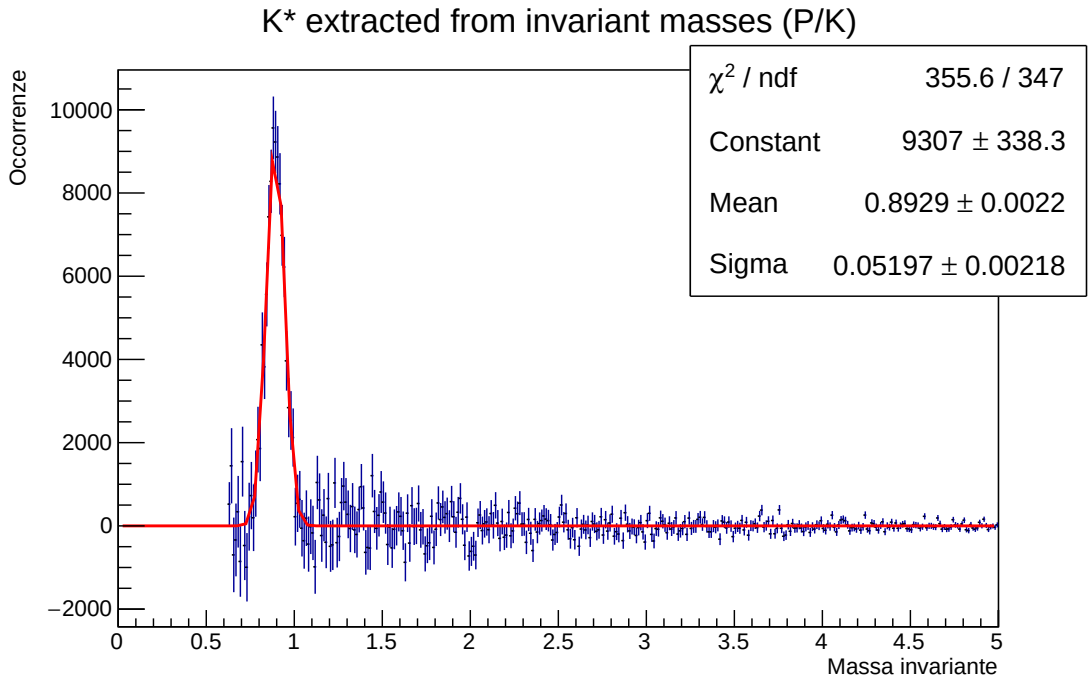

Figura 10: Massa invariante Pione/Kaone

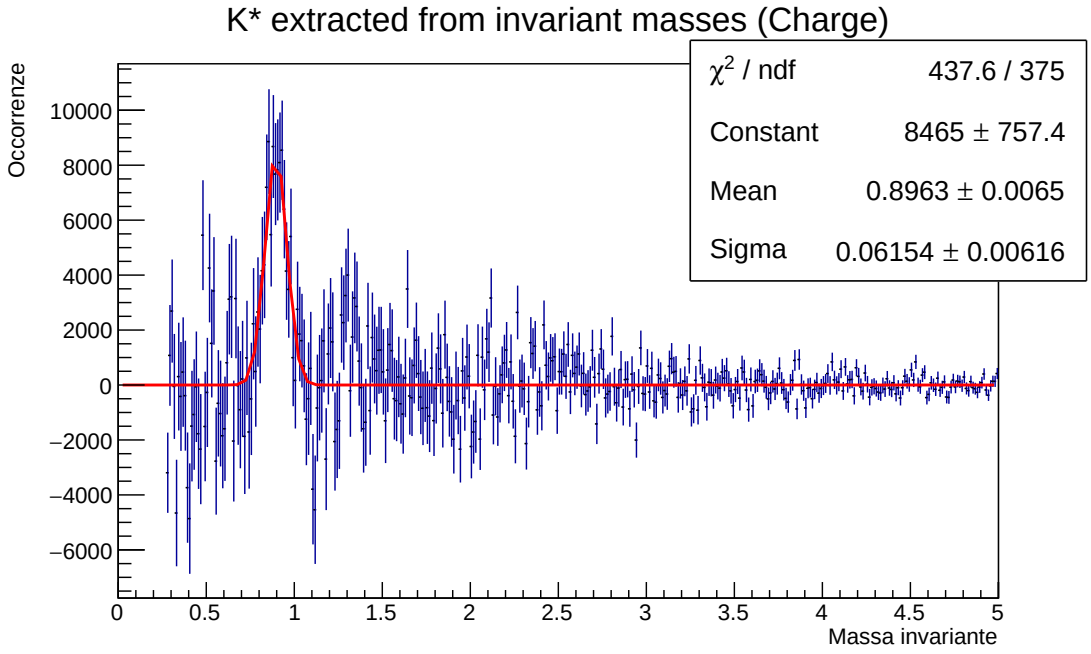

Figura 11: Massa invariante per carica

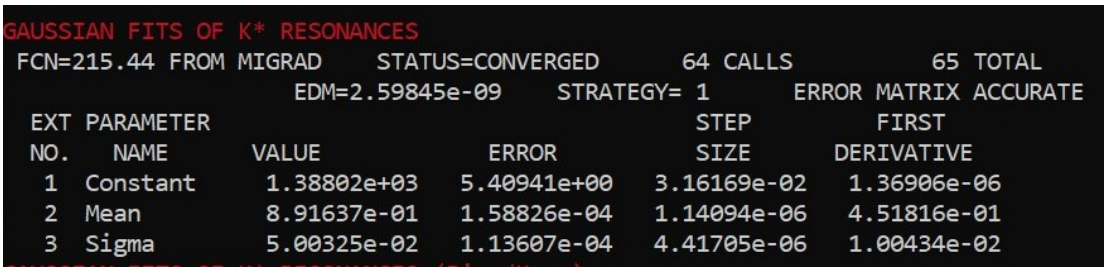

Figura 12: Parametri del fit di K* stampato a terminale

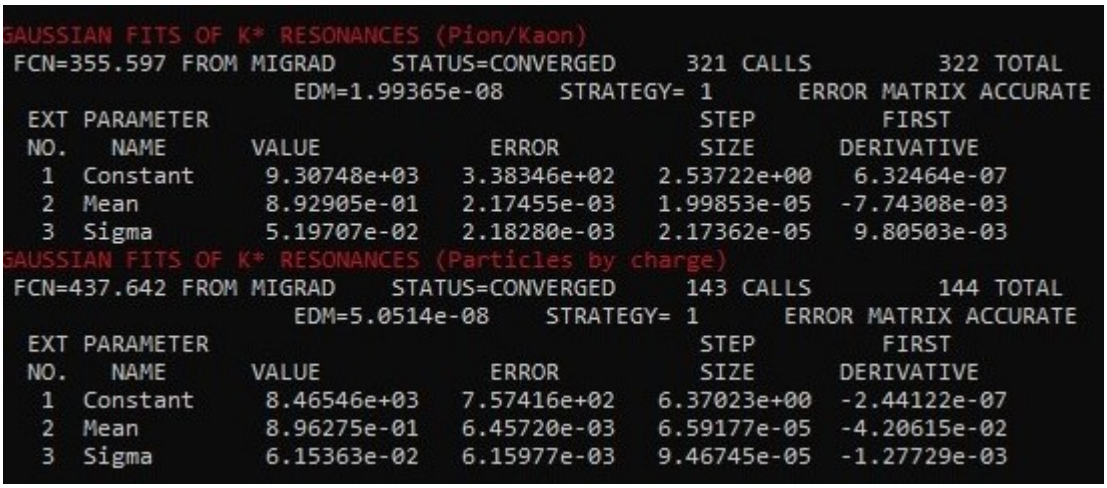

Figura 13: Parametri dei fit stampati a terminale

4.5 Tabelle dei dati

Specie	Occorrenze osservate	Occorrenze attese
$\pi+$	$(39.984 \pm 0.020)\%$	40%
$\pi-$	$(40.032 \pm 0.020)\%$	40%
K+	$(4.9981 \pm 0.0071)\%$	5%
K-	$(4.9983 \pm 0.0071)\%$	5%
p+	$(4.4498 \pm 0.0067)\%$	4.5%
p-	$(4.4881 \pm 0.0067)\%$	4.5%
K*	$(1.0017 \pm 0.0032)\%$	1%

Tabella 1: Occorrenze delle particelle

Distribuzioni	Parametri del fit	χ^2	DOF	χ^2/DOF
Fit a distribuzione angolo ϕ (pol0)	19999 ± 6.3	491.2	499	0.98
Fit a distribuzione angolo θ (pol0)	39999 ± 13	233.248	249	0.94
Fit a distribuzione modulo impulso (expo)	-1.00080 ± 0.00034	726.9	798	0.91

Tabella 2: Fit degli angoli

Distribuzione	Media	σ	Ampiezza	χ^2/DOF
Massa Invariante vere K* (gauss)	0.8916 ± 0.0002	0.05003 ± 0.00011	1388 ± 5.4	0.97
Massa Invariante ottenuta da differenza delle combinazioni πK di carica discorde e concorde (gauss)	0.8929 ± 0.0022	0.0520 ± 0.0022	9310 ± 340	1.02
Massa Invariante ottenuta da differenza delle combinazioni di carica discorde e concorde (gauss)	0.8963 ± 0.0065	0.0615 ± 0.0062	8470 ± 760	1.17

Tabella 3: Analisi delle K*

4.6 Canvas ricreate da macro.c

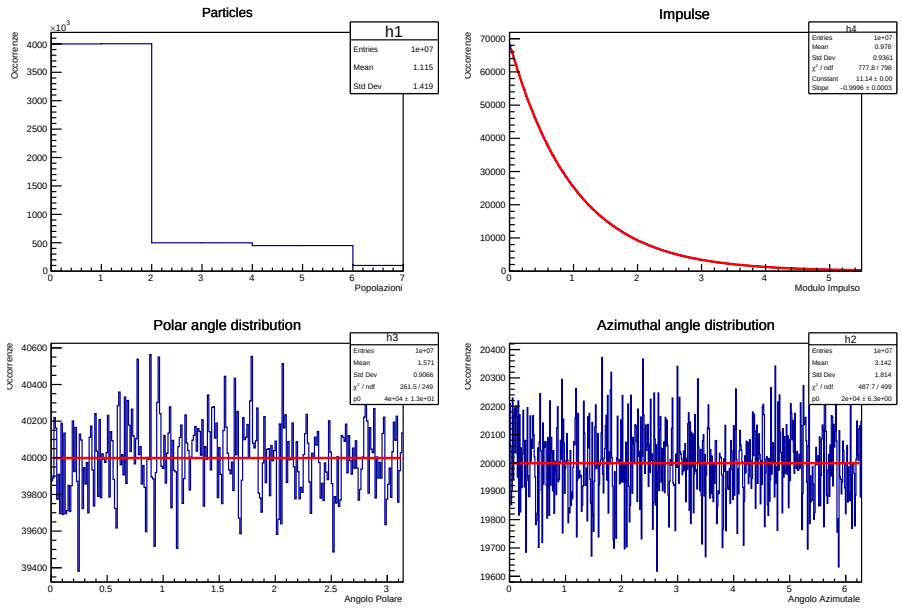

Figura 14: Ricreazione di una Canvas 1 da un'altra generazione

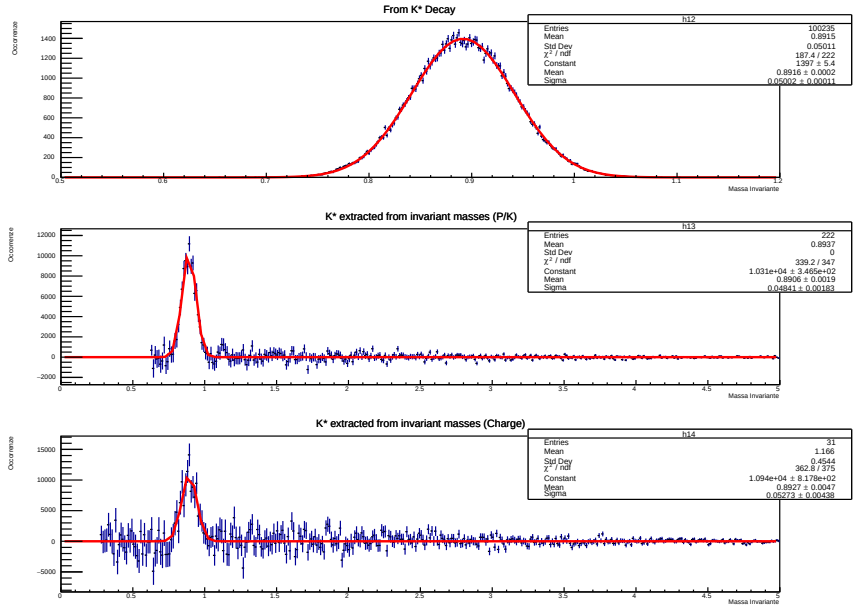

Figura 15: Ricreazione di una Canvas 2 da un'altra generazione

5 Codice

Tutto il codice è anche disponibile presso la repository
https://github.com/gorsedh/laboratorio2/tree/main/RootFinale

Il codice è seguentemente così ordinato:

- `particletype.h`
- `particletype.cpp`
- `resonancetype.h`
- `resonancetype.cpp`
- `particle.h`
- `particle.cpp`
- `main.cpp`
- `macro.c`

5.1 particletype.h

```
#ifndef PARTICLETYPE_H
#define PARTICLETYPE_H
```

```
#include <iostream>

class ParticleType
{
public:
    std::string GetParticleName() const; //i getter
    double GetMass() const;
    int GetCharge() const;
    virtual double GetWidth() const;

    virtual void Print() const; //print

    ParticleType(std::string name, double mass, int
        charge); //costruttore

private: //attributi
    std::string const fName_;
    double const fMass_;
    int const fCharge_;
};

#endif
```

5.2 particletype.cpp

```
//implementazioni particletype

#include "particletype.h"

#include <iostream>

std::string ParticleType::GetParticleName() 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 << "Particle name is " << fName_ << '\n';
    std::cout << "Particle mass is " << fMass_ << '\n';
    std::cout << "Particle charge is " << fCharge_ << '\n
        ';
}

ParticleType::ParticleType(std::string name, double mass,
    int charge) : fName_{name}, fMass_{mass}, fCharge_{
    charge} {};
```

5.3 resonancetype.h

```
#ifndef RESONANCETYPE_H
#define RESONANCETYPE_H

#include <iostream>

class ResonanceType : public ParticleType
{
public:
    double GetWidth() const override; //getter

    void Print() const override; //print

    ResonanceType(std::string name, double mass, int
        charge, double width); //costruttore

private: //attributo di resonance
    double const fWidth_;
};

#endif
```

5.4 resonancetype.cpp

```
//implementazioni resonancetype

#include "particletype.h"
#include "resonancetype.h"

#include <iostream>

void ResonanceType::Print() const
{
    ParticleType::Print();
    std::cout << "Resonance width is " << fWidth_ << '\n'
        ;
}

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

ResonanceType::ResonanceType(std::string name, double
    mass, int charge, double width) : ParticleType{name,
    mass, charge}, fWidth_{width} {};
```

5.5 particle.h

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include "particletype.h"
```

```
#include "resonancetype.h"

#include <iostream>
#include <cmath>

class Particle
{
public:
    Particle(std::string name, double fPx, double fPy_,
        double fPz_);
    Particle() = default;

    //static void AddParticleType(std::string name,
        double mass, int charge);
    static void AddParticleType(std::string name, double
        mass, int charge, double width = 0);

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

    int GetIndex() const;
    double GetXMomentum() const;
    double GetYMomentum() const;
    double GetZMomentum() const;
    double GetMomentumModule() const;

    double GetMass() const;
    double GetInvMass(Particle &p);

    double ParticleEnergy() const;

    void ParticlePrint() const;

    void SetParticle(int index);
    void SetParticle(std::string name);
    void SetParticleMomentum(double xMomentum, double
        yMomentum, double zMomentum);

    void static PrintArray();

private:
    static int FindParticle(std::string name);

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

    static const int fMaxNumParticleType_ = 10;
                    //numero di particelle disponibili
    static ParticleType *fParticleType_[
        fMaxNumParticleType_]; //puntatore al tipo di
        particelle
    static int fNParticleType_;
```

```
                                        //contatore di
        particelle non vuote

    int fIParticle_ = 0; //indice della particella nell'
        array
    double fPx_ = 0;      //componenti dell'impulso
    double fPy_ = 0;
    double fPz_ = 0;
};

#endif
```

5.6 particle.cpp

```
#include "particle.h"
#include "particletype.h"
#include "resonancetype.h"

#include <iostream>
#include <cmath>
#include <cstdlib>

int Particle::fNParticleType_ = 0;
ParticleType *Particle::fParticleType_[
    fMaxNumParticleType_];

Particle::Particle(std::string name, double fPx, double
    fPy, double fPz) : fPx_{fPx}, fPy_{fPy}, fPz_{fPz}
{
    fIParticle_ = FindParticle(name);
};

int Particle::FindParticle(std::string name)
{
    for (int i = 0; i < fNParticleType_; i++)
    {
        if (fParticleType_[i]->GetParticleName() == name)
        {
            return i;
        }
    }
    std::cout << name << "␣not␣found" << '\n';
    return fNParticleType_ + 1;
};

void Particle::Boost(double bx, double by, double bz)
{

    double energy = ParticleEnergy();
```

```
    //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;
}

void Particle::AddParticleType(std::string name, double
    mass, int charge, double width)
{

    int index = FindParticle(name);

    if (width == 0)
    {
        if (index < fNParticleType_)
        {
            std::cout << "Particle " << name << " already
                in the array\n";
            return;
        }

        if (fNParticleType_ < fMaxNumParticleType_)
        {
            fParticleType_[fNParticleType_] = new
                ParticleType(name, mass, charge);
            std::cout << "Particle " << name << " added
                to the array\n";
            fNParticleType_++;
            std::cout << "There are " <<
                fMaxNumParticleType_ - fNParticleType_ <<
                " spaces left in the array\n\n";
        }
        else
        {
            std::cout << "ARRAY IS FULL, DELETE SOME
                PARTICLES\n\n\n";
        }
    }
    else
    {

        if (index < fNParticleType_)
        {
            std::cout << "Resonance " << name << "
                already in the array\n";
```

```
            return;
        }

        if (fNParticleType_ < fMaxNumParticleType_)
        {
            fParticleType_[fNParticleType_] = new
                ResonanceType(name, mass, charge, width);
            std::cout << "Resonance " << name << " added
                to the array\n";
            fNParticleType_++;
            std::cout << "There are " <<
                fMaxNumParticleType_ - fNParticleType_ <<
                " spaces left in the array\n\n";
        }
        else
        {
            std::cout << "ARRAY IS FULL, DELETE SOME
                PARTICLES\n\n\n";
        }
    }
};

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 (fIParticle_ > -1)
    { // add width effect

        // gaussian random numbers

        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)
    {
        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.SetParticleMomentum(pout * sin(theta) * cos(phi)
        , pout * sin(theta) * sin(phi), pout * cos(theta))
        ;
    dau2.SetParticleMomentum(-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;
}

int Particle::GetIndex() const { return fIParticle_; };
double Particle::GetXMomentum() const { return fPx_; };
double Particle::GetYMomentum() const { return fPy_; };
double Particle::GetZMomentum() const { return fPz_; };
double Particle::GetMomentumModule() const { return sqrt(
```

```
    fPx_ * fPx_ + fPy_ * fPy_ + fPz_ * fPz_); };

double Particle::GetMass() const
{
    int i = this->fIParticle_;
    return fParticleType_[i]->GetMass();
};

double Particle::GetInvMass(Particle &p)
{
    double totE = pow(ParticleEnergy() + p.ParticleEnergy
        (), 2);
    double totP = pow(GetXMomentum() + p.GetXMomentum(),
        2) + pow(GetYMomentum() + p.GetYMomentum(), 2) +
        pow(GetZMomentum() + p.GetZMomentum(), 2);
    double invMass = sqrt(totE - totP);
    return invMass;
};

//int Particle::GetiParticle() const {return fIParticle_
    ;};

double Particle::ParticleEnergy() const
{ //this function returns the total energy of the
    particle (c is assumed as 1)
    double energy = 0;
    energy = sqrt(pow(this->GetMass(), 2) + pow(this->
        GetMomentumModule(), 2));
    return energy;
};

void Particle::ParticlePrint() const
{
    int ParticleIndex = GetIndex();
    std::cout << "Index is " << ParticleIndex << "\n";
    std::cout << "Particle type is " << fParticleType_[
        ParticleIndex]->GetParticleName() << "\n";
    std::cout << "Momentum of the particle is:\n x: " <<
        GetXMomentum() << "\n y: " << GetYMomentum()
            << "\n z: " << GetZMomentum() << "\n\n";
};

void Particle::SetParticle(int index)
{
    if (index < fNParticleType_)
    {
        fIParticle_ = index;
    }
    else
    {
```

```
        std::cout << "No particle is associated to the
            index given in SetParticle()\n";
    }
};

void Particle::SetParticle(std::string name)
{
    int index = FindParticle(name);
    if (index < fNParticleType_)
    {
        fIParticle_ = index;
    }
    else
    {
        std::cout << "No particle is associated to the
            name given in SetParticle\n";
    }
};

void Particle::SetParticleMomentum(double xMomentum,
    double yMomentum, double zMomentum)
{
    fPx_ = xMomentum;
    fPy_ = yMomentum;
    fPz_ = zMomentum;
};

void Particle::PrintArray()
{
    std::cout << "\n\033[0;36mArray of particles contains
        " << fNParticleType_ << " particles:\033[0m\n\n
        \033[1;33m";
    for (int i = 0; i < fNParticleType_; i++)
    {
        fParticleType_[i]->Print();
        std::cout << '\n';
    }
    std::cout << "\033[0m\n";
}
```

5.7 main.cpp

```
#include <string>
#include <cmath>
#include <iostream>

#include "TH1F.h"
#include "TRandom.h"
#include "TMath.h"
#include "TCanvas.h"
```

```
#include "TROOT.h"
#include "TF1.h"
#include "TFile.h"
#include "TFitResult.h"
#include "TStyle.h"

#include "particle.h"

int main()
{
    auto startTime = std::chrono::system_clock::now();
    std::cout << "\nSimulation Started\n";

    //compilazione dei file di implementazioni necessari,
        collegati alle librerie in particle.h
    gROOT->LoadMacro("particletype.cpp+");
    gROOT->LoadMacro("resonancetype.cpp+");
    gROOT->LoadMacro("particle.cpp+");

    //dati del programma
    int const numberOfParticles = 100; //particelle per
        array
    int const N = 120;                 //massima size
        dell'array
    int const events = 100000;         //generazioni
    double phi, theta, impulse;

    //grafica
    int const binsForProgressBar = 20;
    int const PercentEvents = events / binsForProgressBar
        ;
    int barFilling = 0;

    //dati delle particelle in uso e altre variabili
        necessarie
    double const massP = 0.13957;
    double const massK = 0.49367;
    double const massProton = 0.93827;
    double const massKres = 0.89166;
    double pi = M_PI;

    //caricamento delle particelle necessarie
    Particle::AddParticleType("Pion+", massP, 1);
                //0 //80% spawn
    Particle::AddParticleType("Pion-", massP, -1);
               //1
    Particle::AddParticleType("Kaon+", massK, 1);
                //2 //10% spawn
    Particle::AddParticleType("Kaon-", massK, -1);
               //3
```

```
Particle::AddParticleType("Proton+", massProton, 1);
      //4 //9% spawn
Particle::AddParticleType("Proton-", massProton, -1);
      //5
Particle::AddParticleType("Kstar", massKres, 0,
    0.050); //6 //1% spawn

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

//creazione delle canvas
auto c1 = new TCanvas();
auto c2 = new TCanvas();
c1->Divide(3, 2);
c2->Divide(3, 2);

//i 12 grafici
auto h1 = new TH1F("h1", "Particles", 7, 0, 7);
                          //tipi di particelle
    generati
auto h2 = new TH1F("h2", "Azimuthal angle
    distribution", 500, 0, 2 * pi); //distribuzioni
    degli angoli azimutali (phi)
auto h3 = new TH1F("h3", "Polar angle distribution",
    250, 0, pi);          // " " " polari (theta)
auto h4 = new TH1F("h4", "Impulse", 800, 0, 5.5);
                            //impulso
auto h5 = new TH1F("h5", "Transverse impulse", 600,
    0, 4);                 //impulso trasverso (no z
    component)
auto h6 = new TH1F("h6", "Energy", 600, 0, 5);
                          //energia della
    particella

auto h7 = new TH1F("h7", "Invariant masses", 400, 0,
    5);              //massa invariante fra tutte le
    particelle generate in ogni evento
auto h8 = new TH1F("h8", "Invariant m. different
    charge", 400, 0, 5); //massa invariante fra tutte
    le particelle generate in ogni evento, in
    combinazioni con carica di segno discorde
auto h9 = new TH1F("h9", "Invariant m. same charge",
    400, 0, 5);       //massa invariante fra tutte le
    particelle generate in ogni evento, in
    combinazioni con carica di segno concorde
auto h10 = new TH1F("h10", "P+/K-, P-/K+", 400, 0, 5)
    ;                 //massa invariante fra tutte le
    particelle generate in ogni evento con
    combinazioni di tipo pione+/Kaone- e pione-/Kaone+
auto h11 = new TH1F("h11", "P+/K+, P-/K-", 400, 0, 5)
```

```
    ;                   //massa invariante fra tutte le
    particelle generate in ogni evento con
    combinazioni di tipo pione+/Kaone+ e pione-/Kaone-
auto h12 = new TH1F("h12", "From K* Decay", 400, 0.5,
     1.2);              //massa invariante fra le
    particelle generate in ogni evento che derivano
    dal decadimento della risonanza K*
//Per le operazioni sugli istogrammi in parte 3
auto h10b = new TH1F("h10b", "P+/K-, P-/K+", 5000,
    0.5, 1.5);  //massa invariante fra tutte le
    particelle generate in ogni evento con
    combinazioni di tipo pione+/Kaone- e pione-/Kaone+
auto h11b = new TH1F("h11b", "P+/K+, P-/K-", 5000,
    0.5, 1.5);  //massa invariante fra tutte le
    particelle generate in ogni evento con
    combinazioni di tipo pione+/Kaone+ e pione-/Kaone-
auto h12b = new TH1F("h12b", "From K* Decay", 5000,
    0.5, 1.5); //massa invariante fra le particelle
    generate in ogni evento che derivano dal
    decadimento della risonanza K*

h12->Sumw2();
//std::cout << "grafici fatti";

gRandom->SetSeed();

std::cout << "\nGeneration in progress\n";

for (int i = 0; i < events; i++)
{ //i 100k eventi

    Particle particles[N];
    int extraParticles = 0; //particelle da
        aggiungere in fondo all'array per i
        decadimenti k*
    if (i % PercentEvents == 0)
    {
        //std::cout << i << '|n';
        std::cout << "[";
        for (int bF = 0; bF < barFilling; bF++)
        {
            std::cout << "\033[0;32m*\033[0m";
        }
        for (int bE = 0; bE < binsForProgressBar -
            barFilling; bE++)
        {
            std::cout << " ";
        }
        std::cout << "]\n";
```

```
            barFilling++;
        }
        for (int j = 0; j < numberOfParticles; j++)
        { //i nostri array da 100

            phi = gRandom->Uniform(2 * pi);
            h2->Fill(phi);
            theta = gRandom->Uniform(pi);
            h3->Fill(theta);
            impulse = gRandom->Exp(1);
            h4->Fill(impulse);

            double PX, PY, PZ;
            PX = impulse * sin(theta) * cos(phi);
            PY = impulse * sin(theta) * sin(phi);
            PZ = impulse * cos(theta);
            particles[j].SetParticleMomentum(PX, PY, PZ);

            double transverseImpulse = sqrt(PX * PX + PY
                * PY);
            h5->Fill(transverseImpulse);

            double p = gRandom->Uniform(1);
            if (p < 0.40)
            { //Pion+
                particles[j].SetParticle("Pion+");
                h1->Fill(0);
            }
            else if (p < 0.80)
            { //Pion-
                particles[j].SetParticle("Pion-");
                h1->Fill(1);
            }
            else if (p < 0.85)
            { //Kaon+
                particles[j].SetParticle("Kaon+");
                h1->Fill(2);
            }
            else if (p < 0.90)
            { //Kaon-
                particles[j].SetParticle("Kaon-");
                h1->Fill(3);
            }
            else if (p < 0.945)
            { //Proton+
                particles[j].SetParticle("Proton+");
                h1->Fill(4);
            }
            else if (p < 0.99)
            { //Proton-
```

```
                particles[j].SetParticle("Proton-");
                h1->Fill(5);
            }
            else if (p < 0.995)
            { //K* into Pion+ Kaon-
                particles[j].SetParticle("Kstar");
                h1->Fill(6);
                particles[numberOfParticles +
                    extraParticles].SetParticle("Pion+");
                particles[numberOfParticles +
                    extraParticles + 1].SetParticle("Kaon-
                    ");
                particles[j].Decay2body(particles[
                    numberOfParticles + extraParticles],
                    particles[numberOfParticles +
                    extraParticles + 1]);
                extraParticles++;
                extraParticles++;
            }
            else
            { //K* into Pion- Kaon+
                particles[j].SetParticle("Kstar");
                h1->Fill(6);
                particles[numberOfParticles +
                    extraParticles].SetParticle("Pion-");
                particles[numberOfParticles +
                    extraParticles + 1].SetParticle("Kaon+
                    ");
                particles[j].Decay2body(particles[
                    numberOfParticles + extraParticles],
                    particles[numberOfParticles +
                    extraParticles + 1]);
                extraParticles++;
                extraParticles++;
            }

            double energy = particles[j].ParticleEnergy()
                ;
            h6->Fill(energy);
        }

        //finita la generazione devo andare a fare le
            operazioni di scorrimento sull'array

        for (int k = 0; k < numberOfParticles +
            extraParticles - 1; k++)
        {
            for (int n = k + 1; n < numberOfParticles +
                extraParticles; n++)
            {
```

```
                h7->Fill(particles[k].GetInvMass(
                    particles[n])); //h7 massa invariante
                    fra tutte le particelle generate in
                    ogni evento

                //h8 combinazione di carica discorde (
                    positive 0,2,4 ///||| negative 1,3,5)
                if (((particles[k].GetIndex() == 0 ||
                    particles[k].GetIndex() == 2 ||
                    particles[k].GetIndex() == 4) && (
                    particles[n].GetIndex() == 1 ||
                    particles[n].GetIndex() == 3 ||
                    particles[n].GetIndex() == 5)) ||
                     ((particles[k].GetIndex() == 1 ||
                         particles[k].GetIndex() == 3 ||
                         particles[k].GetIndex() == 5) && (
                         particles[n].GetIndex() == 0 ||
                         particles[n].GetIndex() == 2 ||
                         particles[n].GetIndex() == 4)))
                {
                     h8->Fill(particles[k].GetInvMass(
                         particles[n]));
                }

                //h9 combinazione di carica concorde (
                    positive 0,2,4 ///||| negative 1,3,5)
                if (((particles[k].GetIndex() == 0 ||
                    particles[k].GetIndex() == 2 ||
                    particles[k].GetIndex() == 4) && (
                    particles[n].GetIndex() == 0 ||
                    particles[n].GetIndex() == 2 ||
                    particles[n].GetIndex() == 4)) ||
                     ((particles[k].GetIndex() == 1 ||
                         particles[k].GetIndex() == 3 ||
                         particles[k].GetIndex() == 5) && (
                         particles[n].GetIndex() == 1 ||
                         particles[n].GetIndex() == 3 ||
                         particles[n].GetIndex() == 5)))
                {
                     h9->Fill(particles[k].GetInvMass(
                         particles[n]));
                }

                //h10 P+/K-, P-/K+ (0,3   1,2)
                if (((particles[k].GetIndex() == 0) && (
                    particles[n].GetIndex() == 3)) ||
                     ((particles[k].GetIndex() == 1) && (
                         particles[n].GetIndex() == 2)))
                {
                     h10->Fill(particles[k].GetInvMass(
```

```
                    particles[n]));
                h10b->Fill(particles[k].GetInvMass(
                    particles[n]));
            }

            //h11 P+/K+, P-/K- (0,2  1,3)
            if (((particles[k].GetIndex() == 0) && (
                particles[n].GetIndex() == 2)) ||
                ((particles[k].GetIndex() == 1) && (
                    particles[n].GetIndex() == 3)))
            {
                h11->Fill(particles[k].GetInvMass(
                    particles[n]));
                h11b->Fill(particles[k].GetInvMass(
                    particles[n]));
            }
        }
    }

    if (extraParticles != 0)
    {
        for (int j = 0; j < extraParticles; j += 2)
        {
            h12->Fill(particles[numberOfParticles + j
                ].GetInvMass(particles[
                numberOfParticles + 1 + j]));
            h12b->Fill(particles[numberOfParticles +
                j].GetInvMass(particles[
                numberOfParticles + 1 + j]));
        }
    }
}

std::cout << "Generation completed\n\n";

gStyle->SetOptFit(1);

c1->cd(1);
h1->Draw();
c1->cd(2);
h2->Draw();
c1->cd(3);
h3->Draw();
c1->cd(4);
h4->Draw();
c1->cd(5);
h5->Draw();
c1->cd(6);
h6->Draw();
c2->cd(1);
```

```
    h7->Draw();
    c2->cd(2);
    h8->Draw();
    c2->cd(3);
    h9->Draw();
    c2->cd(4);
    h10->Draw();
    c2->cd(5);
    h11->Draw();

    std::cout << "\033[0;31mGAUSSIAN FITS OF K* 
        RESONANCES\033[0m" << '\n';
    auto fit6a = new TF1("Fit K* Events", "gaus", 0.60,
        1.30); //relativo fit
    h12->Fit(fit6a);

    c2->cd(6);
    h12->Draw();

    std::cout << '\n';
    c1->Print("c1.pdf", "pdf");
    c2->Print("c2.pdf", "pdf");
    std::cout << '\n';

    //parte 3
    //9.1 check ingressi -> corretti

    //9.2 check ingressi di h1
    double PionPCount = h1->GetBinContent(1);
    double PionNCount = h1->GetBinContent(2);
    double KaonPCount = h1->GetBinContent(3);
    double KaonNCount = h1->GetBinContent(4);
    double ProtonPCount = h1->GetBinContent(5);
    double ProtonNCount = h1->GetBinContent(6);
    double KstarCount = h1->GetBinContent(7);
    double const totalGenerated = numberOfParticles *
        events;
    std::cout << "\nGenerated particles:\n";
    std::cout << "Pion+   are " << PionPCount /
        totalGenerated << "   " << h1->GetBinError(1) /
        totalGenerated << ",          0.40 expected\n";
    std::cout << "Pion-   are " << PionNCount /
        totalGenerated << "   " << h1->GetBinError(2) /
        totalGenerated << ",          0.40 expected\n";
    std::cout << "Kaon+   are " << KaonPCount /
        totalGenerated << "   " << h1->GetBinError(3) /
        totalGenerated << ",          0.05 expected\n";
    std::cout << "Kaon-   are " << KaonNCount /
        totalGenerated << "   " << h1->GetBinError(4) /
        totalGenerated << ",          0.05 expected\n";
```

```
    std::cout << "Proton+ are " << ProtonPCount /
        totalGenerated << " " << h1->GetBinError(5) /
        totalGenerated << ",          0.045 expected\n";
    std::cout << "Proton- are " << ProtonNCount /
        totalGenerated << " " << h1->GetBinError(6) /
        totalGenerated << ",          0.045 expected\n";
    std::cout << "K*      are " << KstarCount /
        totalGenerated << " " << h1->GetBinError(7) /
        totalGenerated << ",          0.01 expected\n\n";

    //9.3 angle fit
    TCanvas *c3 = new TCanvas("c3", "Fit angles");
    c3->Divide(2, 1);
    auto fit1 = new TF1("Fit azimuthal angle", "pol0", 0,
         2 * pi);
    auto fit2 = new TF1("Fit polar angle", "pol0", 0, pi)
        ;
    std::cout << "\033[0;31mAZIMUTHAL AND POLAR ANGLES
        FITS\033[0m" << '\n';
    c3->cd(1);
    h2->Fit(fit1);
    c3->cd(2);
    h3->Fit(fit2);
    std::cout << '\n';
    c3->Print("c3.pdf", "pdf");
    std::cout << '\n';

    //9.4 exponential fit
    TCanvas *c4 = new TCanvas("c4", "Impulse Fit");
    auto fit3 = new TF1("Fit impulse", "expo", 0, 5.5);
    c4->cd(1);
    std::cout << "\033[0;31mIMPULSE EXPONENTIAL FIT\033[0
        m" << '\n';
    h4->Fit(fit3);
    std::cout << '\n';
    c4->Print("c4.pdf", "pdf");
    std::cout << '\n';

    //10
    TCanvas *c5 = new TCanvas("c5", "Impulse Fit");
    c5->Divide(2, 3);
    c5->cd(1);
    h10->Draw();
    c5->cd(2);
    h11->Draw();
    c5->cd(3);
    h12b->Draw();
    c5->cd(4);

    std::cout << "\033[0;31mGAUSSIAN FITS OF K* 
```

```
        RESONANCES (Pion/Kaon)\033[0m" << '\n';
    auto h13 = new TH1F("h13", "K* extracted from
        invariant masses (P/K)", 400, 0, 5);
    h13->Sumw2();
    h13->Add(h10, h11, 1, -1);
    auto fit5a = new TF1("Fit subtraction discordant/
        concordant Pion Kaon", "gaus", 0.60, 1.30); //
        relativo fit
    h13->Fit(fit5a);

    c5->cd(5);
    std::cout << "\033[0;31mGAUSSIAN FITS OF K*
        RESONANCES (Particles by charge)\033[0m" << '\n';
    auto h14 = new TH1F("h14", "K* extracted from
        invariant masses (Charge)", 400, 0, 5);
    h14->Sumw2();
    h14->Add(h8, h9, 1, -1);
    auto fit5c = new TF1("Fit subtraction invariant
        masses by charge", "gaus", 0.65, 1.15); //relativo
         fit
    h14->Fit(fit5c);

    std::cout << '\n';
    c5->Print("c5.pdf", "pdf");
    std::cout << '\n';

    //fine parte 3

    file->Write();
    file->Close();

    //end notification
    std::cout << '\a';

    //time taken for execution
    auto endTime = std::chrono::system_clock::now();
    std::chrono::duration<double> diff = endTime -
        startTime;
    std::cout << "\nTime required: " << diff.count() << "
        s\n";
}
```

5.8 macro.c

```
#include "TH1F.h"
#include "TCanvas.h"
#include "TROOT.h"
#include "TFile.h"
#include "TStyle.h"
```

```
int main(){

    double pi = M_PI;

    TFile *file = TFile::Open("data.root");
    TFile *flMC_file = new TFile("locale.root");

    gStyle->SetOptFit(1);

    auto c1bis = new TCanvas();
    auto c2bis = new TCanvas();
    c1bis->Divide(2, 2);
    c2bis->Divide(1, 3);

    auto h1bis = new TH1F("h1", "Particles", 7, 0, 7);
    auto h2bis = new TH1F("h2", "Azimuthal angle
        distribution", 500, 0, 2 * pi);
    auto h3bis = new TH1F("h3", "Polar angle distribution
        ", 250, 0, pi);
    auto h4bis = new TH1F("h4", "Impulse", 800, 0, 5.5);

    auto h12bis = new TH1F("h12", "From K* Decay", 400,
        0.5, 1.2);
    auto h13bis = new TH1F("h13", "K* extracted from
        invariant masses (P/K)", 400, 0, 5);
    auto h14bis = new TH1F("h14", "K* extracted from
        invariant masses (Charge)", 400, 0, 5);

    h1bis = (TH1F*)file->Get("h1");
    h2bis = (TH1F*)file->Get("h2");
    h3bis = (TH1F*)file->Get("h3");
    h4bis = (TH1F*)file->Get("h4");

    h12bis = (TH1F*)file->Get("h12");
    h13bis = (TH1F*)file->Get("h13");
    h14bis = (TH1F*)file->Get("h14");

    flMC_file->GetList()->Write();

    h1bis->GetYaxis()->SetTitle("Occorrenze");
    h2bis->GetYaxis()->SetTitle("Occorrenze");
    h3bis->GetYaxis()->SetTitle("Occorrenze");
    h4bis->GetYaxis()->SetTitle("Occorrenze");

    h12bis->GetYaxis()->SetTitle("Occorrenze");
    h13bis->GetYaxis()->SetTitle("Occorrenze");
    h14bis->GetYaxis()->SetTitle("Occorrenze");

    h1bis->GetXaxis()->SetTitle("Popolazioni");
    h2bis->GetXaxis()->SetTitle("Angolo Azimutale");
```

```
    h3bis->GetXaxis()->SetTitle("Angolo Polare");
    h4bis->GetXaxis()->SetTitle("Modulo Impulso");

    h12bis->GetXaxis()->SetTitle("Massa Invariante");
    h13bis->GetXaxis()->SetTitle("Massa Invariante");
    h14bis->GetXaxis()->SetTitle("Massa Invariante");

    c1bis->cd(1);
    h1bis->Draw();
    c1bis->cd(4);
    h2bis->Draw();
    c1bis->cd(3);
    h3bis->Draw();
    c1bis->cd(2);
    h4bis->Draw();

    c2bis->cd(1);
    h12bis->Draw();
    c2bis->cd(2);
    h13bis->Draw();
    c2bis->cd(3);
    h14bis->Draw();

    c1bis->Print("FINALE1.pdf", "pdf");
    c2bis->Print("FINALE2.pdf", "pdf");

}
```

6 Specifiche

- Sistema Operativo: Windows 10 Home 64 bit
- Editor C++: Visual Studio Code 1.51.1
- Sottosistema WSL: 20.04.1 LTS
- Compilatore: g++ 4:9.3.0-1ubuntu2
- Root: Root 6-22-02
- Formattatore di codice: Clang-Format 1.9.0
- Editor LaTeX: TeXstudio 2.12.22