Simulazione particelle

Doria Stefano, Fioralli Rossella, Luciano Giuseppe

1) Descrizione:

1)Introduzione:

lo scopo del programma consiste nel creare il prototipo di una simulazione atta ad analizzare il contenuto di eventi risultanti da collisioni di particelle elementari ispirandosi a quello che potrebbe essere un reale esperimento di fisica delle particelle. Nello specifico, l'obiettivo principale del progetto era quello di evidenziare in modo efficace gli effetti derivanti dal decadimento dei kaoni* filtrando il contributo del fondo, ovvero quello legato alla presenza di combinazioni accidentali, che avrebbe altrimenti ostacolato lo studio delle proprietà di tali particelle. Per farlo è stato necessario generare un campione di indagine sufficientemente numeroso sul quale sono state effettuate delle indagini statistiche che hanno permesso, mediante numerosi confronti nella distribuzione di massa invariante, di raggiungere lo scopo prefissato esaltando gli effetti del decadimento. Inoltre per verificare il corretto funzionamento del codice sono stati inseriti i dati relativi esclusivamente al prodotto del decadimento in un istogramma avendo così modo di confrontare i valori derivanti dalle rispettive indagini statistiche con quelli noti e propri della particella.

2) Struttura del codice:

Il codice da noi sviluppato è formato da tre file header che contengono le classi e le funzioni necessarie per la generazione e l'analisi dei dati. Nel file particletype.h si trova la classe ParticleType che definisce un tipo di particella i cui attributi sono: il nome, la massa e la carica. Per questa classe abbiamo implementato un costruttore, una serie di metodi pubblici Getters per accedere ai membri privati e infine un metodo Print() che invece stampa sul terminale i valori dei membri privati di tale oggetto. Invece nel file resonancetype.h si trovano le dichiarazioni della classe ResonanceType, che è una classe derivata di ParticleType e rappresenta le particelle soggette a decadimento radioattivo, ovvero quelle che hanno una larghezza di risonanza non trascurabile. Rispetto alla classe madre viene aggiunta agli attributi la vita media della particella e vengono modificati, quando necessario, i metodi già implementati in ParticleType.

Infine, nel file particle.h si trova la classe Particle, che rappresenta la singola particella generata durante l'esecuzione del programma. I suoi attributi sono le tre componenti cartesiane della quantità di moto e il nome del tipo di particella. Contiene anche 3 membri static: il numero massimo di tipologie di particelle, il numero di particelle generate e una mappa di ParticleType, che rappresenta i tipi di particelle inizializzati. Tra i metodi pubblici vi sono i costruttori, i Getters ed i Setters necessari per creare e manipolare un oggetto di tipo Particle. Tra i metodi ci sono anche AddParticleType() che, con i parametri passati alla funzione, crea un nuovo puntatore ad un oggetto ParticleType e lo aggiunge alla mappa, PrintAllTypes(), che stampa tutti i tipi di particella inseriti e Decay2Body(), che simula attraverso un generatore random la cinematica del decadimento di una particella. Tutti questi metodi pubblici utilizzano a loro volta dei metodi privati come GetModuleP() o Boost(), in modo che il codice di queste funzioni possa essere riutilizzato in ogni metodo della classe.

3) Generazione:

Come già esposto programma si propone di simulare un reale esperimento di fisica delle alte energie nel quale si osservano ripetutamente le interazioni fra un campione numeroso di particelle al fine di riuscire a estrapolare dati sulle stesse mediante analisi statistiche. Per ottenere il numero di eventi desiderato, ovvero i momenti entro i quali le particelle interagiscono, è stato utilizzato un ciclo da centomila iterazioni all'interno del quale è contenuto un secondo ciclo da cento repliche che permette la generazione delle singole particelle secondo i parametri indicati.

Infatti, per quanto riguarda la distribuzione del tipo di particelle, è stata utilizzata la funzione "rndm()" di ROOT per generare un numero casuale con distribuzione uniforme compreso fra zero e uno. In tal modo, confrontando mediante una serie di strutture "if" il valore della singola estrazione con la probabilità di osservare una certa particella, abbiamo assegnato il nome alla stessa attribuendone così indirettamente tutte le proprietà fisiche specifiche. Come richiesto è stato trattato un modello semplificato nel quale presenziano solo tre tipologie di particelle con relative cariche positive e negative ovvero: protoni, pioni e kaoni presenti rispettivamente con una percentuale del 4,5%, 40%, 5% per singola carica con l'aggiunta di un ulteriore corpuscolo dotato una larghezza di risonanza non trascurabile, la k* presente al 1%. Sarà poi quest'ultimo che appena generato verrà fatto decadere in due ulteriori particelle, ovvero pioni e kaoni, la cui carica è opposta e decisa stocasticamente grazie alla generazione di un'ulteriore variabile casuale uniforme. Analogamente per la quantità di moto è stata generata una variabile secondo una distribuzione esponenziale di media pari a -1 per il modulo e due variabili stocastiche uniformi comprese fra $[0,\pi]$ e $[0,2\pi]$ per simulare gli angoli azimutali e polari propri del sistema di coordinate sferiche, così da poter effettuare le trasformazioni necessarie per passare a quelle cartesiane come da implementazione degli oggetti di tipo Particle. Così facendo abbiamo generato complessivamente un numero di particelle superiore a 10^7 che può variare in base al numero di k* rilevati.

4) Analisi:

Dalla tabella 1 e nella figura 1 si osserva che il numero di particelle generate di ogni tipo è compatibile con quanto atteso. Poi, nella stessa figura e nella tabella 2 viene mostrato che gli angoli e l'impulso risultano correttamente generati, secondo distribuzioni rispettivamente uniforme, una tra 0 e 2π e l'altra tra 0 e π, ed esponenziale, con media 1 e decrescente.

Per estrarre il segnale della risonanza si sono confrontati gli istogrammi dei valori della massa invariante calcolati da particelle con carica discorde e di quelli ottenuti da particelle di carica concorde. Per isolare il segnale della risonanza che, essendo elettricamente neutra, decadendo genera particelle di carica opposta, si fa la differenza tra i due istogrammi, il che permette di eliminare il contributo del fondo di combinazioni casuali ed evidenziare una distribuzione gaussiana corrispondente ai prodotti del decadimento ed avente picco in corrispondenza della massa della risonanza. La media di questa distribuzione corrisponde alla massa della K* e la sigma alla sua larghezza.

Sapendo poi che in ogni decadimento vengono prodotti un pione e un kaone di segno opposto, si può giungere ad un risultato più accurato confrontando solo le combinazioni pione-kaone con carica uguale e quelle con carica opposta. Nella tabella 4 e nella figura 2 vengono mostrati i risultati dell'analisi effettuata per trovare la risonanza, con il risultato che i valori di massa e di larghezza della K* calcolati con un fit ad una gaussiana della distribuzione delle masse invarianti delle coppie pione-kaone prodotte da un decadimento sono compatibili, con un fattore di copertura 3, con i valori ottenuti dai fit dell'istogramma

relativo a tutte le particelle e di quello relativo solo a pioni e kaoni, con i valori che risultano più vicini a quelli reali per quest'ultimo.

2) Tabelle:

Specie	**Occorrenze osservate**	**Occorrenze attese**
π+	$(399 \pm 2) \times 10^3$	40×10^6
π-	$(400 \pm 2) \times 10^3$	40×10^6
k+	$(49.9 \pm 0.7) \times 10^3$	5×10^6
k-	$(49.9 \pm 0.7) \times 10^3$	5×10^6
p+	$(4.5 \pm 0.7) \times 10^3$	4.5×10^6
p-	$(4.5 \pm 0.7) \times 10^3$	4.5×10^6
k*	$(1.0 \pm 0.3) \times 10^3$	1×10^6

Tab 1: la tabella mostra le occorrenze di ogni tipo di particella con le relative incertezze ottenute mediante la valutazione degli errori di conteggio forniti dal metodo di root. Sull'ultima colonna invece vengono mostrati i valori di occorrenze attese ottenute mediante valutazioni sulla probabilità di osservare determinate particelle ed il numero totale delle stesse. Si osserva un accordo fra quanto previsto e quanto osservato visto che tutte le previsioni ricadono nell'intervallo di tolleranza.

Distribuzione	**Parametri del fit**	χ^2	***DOF***	χ^2/NDF
Fit a distribuzione angolo azimutale	9999 +/- 3	934.302	999	0.935238
Fit a distribuzione angolo polare	9999 +/- 3	1061.52	999	1.06259
Fit a distribuzione modulo impulso	10 007 +/- 0.0003	965.052	998	0.996986

Tab 2: la tabella nella prima colonna mostra i fit degli istogrammi delle occorrenze rispetto a funzioni costanti per quanto riguarda gli angoli, ed esponenziale per il modulo dell'impulso. I valori delle prime due righe sono prossimi al valore 10'000 ovvero il numero di angoli generati rapportati al numero di bin. Nelle successive due colonne sono poi indicati rispettivamente i valori del χ^2, dei *gradi di libertà e del loro rapporto*

Grafico	**Media (GeV)**	**Sigma (GeV)**	**Ampiezza**	χ^2/DOF
Massa Invariante K* vere (fit gauss)	0.89157±0.00016	0.04982±0.00011	3596±14	0.988
Massa Invariante da differenza combinazioni carica discorde e concorde (fit gauss)	0.9014±0.0042	0.0460±0.0045	(544±43)*10	0.978
Massa Invariante da differenza combinazioni πK carica discorde e concorde (fit gauss)	0.8918±0.0019	0.0500±0.0019	(476±16)*10	0.938

Tab 3: La tabella riporta i parametri ottenuti eseguendo il fit gaussiano di tre istogrammi di distribuzioni di masse invarianti: coppie pione-kaone da un decadimento; differenza tra masse invarianti di tutte le coppie di particelle con carica opposta e le coppie di particelle con stessa carica; differenza tra le masse invarianti di coppie pione-kaone con carica opposta e coppie pione-kaone con stessa carica. La media della distribuzione corrisponde alla massa della K, la sigma alla sua larghezza, mentre l'ampiezza rappresenta il numero di eventi contenuti nel bin corrispondente al massimo della distribuzione.*

3) Figure:

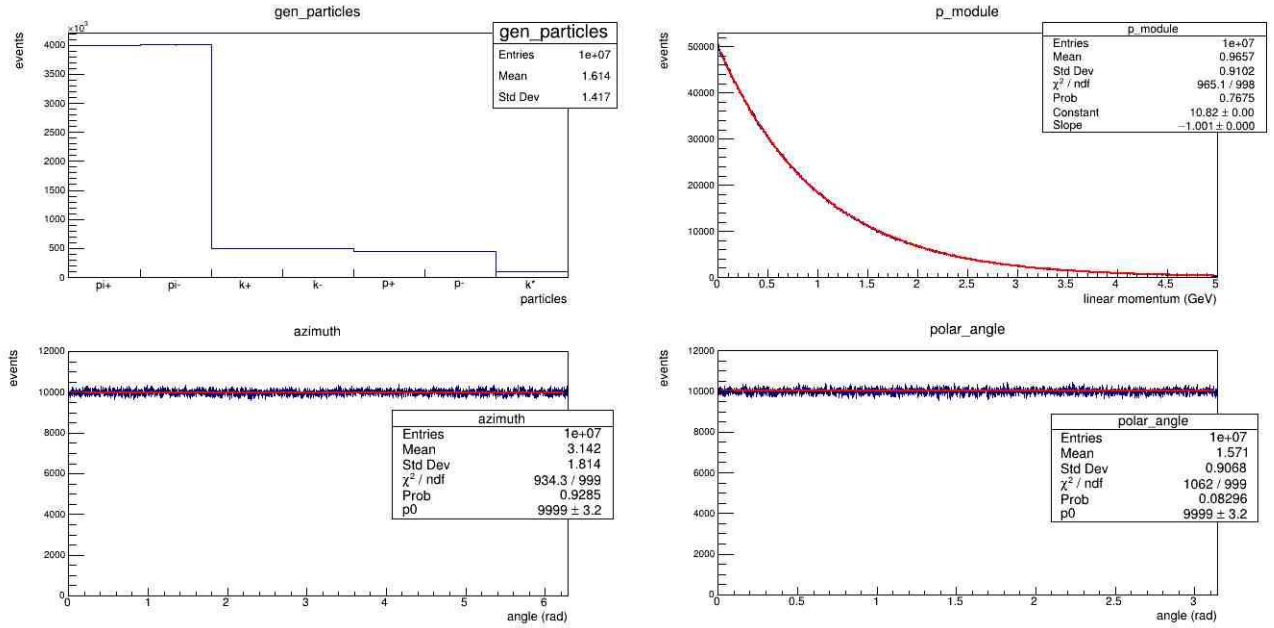

Fig 1: in figura è riportato in alto a sinistra l'istogramma in cui viene associato a ciascun tipo di particella un certo valore: in particolare è assegnato il valore di 0.5 ai pioni positivi, 1.5 ai pioni negativi, 2.5 ai kaoni positivi, 3.5 ai kaoni negativi, 4.5 ai protoni positivi, 5.5 ai protoni negativi e infine 6.5 alle resonance. Come ci aspettiamo abbiamo all'incirca lo stesso numero di ingressi per le coppie di bin che rappresentano lo stesso tipo di particella ma di carica opposta. Successivamente negli altri tre istogrammi sono mostrate la distribuzione esponenziale del modulo dell'impulso in alto a destra, la distribuzione uniforme dell'angolo azimutale in basso a sinistra e la distribuzione dell'angolo polare in basso a destra.

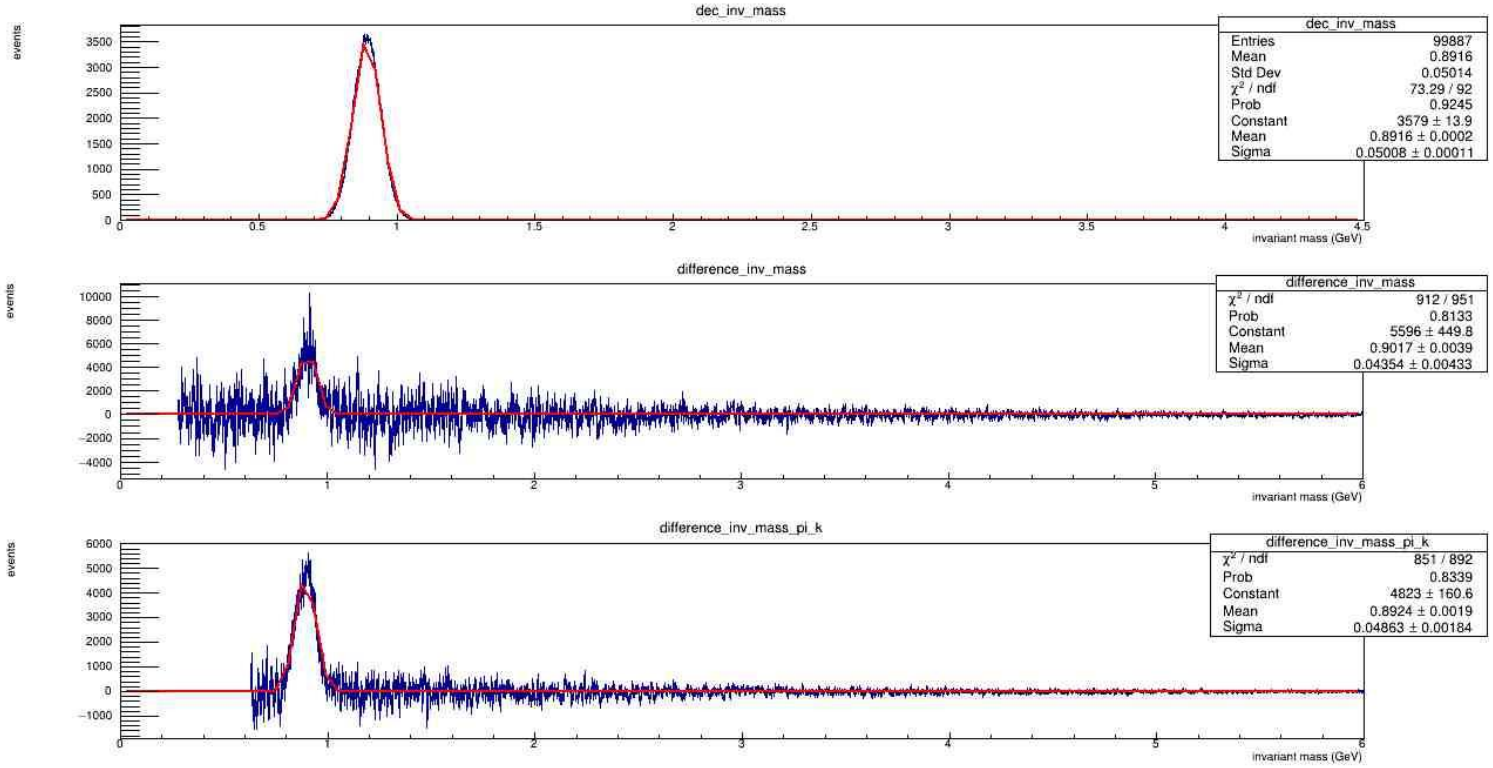

Fig 2: la figura mostra nella parte superiore l'istogramma di controllo nel quale sono presenti solamente i valori della massa invariante delle particelle generate dal decadimento della k. Gli istogrammi al di sotto invece mostrano le occorrenze dei valori delle masse invarianti ottenuti dalle sottrazioni fra le particelle di cariche opposte e da pioni e kaoni con segni di carica discorde. Si osserva come il valore medio di ambedue gli istogrammi sottostanti, che rappresenta la massa della k*, sia compatibile con quello di controllo. Analogamente anche la varianza, che indica il valore della larghezza di risonanza della k*, risulta essere compatibile con il primo in entrambi i casi se si considera un fattore di copertura 3 degli errori statistici.*

4) Appendice:

Listato del codice:

- Particletipe.cpp:

```
#include "particletype.h"

#include <iostream>

ParticleType::ParticleType(std::string name, double mass, int charge)
   : name_{name}, mass_{mass}, charge_{charge} {};

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

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

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

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

void ParticleType::Print() const {
  std::cout << name_ << " "
         << "mass: " << mass_ << " kg "
         << "charge: " << charge_ << " e" << '\n';
}
```

- Particletipe.h:

```
#ifndef PARTICLE_TYPE_H
#define PARTICLE_TYPE_H

#include <string>

class ParticleType
{
public:
  std::string GetName() const; // forse si può togliere
  double GetMass() const;
  int GetCharge() const;
  virtual double GetWidth() const;
  virtual void Print() const;
  explicit ParticleType(std::string name, double mass, int charge);

private:
  std::string const name_;
  double const mass_;
  int const charge_;
};
#endif
```

- Resonancetipe.cpp:

```
#include "resonancetype.h"
#include "string"

ResonanceType::ResonanceType(std::string name, double mass, int charge, double width=0)
    : ParticleType(name, mass, charge), width_{width} {}

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

double ResonanceType::GetWidth() const { return width_; }
```

- Resonancetipe.h:

```
#ifndef RESONANCE_TYPE_H
#define RESONANCE_TYPE_H

#include "particle.h"
#include "particletype.h"
#include "string"

class ResonanceType final : public ParticleType {
public:
  virtual void Print() const override;

   double GetWidth() const override;

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

private:
  double const width_;
};

#endif
```

- Particle.cpp:

```
#include "particle.h"
#include "unordered_map"
#include <algorithm>
#include <cmath>
#include <cstdlib> //for RAND_MAX
#include <iostream>

// inizializzation static member
int const Particle::maxNumParticleType_ = 10;
int Particle::NParticleType_ = 0;
std::unordered_map<std::string, ParticleType *> Particle::particleTypes_;
```

```
// costructor
Particle::Particle(std::string name, double px = 0, double py = 0,
                   double pz = 0)
    : name_{name}, px_{px}, py_{py}, pz_{pz} {
  auto search = particleTypes_.find(name);
  if (search == particleTypes_.end()) {
    std::cout << "Particle type not found" << '\n';
  }
}

// Getters
std::string Particle::GetName() const { return name_; }
double Particle::GetPx() const { return px_; }
double Particle::GetPy() const { return py_; }
double Particle::GetPz() const { return pz_; }
double Particle::GetMass() const {
  auto search = particleTypes_.find(name_);
  return search->second->GetMass();
}
int Particle::GetCharge() const {
  auto search = particleTypes_.find(name_);
  return search->second->GetCharge();
}
double Particle::GetEnergy() const {
  double const m2 = std::pow(GetMass(), 2);
  double const p2 = std::pow(GetModuleP(), 2);
  return std::sqrt(m2 + p2);
}
double Particle::GetInvariantMass(Particle &other_dau) const {
  double const sum_e2{std::pow(other_dau.GetEnergy() + GetEnergy(), 2)};
  double const pxtot = GetPx() + other_dau.GetPx();
  double const pytot = GetPy() + other_dau.GetPy();
  double const pztot = GetPz() + other_dau.GetPz();
  double const sum_p2{std::pow(pxtot, 2) + std::pow(pytot, 2) +
                      std::pow(pztot, 2)};
  if (sum_e2 < sum_p2) {
    std::cout << "error in the energy distribution"
              << "\n";
  }
  return std::sqrt(sum_e2 - sum_p2);
}

double Particle::GetModuleP() const {
  double const p2 = std::pow(px_, 2) + std::pow(py_, 2) + std::pow(pz_, 2);
  return std::sqrt(p2);
}

// Setters
void Particle::SetName(std::string new_name) {
```

```
  auto search = particleTypes_.find(new_name);
  if (search != particleTypes_.end()) {
    name_ = new_name;
  }
}

void Particle::SetP(double px, double py, double pz) {
  px_ = px;
  py_ = py;
  pz_ = pz;
}

void Particle::Print() const {
  std::cout << "particle index:" << FindParticle(name_) << '\n'
          << "particle type name: " << name_ << '\n'
          << "linear momentum (GeV): (" << px_ << ", " << py_ << ", " << pz_
          << ")" << '\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();

  auto search = particleTypes_.find(name_);
  if (search != particleTypes_.end()) { // add width effect

    // gaussian random numbers

    float x1, x2, w, y1;

    double invnum = 1. / RAND_MAX;
    do {
      x1 = 2.0 * std::rand() * invnum - 1.0;
      x2 = 2.0 * std::rand() * invnum - 1.0;
      w = x1 * x1 + x2 * x2;
    } while (w >= 1.0);

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

    massMot += particleTypes_[name_]->GetWidth() * y1;
  }
```

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

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

 double energy =
    std::sqrt(px_ * px_ + py_ * py_ + pz_ * pz_ + massMot * massMot);

 double bx = px_ / energy;
 double by = py_ / energy;
 double bz = pz_ / 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();

 // Boost this Lorentz vector
 double b2 = bx * bx + by * by + bz * bz;
 double gamma = 1.0 / std::sqrt(1.0 - b2);
 double bp = bx * px_ + by * py_ + bz * pz_;
 double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

 px_ += gamma2 * bp * bx + gamma * bx * energy;
 py_ += gamma2 * bp * by + gamma * by * energy;
 pz_ += gamma2 * bp * bz + gamma * bz * energy;
}
```

```
void Particle::AddParticleType(std::string name, double mass, int charge,
                               double width) {
  auto search = particleTypes_.find(name);
  if (search != particleTypes_.end()) {
    return;
  } else {
    if (width == 0) {
      ParticleType *particle = new ParticleType(name, mass, charge);
      particleTypes_[name] = particle;
    } else {
      ResonanceType *particle = new ResonanceType(name, mass, charge, width);
      particleTypes_[name] = particle;
    }
  }
}

void Particle::PrintAllTypes() {
  std::for_each(particleTypes_.begin(), particleTypes_.end(),
                [](std::pair<const std::string, ParticleType *> p) {
                  p.second->Print();
                });
}

int Particle::FindParticle(std::string name) const {
  return std::distance(particleTypes_.begin(), particleTypes_.find(name));
}
```

- Particle.h:

```
#ifndef PARTICLE_H
#define PARTICLE_H

#include <unordered_map>
#include <iostream>
#include <string>

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


class Particle {
public:
  Particle() = default;
  Particle(std::string name, double px, double py, double pz);

    std::string GetName() const; //const members
    double GetPx() const;
    double GetPy() const;
    double GetPz() const;
    double GetMass() const;
```

```
  int GetCharge() const;
  double GetEnergy() const;
  double GetInvariantMass(Particle& other_particle) const;
  int Decay2Body(Particle &dau1,Particle &dau2) const;
  void Print() const;

  void SetName(std::string new_name);
  void SetP(double px,double py,double pz);

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


private:
  static const int maxNumParticleType_;
  static std::unordered_map<std::string, ParticleType *> particleTypes_;
  static int NParticleType_;
  std::string name_;
  double px_;
  double py_;
  double pz_;

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

#endif
```

- Main.cpp:

```
#include <TCanvas.h>
#include <algorithm>
#include <cmath>
#include <iterator>
#include <TFile.h>
#include <TH1.h>
#include <TRandom.h>
#include "particle.h"
#include "particletype.h"
#include "resonancetype.h"

int main()
{
  TH1::AddDirectory(kFALSE);
  TFile *file = new TFile("histo.root", "RECREATE");
  gRandom->SetSeed(200769);
  Particle::AddParticleType("pion +", 0.13957, 1);
```

```
Particle::AddParticleType("pion -", 0.13957, -1);
Particle::AddParticleType("kaon +", 0.49367, 1);
Particle::AddParticleType("kaon -", 0.49367, -1);
Particle::AddParticleType("proton +", 0.93827, 1);
Particle::AddParticleType("proton -", 0.93827, -1);
Particle::AddParticleType("resonance", 0.89166, 0, 0.050);
std::array<Particle, 120> EventParticles{};

TH1F *gen_particles = new TH1F("gen_particles", "gen_particles", 7, 0, 7);

TH1F *azimuth = new TH1F("azimuth", "azimuth", 1000, 0., 2 * M_PI);

TH1F *polar_angle = new TH1F("polar_angle", "polar_angle", 1000, 0., M_PI);

TH1F *p_module = new TH1F("p_module", "p_module", 1000, 0, 5);

TH1F *p_trans = new TH1F("p_trans", "p_trans", 1000, 0, 5);

TH1F *energy = new TH1F("energy", "energy", 1000, 0, 6);

TH1F *all_inv_mass = new TH1F("all_inv_mass", "all_inv_mass", 1000, 0, 6);

all_inv_mass->Sumw2();

TH1F *same_charge_inv_mass =
   new TH1F("same_charge_inv_mass", "same_charge_inv_mass", 1000, 0, 6);

same_charge_inv_mass->Sumw2();

TH1F *opposite_charge_inv_mass = new TH1F(
   "opposite_charge_inv_mass", "opposite_charge_inv_mass", 1000, 0, 6);
opposite_charge_inv_mass->Sumw2();
TH1F *pi_k_same = new TH1F("pi_k_same_charge_inv_mass",
                    "pi_k_same_charge_inv_mass", 1000, 0, 6);
pi_k_same->Sumw2();
TH1F *pi_k_opposite = new TH1F("pi_k_opposite_charge_inv_mass",
                      "pi_k_opposite_charge_inv_mass", 1000, 0, 6);
pi_k_opposite->Sumw2();
TH1F *dec_inv_mass = new TH1F("dec_inv_mass", "dec_inv_mass", 1000, 0, 4.5);
dec_inv_mass->Sumw2();

int star_num{0};

for (int i{0}; i < 1e5; i++)
{
  if (star_num > 20)
  {
    std::cout << "too much k* has been generated" << '\n';
  }
```

```
star_num = 0; // k* counter

for (int event{0}; event < 1e2; event++)
{
 bool is_star{false};                    // tells if a k* is found
 double type = gRandom->Rndm();          // type selector
 double phi = gRandom->Uniform(0., 2 * M_PI); // azimuth angle
 double theta = gRandom->Uniform(0., M_PI);   // polar angle
 double p_m = gRandom->Exp(1.);          // P module
 double px = p_m * std::cos(phi) * std::sin(theta);
 double py = p_m * std::sin(phi) * std::sin(theta);
 double pz = p_m * std::cos(theta);

 azimuth->Fill(phi); // filling histo
 polar_angle->Fill(theta);
 p_module->Fill(p_m);
 p_trans->Fill(std::sqrt(std::pow(px, 2) + std::pow(py, 2)));

 Particle p;
 Particle p1;
 Particle p2;

 if (type <= 0.4)
 {
  p.SetName("pion +");
  gen_particles->Fill(0.5);
 }

 else if (type <= 0.8)
 {
  p.SetName("pion -");
  gen_particles->Fill(1.5);
 }

 else if (type <= 0.85)
 {
  p.SetName("kaon +");
  gen_particles->Fill(2.5);
 }

 else if (type <= 0.90)
 {
  p.SetName("kaon -");
  gen_particles->Fill(3.5);
 }

 else if (type <= 0.945)
 {
  p.SetName("proton +");
```

```
  gen_particles->Fill(4.5);
}

else if (type <= 0.99)
{
  p.SetName("proton -");
  gen_particles->Fill(5.5);
}

else
{
  p.SetName("resonance");
  gen_particles->Fill(6.5);
  is_star = true;
  double decad{gRandom->Rndm()};
  p.SetP(px, py, pz);
  energy->Fill(p.GetEnergy());
  star_num += 1;
  if (decad <= 0.5)
  {
    p1.SetName("pion +");
    p2.SetName("kaon -");
  }
  else
  {
    p1.SetName("pion -");
    p2.SetName("kaon +");
  }
  int all_good{p.Decay2Body(p1, p2)};
  if (all_good != 0)
  {
    std::cout << "something went wrong during decay" << '\n'
          << "\n";
  }
  else
  {
    dec_inv_mass->Fill(p1.GetInvariantMass(p2));
  }
}
if (!is_star)
{
  p.SetP(px, py, pz);
  energy->Fill(p.GetEnergy());
  EventParticles[event + star_num] = p;
}
else
{
  EventParticles[event + star_num] = p1;
  EventParticles[event - 1 + star_num] = p2;
```

```
    }

    for (int compare{0}; compare < event + star_num; compare++)
    {
      Particle old_particle{EventParticles[compare]};
      if (is_star)
      {
        p = p1;
      }
      all_inv_mass->Fill(p.GetInvariantMass(old_particle));
      if (p.GetCharge() == old_particle.GetCharge())
      { // same charge
        same_charge_inv_mass->Fill(p.GetInvariantMass(old_particle));
        const bool first_cond{p.GetName() == "pion +" &&
                              old_particle.GetName() == "kaon +"};
        const bool second_cond{p.GetName() == "pion -" &&
                               old_particle.GetName() == "kaon -"};
        if (first_cond || second_cond)
        {
          pi_k_same->Fill(p.GetInvariantMass(old_particle));
        }
      }
      else
      { // different charge
        opposite_charge_inv_mass->Fill(p.GetInvariantMass(old_particle));
        const bool first_cond{p.GetName() == "pion +" &&
                              old_particle.GetName() == "kaon -"};
        const bool second_cond{p.GetName() == "pion -" &&
                               old_particle.GetName() == "kaon +"};
        if (first_cond || second_cond)
        {
          pi_k_opposite->Fill(p.GetInvariantMass(old_particle));
        }
      }
    }
  }
}

gen_particles->GetXaxis()->SetBinLabel(1, "pi+");
gen_particles->GetXaxis()->SetBinLabel(2, "pi-");
gen_particles->GetXaxis()->SetBinLabel(3, "k+");
gen_particles->GetXaxis()->SetBinLabel(4, "k-");
gen_particles->GetXaxis()->SetBinLabel(5, "p+");
gen_particles->GetXaxis()->SetBinLabel(6, "p-");
gen_particles->GetXaxis()->SetBinLabel(7, "k*");

azimuth->GetXaxis()->SetTitle("angle (rad)");
azimuth->GetXaxis()->SetTitleSize(0.045);
azimuth->GetYaxis()->SetTitle("events");
```

```
azimuth->GetYaxis()->SetTitleSize(0.045);
azimuth->GetYaxis()->SetRangeUser(0.,12000.);

polar_angle->GetXaxis()->SetTitle("angle (rad)");
polar_angle->GetXaxis()->SetTitleSize(0.045);
polar_angle->GetYaxis()->SetTitle("events");
polar_angle->GetYaxis()->SetTitleSize(0.045);
polar_angle->GetYaxis()->SetRangeUser(0.,12000.);

gen_particles->GetXaxis()->SetTitle("particles");
gen_particles->GetXaxis()->SetTitleSize(0.045);
gen_particles->GetYaxis()->SetTitle("events");
gen_particles->GetYaxis()->SetTitleSize(0.045);

dec_inv_mass->GetXaxis()->SetTitle("invariant mass (GeV)");
dec_inv_mass->GetXaxis()->SetTitleSize(0.045);
dec_inv_mass->GetYaxis()->SetTitle("events");
dec_inv_mass->GetYaxis()->SetTitleSize(0.045);

pi_k_opposite->GetXaxis()->SetTitle("invariant mass (GeV)");
pi_k_opposite->GetXaxis()->SetTitleSize(0.045);
pi_k_opposite->GetYaxis()->SetTitle("events");
pi_k_opposite->GetYaxis()->SetTitleSize(0.045);

pi_k_same->GetXaxis()->SetTitle("invariant mass (GeV)");
pi_k_same->GetXaxis()->SetTitleSize(0.045);
pi_k_same->GetYaxis()->SetTitle("events");
pi_k_same->GetYaxis()->SetTitleSize(0.045);

opposite_charge_inv_mass->GetXaxis()->SetTitle("invariant mass (GeV)");
opposite_charge_inv_mass->GetXaxis()->SetTitleSize(0.045);
opposite_charge_inv_mass->GetYaxis()->SetTitle("events");
opposite_charge_inv_mass->GetYaxis()->SetTitleSize(0.045);

same_charge_inv_mass->GetXaxis()->SetTitle("invariant mass (GeV)");
same_charge_inv_mass->GetXaxis()->SetTitleSize(0.045);
same_charge_inv_mass->GetYaxis()->SetTitle("events");
same_charge_inv_mass->GetYaxis()->SetTitleSize(0.045);

all_inv_mass->GetXaxis()->SetTitle("invariant mass (GeV)");
all_inv_mass->GetXaxis()->SetTitleSize(0.045);
all_inv_mass->GetYaxis()->SetTitle("events");
all_inv_mass->GetYaxis()->SetTitleSize(0.045);

energy->GetXaxis()->SetTitle("energy (GeV)");
energy->GetXaxis()->SetTitleSize(0.045);
energy->GetYaxis()->SetTitle("events");
energy->GetYaxis()->SetTitleSize(0.045);
```

```
 p_trans->GetXaxis()->SetTitle("linear momentum (GeV)");
 p_trans->GetXaxis()->SetTitleSize(0.045);
 p_trans->GetYaxis()->SetTitle("events");
 p_trans->GetYaxis()->SetTitleSize(0.045);

 p_module->GetXaxis()->SetTitle("linear momentum (GeV)");
 p_module->GetXaxis()->SetTitleSize(0.045);
 p_module->GetYaxis()->SetTitle("events");
 p_module->GetYaxis()->SetTitleSize(0.045);

 azimuth->Write();
 polar_angle->Write();
 p_module->Write();
 p_trans->Write();
 gen_particles->Write();
 energy->Write();
 all_inv_mass->Write();
 same_charge_inv_mass->Write();
 opposite_charge_inv_mass->Write();
 pi_k_same->Write();
 pi_k_opposite->Write();
 dec_inv_mass->Write();
 file->Close();
}
```

- Analise.cpp:

```
#include "TFile.h"
#include "TH1.h"
#include "TF1.h"
#include <TCanvas.h>
#include <iostream>

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

void analyse()
{
 TH1::AddDirectory(kFALSE);
 TFile *file = new TFile("histo.root");
 TH1F *gen_particles = (TH1F *)file->Get("gen_particles");
 TH1F *azimuth = (TH1F *)file->Get("azimuth");
 TH1F *polar_angle = (TH1F *)file->Get("polar_angle");
 TH1F *p_module = (TH1F *)file->Get("p_module");
```

```
TH1F *p_trans = (TH1F *)file->Get("p_trans");
TH1F *energy = (TH1F *)file->Get("energy");
TH1F *all_inv_mass = (TH1F *)file->Get("all_inv_mass");
TH1F *same_charge_inv_mass = (TH1F *)file->Get("same_charge_inv_mass");
TH1F *opposite_charge_inv_mass =
   (TH1F *)file->Get("opposite_charge_inv_mass");
TH1F *pi_k_same = (TH1F *)file->Get("pi_k_same_charge_inv_mass");
TH1F *pi_k_opposite = (TH1F *)file->Get("pi_k_opposite_charge_inv_mass");
TH1F *dec_inv_mass = (TH1F *)file->Get("dec_inv_mass");
file->Close();

if (gen_particles->GetEntries() == 1e7)
{
 std::cout << "entries in gen particles histo: ok" << '\n';
}
else
{
 std::cout << "entries in gen particles histo: unexpected value" << '\n';
}
if (azimuth->GetEntries() == 1e7)
{
 std::cout << "entries in azimuth histo: ok" << '\n';
}
else
{
 std::cout << "entries in azimuth histo: unexpected value" << '\n';
}
if (polar_angle->GetEntries() == 1e7)
{
 std::cout << "entries in polar angle histo: ok" << '\n';
}
else
{
 std::cout << "entries in polar angle histo: unexpected value" << '\n';
}
if (p_module->GetEntries() == 1e7)
{
 std::cout << "entries in p module histo: ok" << '\n';
}
else
{
 std::cout << "entries in p module histo: unexpected value" << '\n';
}
if (p_trans->GetEntries() == 1e7)
{
 std::cout << "entries in p trans histo: ok" << '\n';
}
else
{
```

```
   std::cout << "entries in p trans histo: unexpected value" << '\n';
  }
  if (energy->GetEntries() == 1e7)
  {
   std::cout << "entries in energy histo: ok" << '\n';
  }
  else
  {
   std::cout << "entries in energy histo: unexpected value" << '\n';
  }
  if (all_inv_mass->GetEntries() < 1e5 * 60 * 119 && //number of combinations of n (maximum)
     all_inv_mass->GetEntries() > 1e5 * 40 * 79)  //number of combinations of n (minimum)
  {
   std::cout << "entries in gen particles histo: ok" << '\n';
  }
  else
  {
   std::cout << "entries in gen particles histo: unexpected value" << '\n';
  }
  if (same_charge_inv_mass->GetEntries() < 1e5 * 70 * 71 && //two times sum of n numbers
(maximum)
     same_charge_inv_mass->GetEntries() > 1e5 * 45 * 44) //two times sum of n numbers
(minimum)
  {
   std::cout << "entries in same charge inv mass histo: ok" << '\n';
  }
  else
  {
   std::cout << "entries in same charge inv mass histo: unexpected value" <<
same_charge_inv_mass->GetEntries()
          << '\n';
  }
  if (opposite_charge_inv_mass->GetEntries() < 1e5 * 70 * 50 && //n positives times n negatives
(maximum)
     opposite_charge_inv_mass->GetEntries() > 1e5 * 45 * 45) //n positives times n negatives
(minimum)
  {
   std::cout << "entries in opposite charge inv mass histo: ok" << '\n';
  }
  else
  {
   std::cout << "entries in opposite charge inv mass histo: unexpected value" <<
opposite_charge_inv_mass->GetEntries()
          << '\n';
  }
  if (pi_k_same->GetEntries() < 1e5 * 60 * 25 * 2 && //two times n k same times n pi same
(maximum)
     pi_k_same->GetEntries() > 1e5 * 30 * 3 * 2) //two times n k same times n pi same (minimum)
  {
```

```
  std::cout << "entries in pi k same histo: ok" << '\n';
 }
 else
 {
  std::cout << "entries in pi k same histo: unexpected value" << '\n';
 }
 if (pi_k_opposite->GetEntries() < 1e5 * 60 * 25 * 2 && //two times n k opposite times n pi opposite
(maximum)
    pi_k_opposite->GetEntries() > 1e5 * 30 * 3 * 2) //two times n k opposite times n pi opposite
(minimum)
 {
  std::cout << "entries in pi k opposite histo: ok" << '\n';
 }
 else
 {
  std::cout << "entries in pi k opposite histo: unexpected value" << '\n';
 }
 if (dec_inv_mass->GetEntries() < 1e6 && dec_inv_mass->GetEntries() > 1e4)
 {
  std::cout << "entries in dec inv mass histo: ok" << '\n';
 }
 else
 {
  std::cout << "entries in dec inv mass histo: unexpected value" << '\n';
 }

 if (gen_particles->GetBinContent(1) - 3 * gen_particles->GetBinError(1) <=
      gen_particles->GetEntries() * 0.4 &&
   gen_particles->GetBinContent(1) + 3 * gen_particles->GetBinError(1) >=
      gen_particles->GetEntries() * 0.4)
 {
  std::cout << "n pi+: ok" << '\n';
 }
 else
 {
  std::cout << "n pi+: unexpected" << '\n';
 }
 if (gen_particles->GetBinContent(2) - 3 * gen_particles->GetBinError(2) <=
      gen_particles->GetEntries() * 0.4 &&
   gen_particles->GetBinContent(2) + 3 * gen_particles->GetBinError(2) >=
      gen_particles->GetEntries() * 0.4)
 {
  std::cout << "n pi-: ok" << '\n';
 }
 else
 {
  std::cout << "n pi-: unexpected" << '\n';
 }
 if (gen_particles->GetBinContent(3) - 3 * gen_particles->GetBinError(3) <=
```

```
      gen_particles->GetEntries() * 0.05 &&
   gen_particles->GetBinContent(3) + 3 * gen_particles->GetBinError(3) >=
      gen_particles->GetEntries() * 0.05)
{
 std::cout << "n k+: ok" << '\n';
}
else
{
 std::cout << "n k+: unexpected" << '\n';
}
if (gen_particles->GetBinContent(4) - 3 * gen_particles->GetBinError(4) <=
      gen_particles->GetEntries() * 0.05 &&
   gen_particles->GetBinContent(4) + 3 * gen_particles->GetBinError(4) >=
      gen_particles->GetEntries() * 0.05)
{
 std::cout << "n k-: ok" << '\n';
}
else
{
 std::cout << "n k-: unexpected" << '\n';
}
if (gen_particles->GetBinContent(5) - 3 * gen_particles->GetBinError(5) <=
      gen_particles->GetEntries() * 0.045 &&
   gen_particles->GetBinContent(5) + 3 * gen_particles->GetBinError(5) >=
      gen_particles->GetEntries() * 0.045)
{
 std::cout << "n p+: ok" << '\n';
}
else
{
 std::cout << "n p+: unexpected" << '\n';
}
if (gen_particles->GetBinContent(6) - 3 * gen_particles->GetBinError(6) <=
      gen_particles->GetEntries() * 0.045 &&
   gen_particles->GetBinContent(6) + 3 * gen_particles->GetBinError(6) >=
      gen_particles->GetEntries() * 0.045)
{
 std::cout << "n p-: ok" << '\n';
}
else
{
 std::cout << "n p-: unexpected" << '\n';
}
if (gen_particles->GetBinContent(7) - 3 * gen_particles->GetBinError(7) <=
      gen_particles->GetEntries() * 0.01 &&
   gen_particles->GetBinContent(7) + 3 * gen_particles->GetBinError(7) >=
      gen_particles->GetEntries() * 0.01)
{
 std::cout << "n k*: ok" << '\n';
```

```
}
else
{
  std::cout << "n k*: unexpected" << '\n';
}

gStyle->SetOptFit(1111);

TF1 *f1 = new TF1("f1", "[0]", 0, 10);
azimuth->Fit(f1);
std::cout << "azimuth: uniform fit parameter: " << f1->GetParameter(0)
        << "+/-" << f1->GetParError(0) << '\n';
std::cout << "ChiSquare/NDF; " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';
polar_angle->Fit(f1);
std::cout << "polar angle: uniform fit parameter: " << f1->GetParameter(0)
        << "+/-" << f1->GetParError(0) << '\n';
std::cout << "ChiSquare/NDF " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';

f1 = new TF1("f2", "expo(0)", 0, 1);
p_module->Fit(f1);
if ((-f1->GetParameter(1) - 1) / f1->GetParError(1) < 2 &&
    (-f1->GetParameter(1) - 1) / f1->GetParError(1) > -2)
{
  std::cout << "p module distrution is as expexted" << '\n';
}
else
{
  std::cout << "unexpexted results for p module distriution" << '\n';
}
std::cout << "p module: expo fit parameter: " << f1->GetParameter(1) << "+/-"
        << -f1->GetParError(1) << '\n';
std::cout << "ChiSquare/NDF " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';

TH1F *diff = new TH1F(
    "difference_inv_mass", "difference_inv_mass", 1000, 0, 6.);
diff->Sumw2();
diff->Add(opposite_charge_inv_mass, same_charge_inv_mass, 1., -1.);
TH1F *diff_pi_k = new TH1F(
    "difference_inv_mass_pi_k", "difference_inv_mass_pi_k", 1000, 0, 6.);
diff_pi_k->Sumw2();
diff_pi_k->Add(pi_k_opposite, pi_k_same, 1., -1.);

std::cout << "maximum: "
        << '\n'
        << "from diff opposite same all: "
        << diff->GetBinCenter(diff->GetMaximumBin())
```

```
        << '\n'
        << "from diff opposite same pi-k: "
        << diff_pi_k->GetBinCenter(diff_pi_k->GetMaximumBin())
        << '\n'
        << "from decay product: "
        << dec_inv_mass->GetBinCenter(dec_inv_mass->GetMaximumBin())
        << '\n';

f1 = new TF1("f3", "gaus(0)", 0, 10);
dec_inv_mass->Fit(f1);
std::cout << "from decay products: "
        << '\n'
        << "k* mass: " << f1->GetParameter(1) << "+/-" << f1->GetParError(1)
        << '\n'
        << "k* width: " << f1->GetParameter(2) << "+/-" << f1->GetParError(2)
        << '\n'
        << "gauss width: " << f1->GetParameter(0) << "+/-" << f1->GetParError(0)
        << '\n';
std::cout << "ChiSquare/NDF " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';
diff->Fit(f1);
std::cout << "from difference opposite and same charge inv mass: "
        << '\n'
        << "k* mass: " << f1->GetParameter(1) << "+/-" << f1->GetParError(1)
        << '\n'
        << "k* width: " << f1->GetParameter(2) << "+/-" << f1->GetParError(2)
        << '\n'
        << "gauss width: " << f1->GetParameter(0) << "+/-" << f1->GetParError(0)
        << '\n';
std::cout << "ChiSquare/NDF " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';
diff_pi_k->Fit(f1);
std::cout << "from difference opposite and same charge pions and kaons inv mass: "
        << '\n'
        << "k* mass: " << f1->GetParameter(1) << "+/-" << f1->GetParError(1)
        << '\n'
        << "k* width: " << f1->GetParameter(2) << "+/-" << f1->GetParError(2)
        << '\n'
        << "gauss width: " << f1->GetParameter(0) << "+/-" << f1->GetParError(0)
        << '\n';
std::cout << "ChiSquare/NDF " << f1->GetChisquare() / f1->GetNDF() << '\n';
std::cout << "probability: " << f1->GetProb() << '\n';

diff->GetXaxis()->SetTitle("invariant mass (GeV)");
diff->GetXaxis()->SetTitleSize(0.045);
diff->GetYaxis()->SetTitle("events");
diff->GetYaxis()->SetTitleSize(0.045);
diff_pi_k->GetXaxis()->SetTitle("invariant mass (GeV)");
diff_pi_k->GetXaxis()->SetTitleSize(0.045);
```

```
diff_pi_k->GetYaxis()->SetTitle("events");
diff_pi_k->GetYaxis()->SetTitleSize(0.045);

TCanvas *c1 = new TCanvas("c1", "n gen, p module and angles", 200, 10, 600, 400);
c1->Divide(2, 2);
c1->cd(1);
gen_particles->Draw("HEP");
c1->cd(2);
p_module->Draw("HEP");
c1->cd(3);
azimuth->Draw("HEP");
c1->cd(4);
polar_angle->Draw("HEP");
c1->Print("N_P_Angles.pdf");
c1->Print("N_P_Angles.C");
c1->Print("N_P_Angles.root");

TCanvas *c2 = new TCanvas("c2", "p trans", 200, 10, 600, 400);
p_trans->Draw("HEP");
c2->Print("PTrans.pdf");
c2->Print("PTrans.C");
c2->Print("PTrans.root");

TCanvas *c3 = new TCanvas("c3", "energy and all inv mass", 200, 10, 600, 400);
c3->Divide(1, 2);
c3->cd(1);
energy->Draw("HEP");
c3->cd(2);
all_inv_mass->Draw("HEP");
c3->Print("Energy_AllInvMass.pdf");
c3->Print("Energy_AllInvMass.C");
c3->Print("Energy_AllInvMass.root");

TCanvas *c4 = new TCanvas("c4", "inv mass same and opposite (all and pi-k)", 200, 10, 600, 400);
c4->Divide(2, 2);
c4->cd(1);
same_charge_inv_mass->Draw("HEP");
c4->cd(2);
opposite_charge_inv_mass->Draw("HEP");
c4->cd(3);
pi_k_same->Draw("HEP");
c4->cd(4);
pi_k_opposite->Draw("HEP");
c4->Print("InvMassSameOpposite.pdf");
c4->Print("InvMassSameOpposite.C");
c4->Print("InvMassSameOpposite.root");

TCanvas *c5 = new TCanvas("c5", "invariant mass from decay and differences", 200, 10, 600, 400);
c5->Divide(1, 3);
```

```
 c5->cd(1);
 dec_inv_mass->Draw("HEP");
 c5->cd(2);
 diff->Draw("HEP");
 c5->cd(3);
 diff_pi_k->Draw("HEP");
 c5->Print("Decay_Diffs.pdf");
 c5->Print("Decay_Diffs.C");
 c5->Print("Decay_Diffs.root");
}
```