Relazione Progetto ROOT
Laboratorio Elettromagnetismo e Ottica

Matteo Mirabelli, Riccardo Pagnoni, Andrea Maria Varisano, Alberto Zaghini

Novembre 2023

1 Introduzione

Il programma implementato ha lo scopo di simulare ed analizzare la produzione di particelle (elementari e non) causata dalla collisione di fasci accelerati di oggetti subatomici all'interno di un esperimento di fisica delle alte energie. In particolare, si è interessati a osservare la risonanza derivante dal decadimento del mesone K^{0^*}, una particella neutra altamente instabile, ovvero il corrispondente picco nell'istogramma dei valori della massa invariante delle coppie di particelle in cui decade.

Una prima parte del programma, appoggiandosi a librerie implementate allo scopo, realizza la simulazione e ne raccoglie i risultati. Ad essa si affianca una macro da eseguire all'interno del pacchetto ROOT, tramite cui è quindi possibile analizzare e rappresentare i dati; i risultati dell'analisi sono stampati a schermo e visualizzati su *canvas* salvate in file `.pdf`, `.root` e `.C`.

2 Struttura del codice

Nel programma sono state implementate le seguenti classi, tramite opportuni file di intestazione (`.hpp`) e di implementazione (`.cpp`), inserendo le definizioni all'interno di un namespace:

- **ParticleType**: descrive le proprietà di base delle tipologie di particelle stabili simulate (nome, massa, carica).
- **ResonanceType**: descrive le proprietà di una tipologia di particella instabile che genera un fenomeno di risonanza. Per farlo eredita quelle di `ParticleType`, aggiungendo l'attributo di larghezza di risonanza.
  L'eredità implementata è di tipo `public`, in quanto `ResonanceType` è un tipo di `ParticleType` e sussiste una relazione di tipo *is-a*.
- **Particle**: descrive le proprietà cinematiche delle singole particelle simulate, ed è costruita per *aggregazione*. Tra le sue variabili private figurano un primo membro non statico del tipo user-defined *Impulse* - implementato tramite struct - che rappresenta l'impulso della particella, ed un array nativo dichiarato `static` contenente *smart pointer* a `ParticleType`. Questo permette di evitare dispendiose duplicazioni tabulando le proprietà di base per tutti i tipi di particelle e risonanze da generare in un unico oggetto in memoria, cui tutte le istanze di eventi possono accedere, e di farlo sfruttando l'ereditarietà tramite l'utilizzo dei puntatori.
  La scelta implementativa degli `unique_ptrs` per la gestione sia dell'array che degli oggetti cui fanno riferimento i suoi elementi permette di evitare il rischio di *memory leaks* legato alla gestione esplicita di risorse allocate sulla memoria dinamica.
  Opportuni indici interi permettono la gestione delle dimensioni dell'array ed il metodo pubblico `AddParticleType` l'aggiunta di nuovi tipi. Per accedere al tipo di particella delle singole istanze, si utilizza l'indice intero `fIndex` e il metodo `findParticle`.
  Sono infine presenti metodi per l'applicazione di un'opportuna trasformazione di Lorentz all'impulso (necessaria per il calcolo delle proprietà dei prodotti di decadimento), il decadimento di particelle instabili (ovvero K^{0^*}) e il calcolo la massa invariante di coppie di particelle.

Ogni classe è inoltre dotata di vari metodi *getter* per leggere le proprietà delle istanze e *setter* per modificarle.

Il programma di generazione salva su un file `.root` assieme agli opportuni istogrammi un `TArray` contenente le proporzioni di particelle generate, per la successiva analisi tramite la macro.

3 Generazione degli eventi

Vengono simulati, tramite generazione Monte Carlo, 10^5 eventi. Ciascuno produce 100 particelle il cui tipo è assegnato secondo definite proporzioni tra 7 stabiliti preventivamente. Si riportano di seguito nomi, simboli e proprietà dei tipi (massa e carica per tutti, ampiezza di risonanza per instabili) accanto alle rispettive proporzioni.

La massa e l'ampiezza di risonanza sono espresse in GeV/c^2, la carica in multipli della carica elementare e

- Pioni positivi (π^+) $m = 0.13957$, $q = +1$ 40%
- Pioni negativi (π^-) $m = 0.13957$, $q = -1$ 40%
- Kaoni positivi (K^+) $m = 0.49367$, $q = +1$ 5%
- Kaoni negativi (K^-) $m = 0.49367$, $q = -1$ 5%
- Protoni (p^+) $m = 0.93827$, $q = +1$ 4,5%
- Antiprotoni (p^-) $m = 0.93827$, $q = -1$ 4,5%
- K^{0^*} $m = 0.89166$, $q = 0$, $\Gamma_{K*} = 0.050$ 1%

Nel caso in cui venga generata una K* ne è simulato il decadimento in una coppia πK, con cariche opposte per la conservazione della carica, secondo la proporzione:

- Pione positivo - kaone negativo ($\pi^+ K^-$) 50%
- Pione negativo - kaone positivo ($\pi^- K^+$) 50%

Tali ulteriori particelle vanno ad aggiungersi alle altre 100 dell'evento.

Per quanto concerne le proprietà cinematiche, viene generato casualmente per ogni particella l'impulso. La grandezza vettoriale è determinata da tre parametri indipendenti, generati secondo le seguenti modalità:

- Angolo polare (θ): distribuzione uniforme nell'intervallo [0, π] (rad)
- Angolo azimutale(ϕ) : distribuzione uniforme nell'intervallo [0, 2π] (rad)
- Modulo dell'impulso: distribuzione esponenziale con media τ=1 (GeV/c)

La distribuzione uniforme delle coordinate angolari permette di avere una distribuzione isotropa della direzione del vettore. Dopo la generazione è effettuata un'opportuna conversione nella rappresentazione cartesiana.

4 Analisi dei dati

Proporzioni dei tipi

In fase di analisi si sono innanzitutto confrontate le abbondanze di particelle generate con i numeri attesi, entrambi riportati in Tabella 1.

Tipo di particella	Occorrenze osservate	Occorrenze attese
π^+	$(40.01 \pm 0.02) \cdot 10^5$	$40 \cdot 10^5$
π^-	$(39.99 \pm 0.02) \cdot 10^5$	$40 \cdot 10^5$
K^+	$(4.994 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
K^-	$(4.994 \pm 0.007) \cdot 10^5$	$5 \cdot 10^5$
p^+	$(4.495 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
p^-	$(4.502 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
K^{0^*}	$(1.006 \pm 0.003) \cdot 10^5$	$1 \cdot 10^5$

Tabella 1: Abbondanza delle particelle generate

Angoli e modulo dell'impulso

Si sono dunque eseguiti due fit di distribuzioni uniformi sugli istogrammi contenenti i valori generati dell'angolo azimutale e quello polare del vettore impulso e uno di distribuzione esponenziale su quello del modulo dell'impulso, al fine di verificare il corretto svolgimento della generazione secondo le distribuzioni casuali illustrate in precedenza. Nella Tabella 2 sono riportati i risultati di tali fit. Si ricorda che il valore atteso per il parametro di ciascuna distribuzione uniforme è il rapporto tra il numero totale di particelle generate (10^7) e il numero di bin del corrispondente istogramma.

Distribuzione	**Parametri del fit**	χ^2	**Gradi di libertà del fit (DOF)**	χ^2/DOF
Fit della distribuzione dell'angolo polare (θ)	55555 $\pm$ 18	175	179	0.979
Fit della distribuzione dell'angolo azimutale (ϕ)	27777 $\pm$ 9	330	359	0.919
Fit della distribuzione del modulo dell'impulso	1.0004 $\pm$ 0.0003	398	398	1.001

Tabella 2: Risultati dei fit delle distribuzioni angolari e dell'impulso

I dati riportati nelle tabelle 1 e 2 sono rappresentati graficamente in Figura 1.

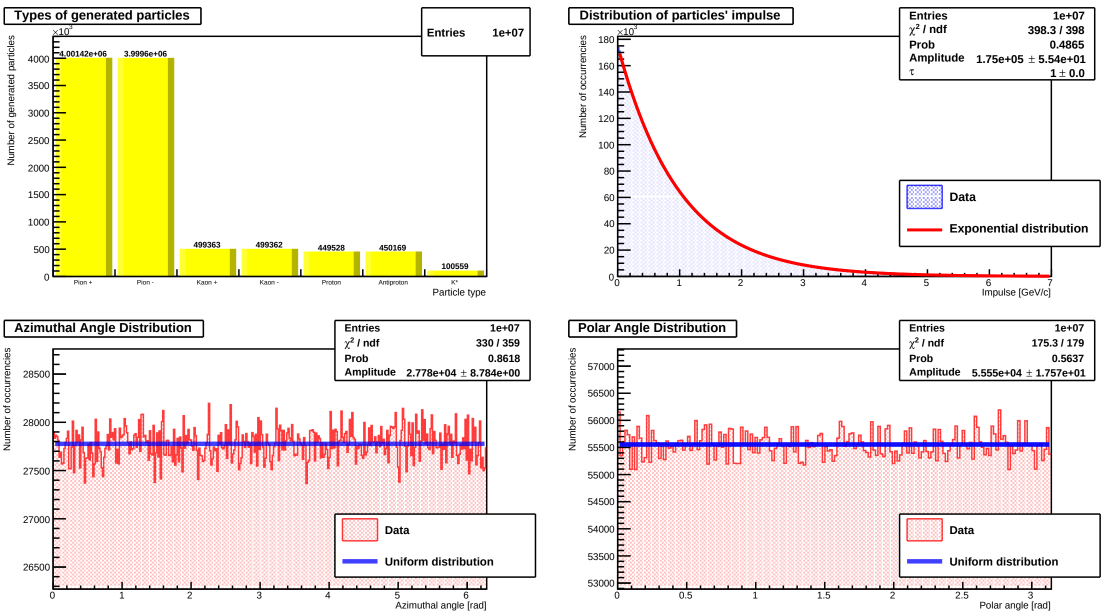

Figura 1: In figura sono rappresentate, in senso orario, rispettivamente l'istogramma delle abbondanze delle particelle, della distribuzione del modulo dell'impulso, dell'angolo polare e dell'angolo azimutale

Estrazione e studio del picco di risonanza

Per ricostruire la risonanza si è seguito il seguente procedimento.
Si è riempito un istogramma con le masse invarianti di tutte le coppie possibili di particelle con carica concorde generate dagli eventi. Queste combinazioni sono accidentali e non possono venire dal decadimento della K*, in quanto questa produce solamente particelle di carica discorde. Si è riempito in seguito un istogramma contenente, invece, le masse invarianti di tutte le coppie possibili di particelle di carica discorde. Queste combinazioni sono sia accidentali che dovute alle particelle decadute dalla K*. La differenza tra il secondo e il primo istogramma restituisce la distribuzione della massa invariante dei prodotti del decadimento.

Successivamente si è ripetuta un'analoga procedura di sottrazione tra istogrammi di massa invariante considerando solamente i pioni e kaoni, ovvero le tipologie di particelle generate dal decadimento di K*, che ha permesso

di ottenere una distribuzione della massa invariante più accurata grazie alla riduzione del fondo da eliminare.
Di entrambi gli istogrammi ottenuti si è eseguito un fit tramite una distribuzione gaussiana, la cui media corrisponde alla stima della massa invariante della K* e la deviazione stardard della sua larghezza di risonanza.

Le distribuzioni ottenute sono state quindi confrontate con un fit gaussiano eseguito sull'istogramma di benchmark, creato prima della fase di simulazione e durante questa riempito con i valori di massa invariante delle sole coppie di prodotti del decadimento delle risonanze K*.
Nella Tabella 3 sono riportati i risultati dei fit sopra indicati.

Distribuzione e fit	**Media del fit (μ)**	**Deviazione standard (σ)**	**Ampiezza**	χ^2/DOF
Massa invariante delle K* generate (fit gaussiano)	0.89176 ± 0.00016	0.04998 ± 0.00011	8020 ± 30	1.229
Massa invariante ottenuta dalla differenza delle combinazioni di tutte le particelle di carica discorde e carica concorde (fit gaussiano)	0.891 ± 0.005	0.048 ± 0.005	7700 ± 700	0.992
Massa invariante ottenuta dalla differenza delle combinazioni πK di carica discorde e carica concorde (fit gaussiano)	0.893 ± 0.003	0.049 ± 0.003	8200 ± 400	1.066

Tabella 3: Risultati dei fit delle distribuzioni delle masse invarianti

I dati riportati nella Tabella 3 sono rappresentati graficamente in Figura 2.

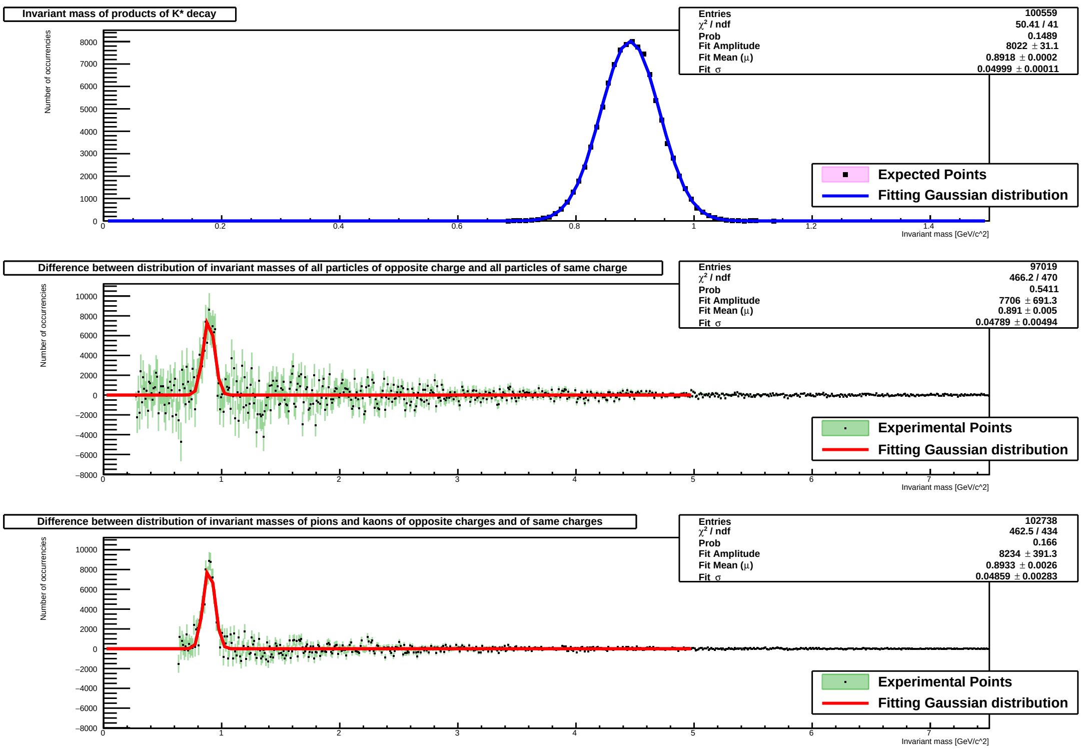

Figura 2: In figura sono rappresentati, dall'alto verso il basso, gli istogrammi delle distribuzioni della massa invariante dei prodotti della sola K*, della differenza delle combinazioni di tutte le particelle di carica discorde e carica concorde e della differenza delle combinazioni πK di carica discorde e carica concorde

5 Listato del codice

- particleType.hpp

```cpp
#ifndef PARTICLETYPE_HPP
#define PARTICLETYPE_HPP

#include <iomanip>
#include <iostream>
#include <stdexcept>
#include <string>

namespace pt {

class ParticleType {
 public:
  // constructors
  ParticleType(std::string const&, double, int);
  ParticleType();
  virtual ~ParticleType() = default;

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

  // Print member function
  virtual void Print() const;

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

}  // namespace pt

#endif
```

- particleType.cxx

```cpp
#include "particleType.hpp"

namespace pt {

// parametric constructor
ParticleType::ParticleType(std::string const& name, double mass, int charge)
    : fName(name), fMass(mass), fCharge(charge) {
  if (mass < 0 || charge < -1 || charge > 1)
    throw std::runtime_error("Invalid input");
}

// default constructor; explicitly defined because of const data members
ParticleType::ParticleType() : fName(""), fMass(0.), fCharge(0) {}

// returns the name of the particle type
std::string const& ParticleType::GetName() const { return fName; }

// returns the mass of the particle type
double ParticleType::GetMass() const { return fMass; }

// returns the charge of the particle type
int ParticleType::GetCharge() const { return fCharge; }

// returns the resonance width (0 for stable particles)
double ParticleType::GetWidth() const { return 0.; }

// prints the particle type data
void ParticleType::Print() const {
  std::cout << "\n PARTICLE TYPE DATA \n\n -------------- \n\n"
            << " Name: " << std::setw(8) << fName << "\n Mass: " << std::setw(8)
            << fMass << "\n Charge: " << std::setw(6) << fCharge << '\n';
```

```
}

}  // namespace pt
```

- resonanceType.hpp

```
#ifndef RESONANCETYPE_HPP
#define RESONANCETYPE_HPP

#include "particleType.hpp"

namespace pt {

class ResonanceType : public ParticleType {
 public:
  // constructors
  ResonanceType(std::string const&, double, int, double);
  ResonanceType();

  // getters
  double GetWidth() const;

  // Print member function
  void Print() const;

 private:
  double const fWidth;
};

}  // namespace pt

#endif
```

- resonanceType.cxx

```
#include "resonanceType.hpp"

namespace pt {

// parametric constructor
ResonanceType::ResonanceType(std::string const& name, double mass, int charge,
                             double width)
    : ParticleType(name, mass, charge), fWidth(width) {}

// default constructor; explicitly defined because of const data members
ResonanceType::ResonanceType() : ParticleType(), fWidth(0.) {}

// returns the resonance width
double ResonanceType::GetWidth() const { return fWidth; }

// prints the resonance type data
void ResonanceType::Print() const {
  ParticleType::Print();
  std::cout << " Width: " << std::setw(7) << fWidth << '\n';
}

}  // namespace pt
```

- particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include <cmath>
#include <memory>

#include "resonanceType.hpp"

// struct for the 3D impulse vector
struct Impulse {
```

```cpp
  // components
  double fPx{0};
  double fPy{0};
  double fPz{0};

  // Print method
  void Print() const;

  // method to compute squared norm
  double SquaredNorm() const;
};

// sum operator overload
Impulse operator+(const Impulse& p1, const Impulse& p2);

namespace pt {

class Particle {
 public:
  // constructors
  Particle(std::string const& name, Impulse P);
  Particle() = default;

  // getters
  int GetIndex() const;

  double GetPx() const;
  double GetPy() const;
  double GetPz() const;

  double GetCharge() const;

  double GetMass() const;
  double GetEnergy() const;

  // setters

  void SetIndex(int index);
  void SetIndex(std::string const& name);

  void SetP(double px, double py, double pz);
  void SetP(Impulse const& p);

  // add new particle / resonance type
  static void AddParticleType(std::string const& name, double mass, int charge,
                              double width = 0.);

  // compute invariant mass of two particles
  double InvMass(Particle const& p) const;

  // simulate decay of a particle into two daughters
  int Decay2body(Particle& dau1, Particle& dau2) const;

  // print methods
  static void PrintParticleTypes();
  void Print() const;

 private:
  static const int fMaxNumParticleType;
  static std::unique_ptr<std::unique_ptr<ParticleType>[]> fParticleTypes;
  static int fNParticleType;
  int fIndex;
  Impulse fP;

  // find particle type index
  static int FindParticle(std::string const& name);

  // apply Lorentz transformation
  void Boost(double bx, double by, double bz);
};
}  // namespace pt

#endif
```

- particle.cxx

```cpp
#include "particle.hpp"

void Impulse::Print() const {
  std::cout << " Impulse:  (" << fPx << ", " << fPy << ", " << fPz << " ) \n";
}

double Impulse::SquaredNorm() const {
  return fPx * fPx + fPy * fPy + fPz * fPz;
}

Impulse operator+(const Impulse& p1, const Impulse& p2) {
  Impulse pTot;

  pTot.fPx = p1.fPx + p2.fPx;
  pTot.fPy = p1.fPy + p2.fPy;
  pTot.fPz = p1.fPz + p2.fPz;

  return pTot;
}

namespace pt {

// --------------- PUBLIC METHODS ---------------

// parametric constructor
Particle::Particle(std::string const& name, Impulse P)
    : fIndex(FindParticle(name)), fP(P) {
  if (fIndex == fNParticleType)
    throw std::runtime_error{" Particle not found \n\n"};
}

// getters
int Particle::GetIndex() const { return fIndex; }

double Particle::GetPx() const { return fP.fPx; }

double Particle::GetPy() const { return fP.fPy; }

double Particle::GetPz() const { return fP.fPz; }

double Particle::GetCharge() const {
  return fParticleTypes[fIndex]->GetCharge();
}

double Particle::GetMass() const { return fParticleTypes[fIndex]->GetMass(); }

double Particle::GetEnergy() const {
  return sqrt(GetMass() * GetMass() + fP.SquaredNorm());
}

// setters
void Particle::SetIndex(int index) {
  if (index >= 0 && index < fNParticleType) fIndex = index;
}

void Particle::SetIndex(std::string const& name) {
  if (FindParticle(name) >= 0 && FindParticle(name) < fNParticleType)
    fIndex = FindParticle(name);
}

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

void Particle::SetP(Impulse const& p) { fP = p; }

// add new particle / resonance type
void Particle::AddParticleType(std::string const& name, double mass, int charge,
                               double width) {
  // check if particle already exists, proceed if not
  // and array size limit not reached
  if (FindParticle(name) == fNParticleType &&
      FindParticle(name) < fMaxNumParticleType) {
```

```cpp
    // add particle or resonance type, depending on width value
    if (width == 0.) {
      // pass the ownership of new ParticleType object
      fParticleTypes[FindParticle(name)] = std::move(std::unique_ptr<ParticleType>(
        new ParticleType(name, mass, charge)));
    } else {
      // pass the ownership of new ResonanceType object
      fParticleTypes[FindParticle(name)] = std::move(std::unique_ptr<ParticleType>(
        new ResonanceType(name, mass, charge, width)));
    }
    // increment number of current particle types
    ++fNParticleType;
  } else if (FindParticle(name) == fMaxNumParticleType) {
    throw std::runtime_error{" Array size limit reached \n\n"};
  } else {
    std::cout << " Particle already exists \n\n";
  }
};

// compute invariant mass of two particles
double Particle::InvMass(Particle const& other) const {
  double TotEnergy = GetEnergy() + other.GetEnergy();
  Impulse TotImpulse = fP + other.fP;

  // apply SR formula
  return std::sqrt((TotEnergy * TotEnergy) - TotImpulse.SquaredNorm());
}

// simulate decay of a particle into two daughters
int Particle::Decay2body(Particle& dau1, Particle& dau2) const {
  if (GetMass() == 0.0) {
    std::cout << "Decayment cannot be preformed if mass is zero\n";
    return 1;
  }

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

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

    // gaussian random numbers

    float x1, x2, w, y1;

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

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

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

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

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

  double norm = 2 * M_PI / RAND_MAX;

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

```
  dau2.SetP(-pout * sin(theta) * cos(phi), -pout * sin(theta) * sin(phi),
            -pout * cos(theta));

  double energy = sqrt(fP.SquaredNorm() + massMot * massMot);

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

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

  return 0;
}


// print methods
void Particle::PrintParticleTypes() {
  for (int i = 0; i < fNParticleType; ++i) {
    fParticleTypes[i]->Print();
  }
  std::cout << '\n';
}

void Particle::Print() const {
  std::cout << "\n PARTICLE DATA \n\n -------------- \n\n"
            << " Index: " << std::setw(8) << fIndex
            << "\n Name: " << std::setw(8) << fParticleTypes[fIndex]->GetName()
            << "\n\n -------------- \n";
  fP.Print();
}

// --------------- STATIC MEMBERS ---------------

const int Particle::fMaxNumParticleType = 10;
int Particle::fNParticleType = 0;

std::unique_ptr<std::unique_ptr<ParticleType>[]> Particle::fParticleTypes =
  std::make_unique<std::unique_ptr<ParticleType>[]>(Particle::fMaxNumParticleType);

// --------------- PRIVATE METHODS ---------------

// find particle type index
int Particle::FindParticle(std::string const& name) {
  int i{0};

  for (; i < fNParticleType; ++i) {
    if (fParticleTypes[i]->GetName() == name) {
      break;
    }
  }

  if (i != fNParticleType) {
    return i;
  } else {
    // std::cout << "Particle not found \n";
    return fNParticleType;
  }
}

// apply Lorentz transformation
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 * fP.fPx + by * fP.fPy + bz * fP.fPz;
  double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

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

}  // namespace pt
```

- main_gen.C

```
#include <array>

#include "TApplication.h"
#include "TArray.h"
#include "TBenchmark.h"
#include "TCanvas.h"
#include "TFile.h"
#include "TH1I.h"
#include "TH2D.h"
#include "TList.h"
#include "TMath.h"
#include "TROOT.h"
#include "TRandom.h"
#include "particle.hpp"

void main_gen() {
  // Automatically load libraries
  R__LOAD_LIBRARY( particleType_cxx.so )
  R__LOAD_LIBRARY( resonanceType_cxx.so )
  R__LOAD_LIBRARY( particle_cxx.so )

  std::cout << " ---- K* decay simulation ----\n\n";

  // Declare and initialize number of types and number of simulated events
  const Int_t n_types = 7;
  const Double_t n_events = 1E5;

  // Add Particle types with respective mass and charge

  pt::Particle::AddParticleType("pion+", 0.13957, 1);
  pt::Particle::AddParticleType("pion-", 0.13957, -1);
  pt::Particle::AddParticleType("kaon+", 0.49367, 1);
  pt::Particle::AddParticleType("kaon-", 0.49367, -1);
  pt::Particle::AddParticleType("proton+", 0.93827, 1);
  pt::Particle::AddParticleType("proton-", 0.93827, -1);
  pt::Particle::AddParticleType("k*", 0.89166, 0, 0.05);

  // ROOT double Array to store proportions of generated particles
  // and number of events

  TArrayD* prop_arr = new TArrayD(n_types + 1);

  // store number of events in last position
  prop_arr->SetAt(n_events, n_types);

  // store proportions of generated particles
  prop_arr->SetAt(0.4, 0);
  prop_arr->SetAt(0.4, 1);
  prop_arr->SetAt(0.05, 2);
  prop_arr->SetAt(0.05, 3);
  prop_arr->SetAt(0.045, 4);
  prop_arr->SetAt(0.045, 5);
  prop_arr->SetAt(0.01, 6);

  // Confirmation message
  std::cout << "Particle Types generated\n";

  // Set seed of random number generator
  gRandom->SetSeed();

  // Initialize array of particles to use for each event
  std::array<pt::Particle, 130> EventParticles;

  // --------------- HISTOGRAM GENERATION ---------------

  // Histogram containing the proportions of generated particle types
  TH1I* type_histo =
      new TH1I("type", "Types of generated particles", n_types, 0, n_types);

  // Histogram containing
  TH1D* phi_histo =
      new TH1D("phi", "Azimutal angle distribution", 360, 0, 2 * TMath::Pi());
  TH1D* theta_histo =
```

```
      new TH1D("theta", "Polar angle distribution", 180, 0, TMath::Pi());

  // Histogram containing particles' impulse modulus
  TH1D* impulse_histo = new TH1D("impulse", "Modulus of impulse", 400, 0, 7.);

  // Histogram containing tranverse impulse of particles
  TH1D* transv_impulse_histo = new TH1D("t_impulse", "Tranverse impulse", 400, 0, 7.);

  // Histogram containing energy of particles
  TH1D* energy_histo = new TH1D("energy", "Energy of generated particles", 400, 0, 7.);

  // --- Method 'Sumw2()' is called on following histograms for future error
  // calculations ---

  // Histogram containing invariant mass of all particles of opposite sign
  // charges
  TH1D* inv_mass_disc_histo =
      new TH1D("inv_mass_disc",
               "Invariant mass of oppositely charged particles", 750, 0, 7.5);
  inv_mass_disc_histo->Sumw2();

  // Histogram containing invariant mass of all particles of same sign charges
  TH1D* inv_mass_conc_histo =
      new TH1D("inv_mass_conc",
               "Invariant mass of identically charged particles", 750, 0, 7.5);
  inv_mass_conc_histo->Sumw2();

  // Histogram containing invariant mass of opposite charge pions and kaons
  TH1D* inv_mass_pk0_histo =
      new TH1D("inv_mass_pk0", "Invariant mass of pi+k- / pi-k+", 750, 0, 7.5);
  inv_mass_pk0_histo->Sumw2();

  // Histogram containing invariant mass of same charge pions and kaons
  TH1D* inv_mass_pk1_histo =
      new TH1D("inv_mass_pk1", "Invariant mass of pi+k+ / pi-k-", 750, 0, 7.5);
  inv_mass_pk1_histo->Sumw2();

  // Histogram containing invariant mass of products of k* decays
  TH1D* inv_mass_kstar_histo =
      new TH1D("inv_mass_kstar", "Invariant mass of products of K* decay", 150, 0, 1.5);
  inv_mass_kstar_histo->Sumw2();

  // Confirmation message
  std::cout << "Histogram generated\n\n";

  // Variables to store generated values

  Double_t phi;    // azimutal angle
  Double_t theta;  // polar angle
  Double_t p_mod;  // impulse modulus
  Impulse p_gen;   // impulse vector

  Double_t rndm_idx;  // random index to assign particle type

  Int_t decay_idx = 0;
  // index to store decay products past the 100th position of the array

  int decay_outcome = 0;

  // ROOT Benchmark to evaluate code performance
  TBenchmark time;

  // Notification message
  std::cout << "Event generation and histogram filling begun...\n";

  // Start benchmark
  time.Start("Event generation and histogram filling");

  // Particle generation and histogram filling

  for (Int_t i = 0; i < n_events; ++i) {
    // Each event
    decay_idx = 0;
```

```cpp

    for (Int_t j = 0; j < 100; ++j) {
      // Generate angles according to uniform dist
      phi = gRandom->Uniform(0, 2 * TMath::Pi());
      theta = gRandom->Uniform(0, TMath::Pi());

      // Generate impulse modulus according to exponential dist
      p_mod = gRandom->Exp(1);

      // Compute impluse vector components
      p_gen.fPx = p_mod * TMath::Sin(theta) * TMath::Cos(phi);
      p_gen.fPy = p_mod * TMath::Sin(theta) * TMath::Sin(phi);
      p_gen.fPz = p_mod * TMath::Cos(theta);

      // Fill angles, impulse and transverse impulse histos
      phi_histo->Fill(phi);
      theta_histo->Fill(theta);
      impulse_histo->Fill(p_mod);
      transv_impulse_histo->Fill(
          TMath::Sqrt(p_gen.fPx * p_gen.fPx + p_gen.fPy * p_gen.fPy));

      // Assign randomly particle type according to proportions

      rndm_idx = gRandom->Rndm();

      if (rndm_idx < prop_arr->GetAt(0)) {
        EventParticles[j].SetIndex("pion+");
        EventParticles[j].SetP(p_gen);
      } else if (rndm_idx < prop_arr->GetAt(0) + prop_arr->GetAt(1)) {
        EventParticles[j].SetIndex("pion-");
        EventParticles[j].SetP(p_gen);
      } else if (rndm_idx <
                 prop_arr->GetAt(0) + prop_arr->GetAt(1) + prop_arr->GetAt(2)) {
        EventParticles[j].SetIndex("kaon+");
        EventParticles[j].SetP(p_gen);
      } else if (rndm_idx < prop_arr->GetAt(0) + prop_arr->GetAt(1) +
                                prop_arr->GetAt(2) + prop_arr->GetAt(3)) {
        EventParticles[j].SetIndex("kaon-");
        EventParticles[j].SetP(p_gen);
      } else if (rndm_idx < prop_arr->GetAt(0) + prop_arr->GetAt(1) +
                                prop_arr->GetAt(2) + prop_arr->GetAt(3) +
                                prop_arr->GetAt(4)) {
        EventParticles[j].SetIndex("proton+");
        EventParticles[j].SetP(p_gen);
      } else if (rndm_idx < prop_arr->GetAt(0) + prop_arr->GetAt(1) +
                                prop_arr->GetAt(2) + prop_arr->GetAt(3) +
                                prop_arr->GetAt(4) + prop_arr->GetAt(5)) {
        EventParticles[j].SetIndex("proton-");
        EventParticles[j].SetP(p_gen);
      } else {
        EventParticles[j].SetIndex("k*");
        EventParticles[j].SetP(p_gen);

        // Generate decay products of K* and store them in the array
        rndm_idx = gRandom->Rndm();

        if (rndm_idx < 0.5) {
          EventParticles[100 + decay_idx].SetIndex("pion+");
          EventParticles[101 + decay_idx].SetIndex("kaon-");
          decay_outcome = EventParticles[j].Decay2body(
              EventParticles[100 + decay_idx], EventParticles[101 + decay_idx]);
        } else {
          EventParticles[100 + decay_idx].SetIndex("pion-");
          EventParticles[101 + decay_idx].SetIndex("kaon+");
          decay_outcome = EventParticles[j].Decay2body(
              EventParticles[100 + decay_idx], EventParticles[101 + decay_idx]);
        }

        // if decay is carried out correctly
        if (decay_outcome == 0) {
          // fill invariant mass of decay products histo
          inv_mass_kstar_histo->Fill(EventParticles[100 + decay_idx].InvMass(
              EventParticles[101 + decay_idx]));

          // move decay index forward
          decay_idx += 2;
```

```
      }
    }

    // fill type, energy histos
    type_histo->Fill(EventParticles[j].GetIndex());
    energy_histo->Fill(EventParticles[j].GetEnergy());
  }

  // fill invariant mass histos for opposite / same charge

  for (Int_t a = 0; a < 100 + decay_idx; ++a) {
    for (Int_t b = a + 1; b < 100 + decay_idx; ++b) {
      if (EventParticles[a].GetCharge() * EventParticles[b].GetCharge() < 0) {
        // Fill histo of pions and kaons with opposite charges
        if ((EventParticles[a].GetIndex() == 0 &&
             EventParticles[b].GetIndex() == 3) ||
            (EventParticles[a].GetIndex() == 3 &&
             EventParticles[b].GetIndex() == 0) ||
            (EventParticles[a].GetIndex() == 1 &&
             EventParticles[b].GetIndex() == 2) ||
            (EventParticles[a].GetIndex() == 2 &&
             EventParticles[b].GetIndex() == 1)) {
          inv_mass_pk0_histo->Fill(EventParticles[a].InvMass(EventParticles[b]));
        }

        // Fill histo of invariant mass of opposite sign charges
        inv_mass_disc_histo->Fill(EventParticles[a].InvMass(EventParticles[b]));

      } else if (EventParticles[a].GetCharge() *
                     EventParticles[b].GetCharge() >
                 0) {
        if ((EventParticles[a].GetIndex() == 0 &&
             EventParticles[b].GetIndex() == 2) ||
            (EventParticles[a].GetIndex() == 2 &&
             EventParticles[b].GetIndex() == 0) ||
            (EventParticles[a].GetIndex() == 1 &&
             EventParticles[b].GetIndex() == 3) ||
            (EventParticles[a].GetIndex() == 3 &&
             EventParticles[b].GetIndex() == 1)) {
          // Fill histo of pions and kaons with same charges
          inv_mass_pk1_histo->Fill(EventParticles[a].InvMass(EventParticles[b]));
        }

        // Fill histo of invariant mass of same sign charges
        inv_mass_conc_histo->Fill(EventParticles[a].InvMass(EventParticles[b]));
      }
    }
  }

  // Reset array of particles
  EventParticles.fill(pt::Particle());
}

// Stop benchmark
time.Stop("Event generation and histogram filling");
// Confirmation message
std::cout << "Event generation and histogram filling ended\n";

// Print benchmark results
time.Show("Event generation and histogram filling");
std::cout << '\n';

// Store histos in ROOT List
TList* list = new TList();
list->Add(type_histo);
list->Add(phi_histo);
list->Add(theta_histo);
list->Add(impulse_histo);
list->Add(transv_impulse_histo);
list->Add(energy_histo);
list->Add(inv_mass_disc_histo);
list->Add(inv_mass_conc_histo);
list->Add(inv_mass_pk0_histo);
list->Add(inv_mass_pk1_histo);
list->Add(inv_mass_kstar_histo);

```

```
  // Canvas definition

  TCanvas* canva1 = new TCanvas("canva1", "Types, Angles, Energy and Impulse",
                                200, 10, 1400, 900);
  canva1->Divide(2, 3);

  TCanvas* canva2 =
      new TCanvas("canva2", "Invariant Masses", 200, 10, 1400, 900);
  canva2->Divide(2, 3);

  // Confirmation message
  std::cout << "Canvas created" << '\n';

  // Draw first five histograms (Types, Angles, Energy and Impulse)

  canva1->cd();
  for (Int_t canva_idx = 1; canva_idx <= 6; ++canva_idx) {
    canva1->cd(canva_idx);
    list->At(canva_idx - 1)->Draw();
  }

  // Draw last five histograms (Invariant Masses)
  canva2->cd();
  for (Int_t canva_idx = 1; canva_idx <= 5; ++canva_idx) {
    canva2->cd(canva_idx);
    list->At(canva_idx + 5)->Draw();
  }

  // Confirmation message
  std::cout << "Histos drawn \n";

  // Create ROOT file and write List and Array of proportions

  TFile* file = new TFile("histos.root", "RECREATE");

  file->WriteObject(prop_arr, "prop_arr");
  file->WriteObject(list, "list");

  file->Close();

  // Confirmation message
  std::cout << "Histos written to file '' histos.root '' " << '\n'
            << "Analyse in Root with analyze_histo.C macro\n";

  return;
}
```

- analyze_histos.C

```
#include <iostream>

#include "TCanvas.h"
#include "TF1.h"
#include "TFile.h"
#include "TH2D.h"
#include "TLegend.h"
#include "TList.h"
#include "TMath.h"
#include "TROOT.h"
#include "TStyle.h"

// function for cosmetics
void setStyle() {
  gROOT->SetStyle("Plain");
  gStyle->SetOptStat(1110);
  gStyle->SetOptFit(1111);
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(1);
}

// function for cleaning up objects
void cleanUp() {
  delete gROOT->FindObject("canva_retrieved");
  delete gROOT->FindObject("canva_k_star");
```

```
  delete gROOT->FindObject("type");
  delete gROOT->FindObject("phi");
  delete gROOT->FindObject("theta");
  delete gROOT->FindObject("impulse");
  delete gROOT->FindObject("inv_mass_disc");
  delete gROOT->FindObject("inv_mass_conc");
  delete gROOT->FindObject("inv_mass_pk0");
  delete gROOT->FindObject("inv_mass_pk1");
  delete gROOT->FindObject("inv_mass_kstar");
  delete gROOT->FindObject("list");
  delete gROOT->FindObject("prop_arr");
  delete gROOT->FindObject("unif");
  delete gROOT->FindObject("gauss");
  delete gROOT->FindObject("exp");
}

// function to analize particle types distribution
void analyze_particle_type(TH1I* histo, TArrayD* prop_array) {
  std::cout << "ANALYZING PARTYCLE TYPES DISTRIBUTION \n\n";
  Int_t n_types = prop_array->GetSize() - 1;
  Double_t n_generations = 100 * prop_array->At(n_types);

  for (Int_t i = 0; i < n_types; ++i) {
    std::cout << "Particle: " << histo->GetXaxis()->GetBinLabel(i + 1) << '\n';
    std::cout << "Entries: " << histo->GetBinContent(i + 1) << " +- "
              << histo->GetBinError(i + 1) << '\n';
    std::cout << "Expected: " << prop_array->At(i) * n_generations << '\n';
    ((histo->GetBinContent(i + 1) - histo->GetBinError(i + 1)) <
         prop_array->At(i) * n_generations &&
     prop_array->At(i) * n_generations <
         (histo->GetBinContent(i + 1) + histo->GetBinError(i + 1)))
        ? std::cout << "Expectation confirmed\n\n"
        : std::cout << "Something went wrong. Visual check suggested\n\n";
  }

  std::cout << "\n--------------------- \n\n";
}

// macro body
void analyze_histos() {
  setStyle();
  cleanUp();

  // Retrieve list from file
  TFile* histos_file = new TFile("histos.root", "read");
  TList* histos_list = histos_file->Get<TList>("list");

  // Retrieve types histogram (from list)
  TH1I* type = (TH1I*)histos_list->FindObject("type");
  // Retrieve proportions array (from file)
  TArrayD* proportions_array = histos_file->Get<TArrayD>("prop_arr");

  // Create canva for retrieved histograms (Types, Impulse and Angles)
  TCanvas* canva_retrieved = new TCanvas(
      "canva_retrieved", "Types proportions, Impulse and Angles distributions",
      200, 10, 1400, 800);
  canva_retrieved->Divide(2, 2);

  canva_retrieved->cd(1);

  // Types histogram labelling and drawing

  type->GetYaxis()->SetTitle("Number of generated particles");
  type->GetXaxis()->SetTitle("Particle type");
  type->GetXaxis()->SetBinLabel(1, "Pion +");
  type->GetXaxis()->SetBinLabel(2, "Pion -");
  type->GetXaxis()->SetBinLabel(3, "Kaon +");
  type->GetXaxis()->SetBinLabel(4, "Kaon -");
  type->GetXaxis()->SetBinLabel(5, "Proton");
  type->GetXaxis()->SetBinLabel(6, "Antiproton");
  type->GetXaxis()->SetBinLabel(7, "K*");
  type->SetMinimum(0);
  type->SetMaximum(1.1 * type->GetMaximum());

  // Cosmetics and display settings
  gStyle->SetOptStat(10);
```

```
type->SetFillColor(kYellow);
type->SetBarWidth(0.9);
type->SetBarOffset(0.05);
type->SetMarkerSize(1.5);

type->Draw("bar1,text0");

// Call function to verify compliance with expected proportions
analyze_particle_type(type, proportions_array);

canva_retrieved->cd(3);

// Uniform distribution function for fit
TF1* uniform_dist = new TF1("unif", "[0]", 0, 2 * TMath::Pi());
uniform_dist->SetParName(0, "Amplitude");
uniform_dist->SetLineColorAlpha(kBlue, 0.75);
uniform_dist->SetLineWidth(3);

// Retrieve and label azimuthal and polar angles histograms
// fit uniform distribution to each and draw everything

TH1D* phi_histo = (TH1D*)histos_list->FindObject("phi");

phi_histo->SetTitle("Azimuthal Angle Distribution");
phi_histo->GetXaxis()->SetTitle("Azimuthal angle [rad]");
phi_histo->GetYaxis()->SetTitle("Number of occurrencies");
phi_histo->SetMinimum(0.96 * phi_histo->GetMinimum());
phi_histo->SetMaximum(1.02 * phi_histo->GetMaximum());
phi_histo->SetLineColorAlpha(kRed, 0.75);
phi_histo->SetFillColorAlpha(kRed, 0.25);
phi_histo->SetFillStyle(3001);

phi_histo->Fit(uniform_dist, "q", "", 0, 2 * TMath::Pi());

std::cout << "\n AZIMUTHAL ANGLE PHI -- FIT FUNCTION\n";
std::cout << "\n Parameter (Amplitude): " << uniform_dist->GetParameter(0)
          << " +- " << uniform_dist->GetParError(0);
std::cout << "\n Chi square: " << uniform_dist->GetChisquare();
std::cout << "\n NDF: " << uniform_dist->GetNDF();
std::cout << "\n Chi/NDF: "
          << uniform_dist->GetChisquare() / uniform_dist->GetNDF();
std::cout << "\n Probability: " << uniform_dist->GetProb() << '\n';
std::cout << "\n--------------------- \n\n";

// Add legend
TLegend* legend_phi = new TLegend(0.62, 0.14, 0.99, 0.35);
legend_phi->AddEntry(phi_histo, "Data", "f");
legend_phi->AddEntry(uniform_dist, "Uniform distribution", "l");

phi_histo->Draw();
uniform_dist->Draw("SAME");
legend_phi->Draw("SAME");

// adjust fitting function range
uniform_dist->SetRange(0., TMath::Pi());

canva_retrieved->cd(4);

TH1D* theta_histo = (TH1D*)histos_list->FindObject("theta");

theta_histo->SetTitle("Polar Angle Distribution");
theta_histo->GetXaxis()->SetTitle("Polar angle [rad]");
theta_histo->GetYaxis()->SetTitle("Number of occurrencies");
theta_histo->SetMinimum(0.96 * theta_histo->GetMinimum());
theta_histo->SetMaximum(1.02 * theta_histo->GetMaximum());
theta_histo->SetLineColorAlpha(kRed, 0.75);
theta_histo->SetFillColorAlpha(kRed, 0.25);
theta_histo->SetFillStyle(3001);

theta_histo->Fit(uniform_dist, "q", "", 0, TMath::Pi());

std::cout << "\n POLAR ANGLE THETA -- FIT FUNCTION\n";
std::cout << "\n Parameter (Amplitude): " << uniform_dist->GetParameter(0)
          << " +- " << uniform_dist->GetParError(0);
std::cout << "\n Chi square: " << uniform_dist->GetChisquare();
std::cout << "\n NDF: " << uniform_dist->GetNDF();
```

```cpp
std::cout << "\n Chi/NDF: "
          << uniform_dist->GetChisquare() / uniform_dist->GetNDF();
std::cout << "\n Probability: " << uniform_dist->GetProb() << '\n';
std::cout << "\n--------------------- \n\n";

// Add legend
TLegend* legend_theta = new TLegend(0.62, 0.14, 0.99, 0.35);
legend_theta->AddEntry(theta_histo, "Data", "f");
legend_theta->AddEntry(uniform_dist, "Uniform distribution", "l");

theta_histo->Draw();
uniform_dist->Draw("SAME");
legend_theta->Draw("SAME");

// Retrieve and label impulse histogram,
// fit exponential distribution and draw everything

canva_retrieved->cd(2);

TH1D* impulse_histo = (TH1D*)histos_list->FindObject("impulse");

impulse_histo->SetTitle("Distribution of particles' impulse");
impulse_histo->GetXaxis()->SetTitle("Impulse [GeV/c]");
impulse_histo->GetYaxis()->SetTitle("Number of occurrencies");
impulse_histo->SetLineColorAlpha(kBlue, 0.75);
impulse_histo->SetFillColorAlpha(kBlue, 0.35);
impulse_histo->SetFillStyle(3001);

TF1* exp_dist = new TF1("exp", "([0]/[1])*exp(-x/[1])", 0, 7.);
exp_dist->SetParameters(1000, 1);
exp_dist->SetParName(0, "Amplitude");
exp_dist->SetParName(1, "#tau");

impulse_histo->Fit("exp", "q", "", 0, 7.);

TF1* fit_func = impulse_histo->GetFunction("exp");
fit_func->SetLineColor(kRed);
fit_func->SetLineWidth(2);

std::cout << "\n IMPULSE -- FIT FUNCTION\n";
std::cout << "\n Parameter 0 (Amplitude): " << fit_func->GetParameter(0)
          << " +- " << fit_func->GetParError(0);
std::cout << "\n Parameter 1 (average): " << fit_func->GetParameter(1)
          << " +- " << fit_func->GetParError(1);
std::cout << "\n Chi square: " << fit_func->GetChisquare();
std::cout << "\n NDF: " << fit_func->GetNDF();
std::cout << "\n Chi/NDF: " << fit_func->GetChisquare() / fit_func->GetNDF();
std::cout << "\n Probability: " << fit_func->GetProb() << '\n';

// Check if average is compatible with 1 GeV/c
(fit_func->GetParameter(1) - fit_func->GetParError(1) < 1. &&
 1. < fit_func->GetParameter(1) + fit_func->GetParError(1))
    ? std::cout << "\nExpectation confirmed \n"
    : std::cout << "\nSomething went wrong: incompatible average value. "
                   "Visual check suggested \n";
std::cout << "\n--------------------- \n\n";

// Add legend
TLegend* legend_impulse = new TLegend(0.62, 0.20, 0.99, 0.42);
legend_impulse->AddEntry(impulse_histo, "Data", "f");
legend_impulse->AddEntry(fit_func, "Exponential distribution", "l");

impulse_histo->Draw();
fit_func->Draw("SAME");
legend_impulse->Draw("SAME");

// -------- K* RESONANCE ANALYSIS --------

// Retrieve invariant masses histograms

TCanvas* canva_k_star =
    new TCanvas("canva_k_star", "K* Resonance Analysis", 200, 10, 1300, 900);
canva_k_star->Divide(1, 3);

TH1D* disc_histo = (TH1D*)histos_list->FindObject("inv_mass_disc");
TH1D* conc_histo = (TH1D*)histos_list->FindObject("inv_mass_conc");
```

```
TH1D* disc_pk_histo = (TH1D*)histos_list->FindObject("inv_mass_pk0");
TH1D* conc_pk_histo = (TH1D*)histos_list->FindObject("inv_mass_pk1");
TH1D* kstar_histo = (TH1D*)histos_list->FindObject("inv_mass_kstar");

// Close read file
histos_file->Close();

canva_k_star->cd(2);

// Difference histograms between invariant masses of all particles of opposite
// charge and all particles of same charge

TH1D* diff1_histo = new TH1D(*disc_histo);
diff1_histo->SetTitle(
    "Difference between distribution of invariant masses of all particles of "
    "opposite charge and all particles of same charge");

diff1_histo->Add(conc_histo, -1);

diff1_histo->SetEntries(disc_histo->GetEntries() - conc_histo->GetEntries());

diff1_histo->SetMinimum(1.7 * diff1_histo->GetMinimum());
diff1_histo->SetMaximum(1.3 * diff1_histo->GetMaximum());

diff1_histo->GetXaxis()->SetTitle("Invariant mass [GeV/c^2]");
diff1_histo->GetYaxis()->SetTitle("Number of occurrencies");
diff1_histo->SetLineColorAlpha(kGreen + 2, 0.35);
diff1_histo->SetFillColorAlpha(kGreen + 2, 0.35);
diff1_histo->SetMarkerStyle(21);
diff1_histo->SetMarkerSize(0.25);

// Fit gaussian distribution function
TF1* gauss_dist = new TF1("gauss", "gaus([0], [1], [2])", 0., 7.);
gauss_dist->SetParameters(1200, 0.89, 0.05);
gauss_dist->SetParName(0, "Fit Amplitude");
gauss_dist->SetParName(1, "Fit Mean (#mu)");
gauss_dist->SetParName(2, "Fit #sigma");

diff1_histo->Fit("gauss", "q", "", 0., 5.);

fit_func = diff1_histo->GetFunction("gauss");
fit_func->SetLineColor(kRed);
fit_func->SetLineWidth(2);

// Add legend
TLegend* legend_diff1 = new TLegend(0.74, 0.16, 0.98, 0.34);
legend_diff1->AddEntry(diff1_histo, "Experimental Points", "fp");
legend_diff1->AddEntry(fit_func, "Fitting Gaussian distribution", "l");

diff1_histo->Draw();
fit_func->Draw("SAME");
legend_diff1->Draw("SAME");

// Print fit function parameters and Chi square data
std::cout << "\n ALL PARTICLES DIFFERENCE -- GAUSSIAN FIT FUNCTION\n";
std::cout << "\n Parameter 0: " << fit_func->GetParameter(0) << " +- "
          << fit_func->GetParError(0);
std::cout << "\n Parameter 1 (average): " << fit_func->GetParameter(1)
          << " +- " << fit_func->GetParError(1);
std::cout << "\n Parameter 2 (RMS): " << fit_func->GetParameter(2) << " +- "
          << fit_func->GetParError(2);
std::cout << "\n Chi square: " << fit_func->GetChisquare();
std::cout << "\n NDF: " << fit_func->GetNDF();
std::cout << "\n Chi/NDF: " << fit_func->GetChisquare() / fit_func->GetNDF();
std::cout << "\n Probability: " << fit_func->GetProb() << '\n';
std::cout << "\n--------------------- \n\n";

// Difference histogram between invariant masses of pions and kaons of
// opposite charges and of same charge

canva_k_star->cd(3);

TH1D* diff2_histo = new TH1D(*disc_pk_histo);
diff2_histo->SetTitle(
    "Difference between distribution of invariant masses "
    "of pions and kaons of opposite charges and of same "
```

```
      "charges");

  diff2_histo->Add(conc_pk_histo, -1);

  diff2_histo->SetEntries(disc_pk_histo->GetEntries() -
                          conc_pk_histo->GetEntries());

  diff2_histo->SetMinimum(diff1_histo->GetMinimum());
  diff2_histo->SetMaximum(diff1_histo->GetMaximum());

  diff2_histo->GetXaxis()->SetTitle("Invariant mass [GeV/c^2]");
  diff2_histo->GetYaxis()->SetTitle("Number of occurrencies");
  diff2_histo->SetLineColorAlpha(kGreen + 2, 0.35);
  diff2_histo->SetFillColorAlpha(kGreen + 2, 0.35);
  diff2_histo->SetMarkerStyle(21);
  diff2_histo->SetMarkerSize(0.25);

  // Fitting with gaussian distribution

  diff2_histo->Fit("gauss", "q", "", 0., 5.);

  fit_func = diff2_histo->GetFunction("gauss");
  fit_func->SetLineColor(kRed);
  fit_func->SetLineWidth(2);

  // Add legend
  TLegend* legend_diff2 = new TLegend(0.74, 0.16, 0.98, 0.34);
  legend_diff2->AddEntry(diff2_histo, "Experimental Points", "fp");
  legend_diff2->AddEntry(fit_func, "Fitting Gaussian distribution", "l");

  diff2_histo->Draw();
  fit_func->Draw("SAME");
  legend_diff2->Draw("SAME");

  // Print fit function parameters and Chi square data
  std::cout << "\n PIONS & KAONS DIFFERENCE -- GAUSSIAN FIT FUNCTION\n";
  std::cout << "\n Parameter 0: " << fit_func->GetParameter(0) << " +- "
            << fit_func->GetParError(0);
  std::cout << "\n Parameter 1 (average): " << fit_func->GetParameter(1)
            << " +- " << fit_func->GetParError(1);
  std::cout << "\n Parameter 2 (RMS): " << fit_func->GetParameter(2) << " +- "
            << fit_func->GetParError(2);
  std::cout << "\n Chi square: " << fit_func->GetChisquare();
  std::cout << "\n NDF: " << fit_func->GetNDF();
  std::cout << "\n Chi/NDF: " << fit_func->GetChisquare() / fit_func->GetNDF();
  std::cout << "\n Probability: " << fit_func->GetProb() << '\n';
  std::cout << "\n--------------------- \n\n";

  canva_k_star->cd(1);

  // Fit and draw K* decay products histogram

  kstar_histo->GetXaxis()->SetTitle("Invariant mass [GeV/c^2]");
  kstar_histo->GetYaxis()->SetTitle("Number of occurrencies");
  kstar_histo->SetLineColorAlpha(kMagenta - 7, 0.35);
  kstar_histo->SetFillColorAlpha(kMagenta - 7, 0.35);
  kstar_histo->SetMarkerStyle(21);
  kstar_histo->SetMarkerSize(0.7);

  // fit gaussian function
  kstar_histo->Fit("gauss", "q", "", 0, 4.);
  fit_func = kstar_histo->GetFunction("gauss");
  fit_func->SetLineColor(kBlue);
  fit_func->SetLineWidth(2);

  // Print fit function parameters and Chi square data
  std::cout << "\n K* BENCHMARK HISTOGRAM -- GAUSSIAN FIT FUNCTION\n";
  std::cout << "\n Parameter 0: " << fit_func->GetParameter(0) << " +- "
            << fit_func->GetParError(0);
  std::cout << "\n Parameter 1 (average): " << fit_func->GetParameter(1)
            << " +- " << fit_func->GetParError(1);
  std::cout << "\n Parameter 2 (RMS): " << fit_func->GetParameter(2) << " +- "
            << fit_func->GetParError(2);
  std::cout << "\n Chi square: " << fit_func->GetChisquare();
  std::cout << "\n NDF: " << fit_func->GetNDF();
  std::cout << "\n Chi/NDF: " << fit_func->GetChisquare() / fit_func->GetNDF();
```

```cpp
    std::cout << "\n Probability: " << fit_func->GetProb() << '\n';
    std::cout << "\n--------------------- \n\n";

    // Add legend
    TLegend* legend_kstar = new TLegend(0.74, 0.16, 0.98, 0.34);
    legend_kstar->AddEntry(kstar_histo, "Expected Points", "fp");
    legend_kstar->AddEntry(fit_func, "Fitting Gaussian distribution", "l");

    kstar_histo->Draw("e");
    fit_func->Draw("SAME");
    legend_kstar->Draw("SAME");

    // ---- OUTPUT FILES ----

    // PDF Output
    canva_retrieved->Print("data.pdf(", "Title:Canva retrieved");
    canva_k_star->Print("data.pdf)", "Title:Canva k star");

    // ROOT Output
    TFile* data_out_root = new TFile("data.root", "recreate");

    canva_retrieved->Write();
    canva_k_star->Write();

    data_out_root->Close();

    // C Output
    TFile* data_out_c = new TFile("data.C", "recreate");

    canva_retrieved->Write();
    canva_k_star->Write();

    data_out_c->Close();
}
```