Generazione e analisi di eventi di collisione fra particelle

Lorenzo Redighieri   Federico Tronchi   Luca Zoppetti

30/12/2023

1 Introduzione

Nel corso di tre sessioni di laboratorio sono stati approfonditi i temi del polimorfismo dinamico, del metodo Monte Carlo e dell'analisi dati costruendo un simulatore di eventi di collisione fra particelle con l'ausilio del framework ROOT. Negli eventi di collisione sono state trattate 7 particelle: pioni positivi e negativi ($\pi\pm$), kaoni positivi e negativi ($k\pm$), protoni ($p+$), antiprotoni ($p-$) e particelle K*, che decadono rapidamente in un pione e un kaone di carica opposta. Il programma, il cui scopo è rilevare la presenza della particella K* tramite l'analisi degli effetti del suo decadimento, è diviso in due componenti principali: quella di generazione casuale degli eventi e quella di analisi dati. La prima si occupa di generare e salvare in un file ROOT gli istogrammi con i dati relativi agli eventi di collisione fra le particelle. È implementata nella macro `generate.cpp`, che può essere eseguita in modalità compilata dopo aver compilato nell'ordine anche i file da cui dipende: `ParticleType.cpp`, `ResonanceType.cpp` e `Particle.cpp`. Per eseguirla è necessario indicare due parametri come argomento della funzione `generate()` nel terminale di ROOT: il numero di eventi da generare e il nome del file ROOT in cui salvare gli istogrammi. La seconda parte invece si occupa di eseguire le sottrazioni fra istogrammi e i fit per rilevare il segnale della particella K*. È una macro autonoma implementata interamente nel file `analyse.cpp`. Dopo averla compilata, essa può essere eseguita indicando il nome del file da analizzare come parametro della funzione `analyse()` nel terminale di ROOT. Di seguito vengono trattate le metodologie utilizzate nella scrittura del codice e i risultati del lavoro.

2 Struttura del codice

Per rappresentare le particelle sono state implementate tre classi: `ParticleType`, `ResonanceType` e `Particle`.

`ParticleType` contiene le informazioni basilari di una particella: nome, massa e carica, rappresentate da membri privati costanti. Come metodi pubblici sono presenti un costruttore parametrico, un distruttore virtuale, i *getters* dei membri privati (di cui uno virtuale) e la funzione virtuale `print` che stampa i valori di questi ultimi. Le funzioni virtuali sono poi reimplementate nella classe figlia `ResonanceType`, utilizzata per rappresentare una particella instabile, che eredita `ParticleType` con l'aggiunta della larghezza di risonanza, anch'essa membro privato costante. Il *getter* virtuale precedentemente citato è implementato per restituire la larghezza di risonanza.

Queste due classi definiscono le proprietà generiche che caratterizzano una tipologia di particelle, tuttavia, per rappresentare una particolare particella, occorre aggiungervi le informazioni cinematiche, che sono gestite nella classe `Particle`. Essa possiede come membri privati un vettore statico di puntatori (per sfruttare il polimorfismo dinamico) a `ParticleType` contenente i tipi di particelle, un `std::optional<int>` in cui è salvato l'indice dell'elemento che rappresenta il tipo della particella (`std::optional` aiuta a gestire i casi in cui viene impostato un indice errato) e una `struct` chiamata `Momentum` per rappresentare la quantità di moto. L'utilizzo di un vettore statico per i tipi di particelle aumenta l'efficienza del programma, poiché altrimenti sarebbe necessario copiare tutte le informazioni comuni alle particelle per ogni istanza di `Particle`.

La classe `Particle` dispone dei seguenti metodi pubblici:

- Il costruttore per creare la particella tramite il nome e la quantità di moto.
- I *getters* per i membri privati e la massa invariante.
- I *setters* per l'indice della particella e la quantità di moto.
- Tre metodi statici per contare i tipi di particelle disponibili, aggiungerne di nuovi e stamparli.
- Due metodi per il decadimento delle particelle instabili.
- Un metodo che stampa tutte le informazioni della particella.

3 Generazione degli eventi

La generazione degli eventi è stata gestita tramite la macro `generate.cpp`.

Le classi base del programma sono costruite per funzionare con qualsiasi particella e con qualsiasi numero di tipologie di particelle, che in questo specifico caso sono 7: pioni positivi e negativi, kaoni positivi e negativi, protoni, antiprotoni e particelle K*. Un evento consiste nella generazione di 100 particelle secondo le proporzioni riportate nella Tabella 1.

La quantità di moto di ciascuna particella è stata generata attraverso le seguenti distribuzioni: il suo angolo azimutale è stato estratto da una distribuzione uniforme tra 0 e 2π, l'angolo polare da un'altra distribuzione uniforme tra 0 e π e il modulo da una distribuzione esponenziale decrescente con media pari a 1 GeV.

Tipo	Massa (GeV/c^2)	Carica (e)	Larghezza (GeV/c^2)	Percentuale
Pione+	1.3957×10^{-1}	+1	0	40 %
Pione-	1.3957×10^{-1}	-1	0	40 %
Kaone+	4.9367×10^{-1}	+1	0	10 %
Kaone-	4.9367×10^{-1}	-1	0	10 %
Protone+	9.3827×10^{-1}	+1	0	4.5 %
Protone-	9.3827×10^{-1}	-1	0	4.5 %
K*	8.9166×10^{-1}	0	5.0×10^{-2}	1 %

Tabella 1: Nella tabella sono riportate le proprietà che caratterizzano le tipologie di particelle utilizzate nella simulazione e le proporzioni con le quali sono state generate in ogni evento. L'unità di misura e rappresenta la carica fondamentale, $e = 1.602176634 \times 10^{-19} C$.

Nell'eventualità in cui venga generata una particella K*, questa è fatta decadere attraverso il metodo `decayToBody()` in un pione e un kaone di carica opposta. Le particelle vengono successivamente combinate per creare gli istogrammi di massa invariante utilizzati nella parte di analisi. Vengono considerate le seguenti combinazioni:

- tutte le particelle;
- particelle di carica discorde;
- particelle di carica concorde;
- kaoni e pioni di carica discorde;
- kaoni e pioni di carica concorde;
- kaoni e pioni provenienti dal decadimento della stessa K*.

4 Analisi dati

I dati raccolti sono stati analizzati attraverso la macro `analyse.cpp`. Le abbondanze di particelle generate (Tab. 2) risultano compatibili con le percentuali di generazione (Tab. 1).

Per gli istogrammi degli angoli e della quantità di moto (Fig. 1) sono stati eseguiti dei fit (Tab. 3) per verificare la compatibilità con i dati di partenza della generazione, uniformi per i primi due ed esponenziale per il terzo, che risultano tutti compatibili con quanto atteso. Per ciascun fit è stato riportato il valore $\widetilde{\chi}^2 = \frac{\chi^2}{DOF}$, che costituisce una misura dell'accordo fra i dati e la relazione funzionale ipotizzata. Un'ipotesi associata a un $\widetilde{\chi}^2 \sim 1$ è considerata accettabile.

Il segnale della particella K* è stato estratto prima studiando la distribuzione della massa invariante dei prodotti del decadimento (*Decay products*, Fig. 2), poi attraverso una sottrazione di istogrammi. Questo secondo metodo si basa sul fatto che il decadimento della K* (che genera pioni e kaoni di carica opposta) produce una leggera differenza nella distribuzione di massa invariante fra le particelle di carica discorde e di carica concorde. L'istogramma *Opposite charge - same charge* (Fig. 2) è stato definito come la sottrazione dell'istogramma di massa invariante di particelle aventi stessa carica dall'istogramma di massa invariante di particelle di carica opposta. Attraverso il fit ad una distribuzione gaussiana sono state ricavate massa e larghezza della K*, corrispondenti a media e deviazione standard della gaussiana. Per l'istogramma *Opposite charge - same charge (kaon & pion)* (Fig. 2) si è proceduto in maniera analoga, sottraendo l'istogramma di massa invariante dei kaoni e pioni con stessa carica dall'istogramma di massa invariante dei kaoni e pioni con carica opposta. Tutti i valori (Tab. 4) di massa e larghezza ricavati dai fit gaussiani risultano compatibili entro 2σ con quelli in input alla generazione.

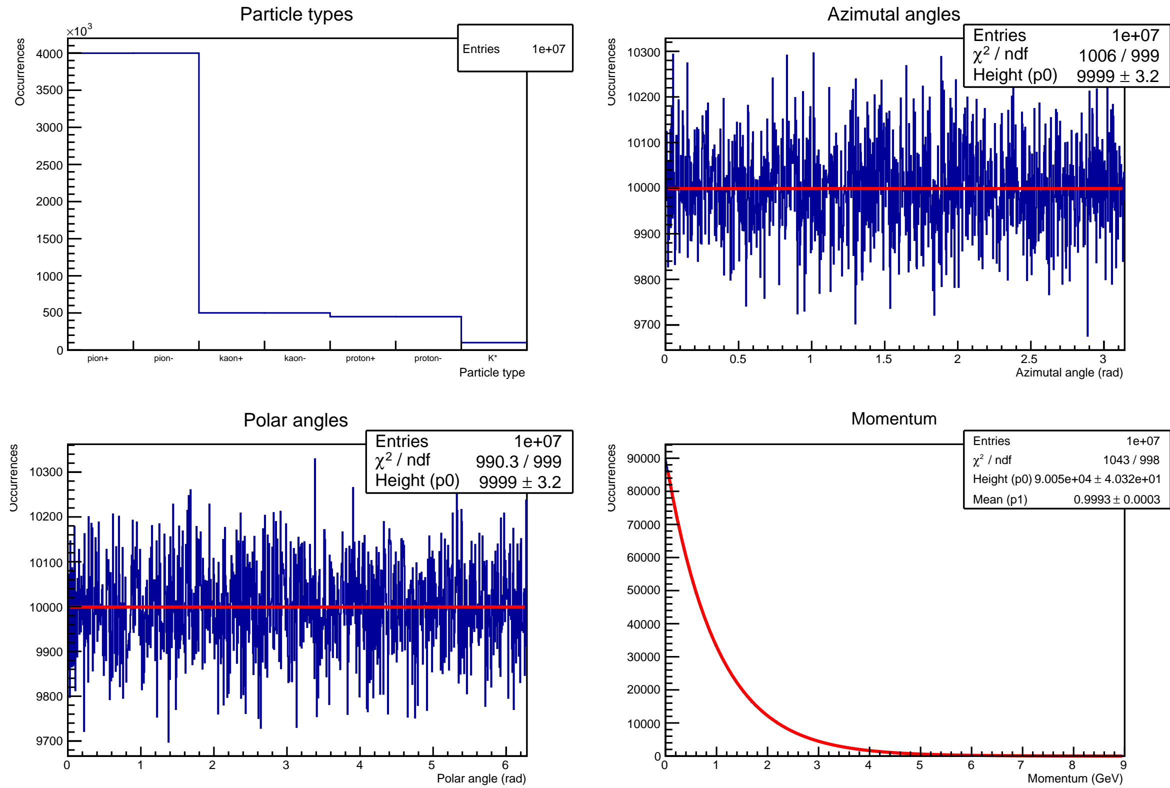

Figura 1: La figura contiene gli istogrammi relativi ad abbondanza di distribuzione delle particelle, angolo azimutale, angolo polare e quantità di moto. Ad eccezione del primo sono state definite delle funzioni per il fit degli istogrammi, che sono raffigurate in rosso. I dati raccolti dal primo istogramma sono riportati in Tabella 2. Nelle legende sono visualizzati i parametri ottenuti dai fit, riportati anche in Tabella 3.

Specie	Occorrenze Osservate	Occorrenze Attese
Pione+	$(3.999 \pm 0.002) \times 10^6$	4×10^6
Pione-	$(4.000 \pm 0.002) \times 10^6$	4×10^6
Kaone+	$(5.008 \pm 0.007) \times 10^5$	5×10^5
Kaone-	$(5.002 \pm 0.007) \times 10^5$	5×10^5
Protone	$(4.502 \pm 0.007) \times 10^5$	4.5×10^5
Antiprotone	$(4.498 \pm 0.007) \times 10^5$	4.5×10^5
K*	$(1.003 \pm 0.003) \times 10^5$	1×10^5

Tabella 2: Nella tabella sono riportati il tipo delle particelle generate, l'occorrenza di particelle generate per ogni tipo, corrispondente al numero di occorrenze per bin nell'istogramma *Particle types* (Fig. 1), e il numero atteso di particelle generate. Trattandosi di un esperimento di conteggio, l'incertezza sul numero osservato di particelle è stata calcolata come $\sqrt{N}$. Le occorrenze osservate risultano compatibili con quelle attese.

Distribuzione	Ordinata all'origine	Media (GeV)	χ^2	DOF	$\widetilde{\chi}^2$
Fit uniforme, angolo azimutale	$(9.999 \pm 0.003) \times 10^3$	ND	1006	999	1.01
Fit uniforme, angolo polare	$(9.999 \pm 0.003) \times 10^3$	ND	990.3	999	0.99
Fit esponenziale, modulo dell'impulso	$(9.005 \pm 0.004) \times 10^4$	$(9.993 \pm 0.003) \times 10^{-1}$	1043	998	1.05

Tabella 3: Nella tabella vengono mostrati i risultati dell'adattamento dei fit agli istogrammi *Azimutal angles, Polar angles, Momentum* (Fig. 1). Ricavando dai fit χ^2 e gradi di libertà (DOF) è stato calcolato $\widetilde{\chi}^2$, confermando che i valori generati sono consistenti con le distribuzioni attese poiché $\widetilde{\chi}^2 \sim 1$. ND indica che il parametro non è disponibile per il fit di riferimento.

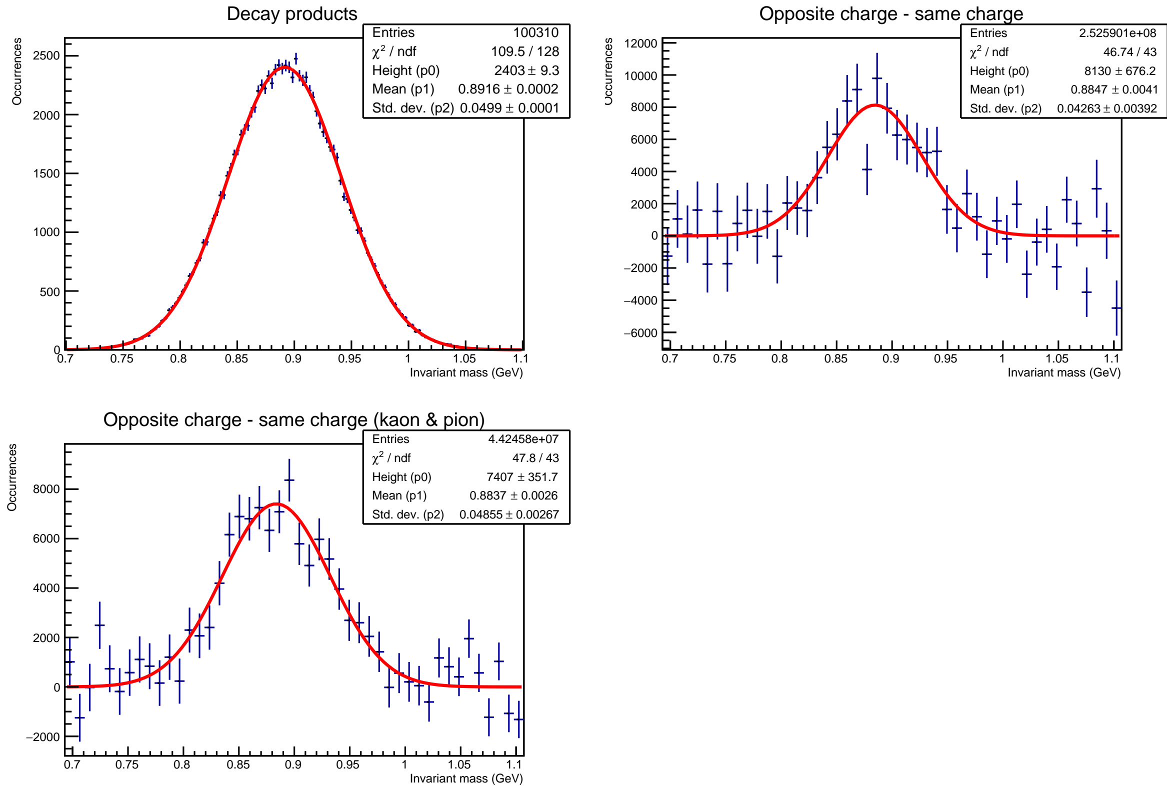

Figura 2: In figura vengono mostrati gli istogrammi di massa invariante utilizzati per rilevare il segnale della particella K*. Nel primo sono riportati i valori di massa invariante calcolati unicamente tra i prodotti dei decadimenti. Il secondo e il terzo istogramma sono il risultato di una sottrazione di istogrammi, utilizzata per far emergere il segnale di K* dal fondo. Attraverso un fit, i cui parametri sono visibili nella legenda e in Tabella 4, è stata verificata la consistenza delle distribuzioni con una gaussiana, rappresentata in rosso.

Istogramma	Ampiezza	Media (GeV)	Sigma (GeV)	$\widetilde{\chi}^2$
Massa invariante ottenuta dalle coppie di particelle decadute dalla particella K*	$(2.403 \pm 0.018) \times 10^3$	$(8.916 \pm 0.004) \times 10^{-1}$	$(4.99 \pm 0.02) \times 10^{-2}$	0.86
Massa invariante ottenuta da differenza delle combinazioni di carica discorde e concorde	$(8.1 \pm 1.4) \times 10^3$	$(8.85 \pm 0.08) \times 10^{-1}$	$(4.3 \pm 0.8) \times 10^{-2}$	1.09
Massa invariante ottenuta da differenza delle combinazioni kaone-pione di carica discorde e concorde	$(7.4 \pm 0.8) \times 10^2$	$(8.84 \pm 0.06) \times 10^{-1}$	$(4.9 \pm 0.6) \times 10^{-2}$	1.11

Tabella 4: La tabella riporta i risultati dei fit agli istogrammi (Fig. 2) definiti per misurare massa e ampiezza della K*, corrispondenti rispettivamente a media e deviazione standard della distribuzione gaussiana. Le incertezze sono state calcolate come il doppio dell'incertezza fornita dal fit, per cui i risultati sono compatibili entro 2σ con i dati in input alla generazione. I fit sono accettabili in quanto $\widetilde{\chi}^2 \sim 1$.

Appendice

ParticleType.hpp

```
#include <string>

#ifndef PARTICLE_TYPE_HPP
#define PARTICLE_TYPE_HPP

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

  std::string getName() const;
  double getMass() const;
  int getCharge() const;
  virtual double getWidth() const;
  virtual void print() const;

 private:
  std::string const m_name;
  double const m_mass;
  int const m_charge;
};

#endif
```

ParticleType.cpp

```
#include "ParticleType.hpp"

#include <iostream>

// constructor

ParticleType::ParticleType(std::string const& name, double mass, int charge)
    : m_name{name}, m_mass{mass}, m_charge{charge} {}

// public methods

std::string ParticleType::getName() const { return m_name; }

double ParticleType::getMass() const { return m_mass; }

int ParticleType::getCharge() const { return m_charge; }

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

```

```cpp
void ParticleType::print() const {
  std::cout << "Name: " << m_name << '\n'
            << "Mass: " << m_mass << '\n'
            << "Charge: " << m_charge << '\n';
}
```

ResonanceType.hpp

```cpp
#ifndef RESONANCE_TYPE_HPP
#define RESONANCE_TYPE_HPP

#include "ParticleType.hpp"

class ResonanceType : public ParticleType {
 public:
  ResonanceType(std::string const&, double, int, double);
  ~ResonanceType() override = default;

  double getWidth() const override;
  void print() const override;

 private:
  double const m_width;
};

#endif
```

ResonanceType.cpp

```cpp
#include "ResonanceType.hpp"

#include <iostream>

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

// public methods
double ResonanceType::getWidth() const { return m_width; }

void ResonanceType::print() const {
  std::cout << "Name: " << getName() << '\n'
            << "Mass: " << getMass() << '\n'
            << "Charge: " << getCharge() << '\n'
            << "Width: " << m_width << '\n';
}
```

Particle.hpp

```cpp
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include <memory>
#include <optional>
#include <string>
#include <vector>

class ParticleType;

struct PolarVector {
  double r;
  double theta;
  double phi;
};

struct Momentum {
  double x;
  double y;
  double z;

  Momentum(double, double, double);
```

```
23   Momentum(PolarVector const&);
24
25   PolarVector getPolar() const;
26   Momentum operator+(Momentum const&) const;
27   double operator*(Momentum const&) const;
28 };
29
30 class Particle {
31  public:
32   Particle(std::string const& = "", Momentum const& = {0., 0., 0.});
33
34   void printData() const;
35
36   int decayToBody(Particle&, Particle&) const;
37
38   // setters
39
40   void setIndex(std::string const&);
41   void setIndex(int);
42   void setMomentum(double, double, double);
43   void setMomentum(Momentum const&);
44
45   // getters
46
47   std::optional<int> getIndex() const;
48   Momentum getMomentum() const;
49   double getEnergy() const;
50   double getMass() const;
51   double getCharge() const;
52   std::string getName() const;
53   double getInvariantMass(Particle const&) const;
54
55   // static methods
56
57   static int countParticleTypes();
58   static void addParticleType(std::string const&, double, int, double = 0.);
59   static void printParticleTypes();
60
61  private:
62   Momentum m_momentum;
63   std::optional<int> m_index;
64
65   void boost(double, double, double);
66
67   static std::vector<std::unique_ptr<ParticleType>> m_particle_types;
68   static std::optional<int> mFindParticleIndex(std::string const&);
69 };
70
71 #endif
```

Particle.cpp

```
1  #include "Particle.hpp"
2
3  #include <algorithm>
4  #include <cmath>
5  #include <iostream>
6
7  #include "ParticleType.hpp"
8  #include "ResonanceType.hpp"
9  #include "TMath.h"
10
11 // init static members
12
13 std::vector<std::unique_ptr<ParticleType>> Particle::m_particle_types{};
14
15 // momentum constructors
16
17 Momentum::Momentum(double x, double y, double z) : x{x}, y{y}, z{z} {}
18
19 Momentum::Momentum(PolarVector const& polar)
20     : x{polar.r * std::sin(polar.theta) * std::cos(polar.phi)},
21       y{polar.r * std::sin(polar.theta) * std::sin(polar.phi)},
```

```
22       z{polar.r * std::cos(polar.theta)} {}
23
24 // momentum functions
25
26 PolarVector Momentum::getPolar() const {
27   double r = std::sqrt(x * x + y * y + z * z);
28   double theta = std::acos(z / r);
29   double phi = std::atan(y / x);
30
31   // convert to the correct phi by adjusting the atan(y / x) result based on
32   // x and y coordinates
33   if (phi > 0 && x < 0) {
34     phi += TMath::Pi();
35   } else if (phi < 0 && x > 0) {
36     phi += TMath::Pi() * 2.;
37   } else if (phi < 0 && x < 0) {
38     phi += TMath::Pi();
39   }
40
41   return {r, theta, phi};
42 };
43
44 // momentum operators
45
46 Momentum Momentum::operator+(Momentum const& momentum) const {
47   return {x + momentum.x, y + momentum.y, z + momentum.z};
48 }
49
50 double Momentum::operator*(Momentum const& momentum) const {
51   return x * momentum.x + y * momentum.y + z * momentum.z;
52 }
53
54 // constructor
55
56 Particle::Particle(std::string const& name, Momentum const& momentum)
57     : m_momentum{momentum}, m_index{std::nullopt} {
58   if (name != "") {
59     setIndex(name);
60   }
61 }
62
63 // public methods
64
65 void Particle::printData() const {
66   if (m_index != std::nullopt) {
67     std::cout << "Index: " << m_index.value() << '\n'
68               << "Name: " << m_particle_types[m_index.value()]->getName()
69               << '\n'
70               << "Momentum: (" << m_momentum.x << ", " << m_momentum.y << ", "
71               << m_momentum.z << ")\n";
72   }
73 }
74
75 int Particle::decayToBody(Particle& dau1, Particle& dau2) const {
76   if (getMass() == 0.0) {
77     printf("Decayment cannot be preformed if mass is zero\n");
78     return 1;
79   }
80
81   double massMot = getMass();
82   double massDau1 = dau1.getMass();
83   double massDau2 = dau2.getMass();
84
85   if (m_index != std::nullopt) {  // add width effect
86
87     // gaussian random numbers
88
89     float x1, x2, w, y1;
90
91     double invnum = 1. / RAND_MAX;
92     do {
93       x1 = 2.0 * std::rand() * invnum - 1.0;
94       x2 = 2.0 * std::rand() * invnum - 1.0;
```

```
95        w = x1 * x1 + x2 * x2;
96      } while (w >= 1.0);
97
98      w = std::sqrt((-2.0 * std::log(w)) / w);
99      y1 = x1 * w;
100
101     massMot += m_particle_types[m_index.value()]->getWidth() * y1;
102   }
103
104   if (massMot < massDau1 + massDau2) {
105     printf(
106         "Decayment cannot be preformed because mass is too low in this "
107         "channel\n");
108     return 2;
109   }
110
111   double pout =
112       std::sqrt(
113           (massMot * massMot - (massDau1 + massDau2) * (massDau1 + massDau2)) *
114           (massMot * massMot - (massDau1 - massDau2) * (massDau1 - massDau2))) /
115       massMot * 0.5;
116
117   double norm = 2 * M_PI / RAND_MAX;
118
119   double phi = std::rand() * norm;
120   double theta = std::rand() * norm * 0.5 - M_PI / 2.;
121
122   dau1.setMomentum(pout * std::sin(theta) * std::cos(phi),
123                    pout * std::sin(theta) * std::sin(phi),
124                    pout * std::cos(theta));
125   dau2.setMomentum(-pout * std::sin(theta) * std::cos(phi),
126                    -pout * std::sin(theta) * std::sin(phi),
127                    -pout * std::cos(theta));
128
129   double energy =
130       std::sqrt(m_momentum.x * m_momentum.x + m_momentum.y * m_momentum.y +
131                 m_momentum.z * m_momentum.z + massMot * massMot);
132
133   double bx = m_momentum.x / energy;
134   double by = m_momentum.y / energy;
135   double bz = m_momentum.z / energy;
136
137   dau1.boost(bx, by, bz);
138   dau2.boost(bx, by, bz);
139
140   return 0;
141 }
142
143 // getters
144
145 std::optional<int> Particle::getIndex() const { return m_index; }
146
147 Momentum Particle::getMomentum() const { return m_momentum; }
148
149 double Particle::getEnergy() const {
150   if (m_index != std::nullopt) {
151     return std::sqrt(std::pow(m_particle_types[m_index.value()]->getMass(), 2) +
152                      m_momentum * m_momentum);
153   } else {
154     std::cout
155         << "ERROR: This particle has no energy because its index is invalid!"
156         << '\n';
157
158     return 0;
159   }
160 }
161
162 double Particle::getMass() const {
163   if (m_index != std::nullopt) {
164     return m_particle_types[m_index.value()]->getMass();
165   } else {
166     std::cout
167         << "ERROR: This particle has no mass because its index is invalid!"
```

```cpp
        << '\n';

    return 0;
  }
}

double Particle::getCharge() const {
  if (m_index != std::nullopt) {
    return m_particle_types[m_index.value()]->getCharge();
  } else {
    std::cout
        << "ERROR: This particle has no charge because its index is invalid!"
        << '\n';

    return 0;
  }
}

std::string Particle::getName() const {
  if (m_index != std::nullopt) {
    return m_particle_types[m_index.value()]->getName();
  } else {
    std::cout
        << "ERROR: This particle has no name because its index is invalid!"
        << '\n';

    return "";
  }
}

double Particle::getInvariantMass(Particle const& p) const {
  auto newMomentum = m_momentum + p.getMomentum();

  return std::sqrt(std::pow(getEnergy() + p.getEnergy(), 2) -
                   newMomentum * newMomentum);
}

// setters

void Particle::setIndex(std::string const& name) {
  m_index = mFindParticleIndex(name);
}

void Particle::setIndex(int index) {
  if (index >= 0 && index < static_cast<int>(m_particle_types.size())) {
    m_index = index;
  } else {
    std::cout << "ERROR: The index \"" << index
              << "\" does not refer to a particle type!" << '\n';
  }
}

void Particle::setMomentum(double px, double py, double pz) {
  m_momentum = {px, py, pz};
}

void Particle::setMomentum(Momentum const& momentum) { m_momentum = momentum; }

// static methods

int Particle::countParticleTypes() { return m_particle_types.size(); }

void Particle::addParticleType(std::string const& name, double mass, int charge,
                               double width) {
  auto existing_index = mFindParticleIndex(name);

  if (existing_index == std::nullopt) {
    if (width == 0) {
      m_particle_types.push_back(
          std::unique_ptr<ParticleType>{new ParticleType{name, mass, charge}});
    } else {
      m_particle_types.push_back(std::unique_ptr<ParticleType>{
          new ResonanceType{name, mass, charge, width}});
```

```
    }
  } else {
    std::cout << "ERROR: The \"" << name << "\" particle type already exists!"
              << '\n';
  }
}

void Particle::printParticleTypes() {
  auto v_end = m_particle_types.end();

  for (auto it = m_particle_types.begin(); it < v_end; ++it) {
    (*it)->print();

    if (it != v_end - 1) {
      std::cout << '\n';
    }
  }
}

// private methods

std::optional<int> Particle::mFindParticleIndex(std::string const& name) {
  auto v_begin = m_particle_types.begin();
  auto v_end = m_particle_types.end();

  auto it = std::find_if(v_begin, v_end,
                         [&name](std::unique_ptr<ParticleType> const& pt) {
                           return pt->getName() == name;
                         });

  if (it == v_end) {
    return std::nullopt;
  }

  return std::distance(m_particle_types.begin(), it);
}

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 * m_momentum.x + by * m_momentum.y + bz * m_momentum.z;
  double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

  m_momentum.x += gamma2 * bp * bx + gamma * bx * energy;
  m_momentum.y += gamma2 * bp * by + gamma * by * energy;
  m_momentum.z += gamma2 * bp * bz + gamma * bz * energy;
}
```

generate.cpp

```
#include <iostream>
#include <string>
#include <vector>

#include "Particle.hpp"
#include "ParticleType.hpp"
#include "ResonanceType.hpp"
#include "TBenchmark.h"
#include "TFile.h"
#include "TH1.h"
#include "TList.h"
#include "TMath.h"
#include "TRandom.h"

/**
 * Helper function to fill invariant mass histograms with data coming from two
 * particles.
 */
void fillHistograms(Particle const& particle_1, Particle const& particle_2,
                    TH1F* invm_all_h, TH1F* invm_opposite_charge_h,
```

```
                    TH1F* invm_same_charge_h, TH1F* invm_pion_kaon_opposite_h,
                    TH1F* invm_pion_kaon_same_h) {
  auto invariant_mass = particle_1.getInvariantMass(particle_2);

  // invariant mass with all particles
  invm_all_h->Fill(invariant_mass);

  // invariant mass with opposite charge particles
  if (particle_2.getCharge() * particle_1.getCharge() < 0) {
    invm_opposite_charge_h->Fill(invariant_mass);
  }

  // invariant mass with same charge particles
  if (particle_2.getCharge() * particle_1.getCharge() > 0) {
    invm_same_charge_h->Fill(invariant_mass);
  }

  // invariant mass with pion+ and kaon- or pion- and kaon+
  if ((particle_1.getName() == "pion+" && particle_2.getName() == "kaon-") ||
      (particle_1.getName() == "kaon-" && particle_2.getName() == "pion+") ||
      (particle_1.getName() == "pion-" && particle_2.getName() == "kaon+") ||
      (particle_1.getName() == "kaon+" && particle_2.getName() == "pion-")) {
    invm_pion_kaon_opposite_h->Fill(invariant_mass);
  }

  // invariant mass with pion+ and kaon+ or piaon- and kaon-
  if ((particle_1.getName() == "pion+" && particle_2.getName() == "kaon+") ||
      (particle_1.getName() == "kaon+" && particle_2.getName() == "pion+") ||
      (particle_1.getName() == "pion-" && particle_2.getName() == "kaon-") ||
      (particle_1.getName() == "kaon-" && particle_2.getName() == "pion-")) {
    invm_pion_kaon_same_h->Fill(invariant_mass);
  }
}

void generate(int n_gen, const char* file_name) {
  gBenchmark->Start("Benchmark");

  R__LOAD_LIBRARY(ParticleType_cpp.so)
  R__LOAD_LIBRARY(ResonanceType_cpp.so)
  R__LOAD_LIBRARY(Particle_cpp.so)

  // mass and width measured in GeV/c^2
  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("k*", 0.89166, 0, 0.050);

  gRandom->SetSeed();

  TList* histo_list = new TList();

  // particle histograms
  TH1I* particle_types_h =
      new TH1I("particle_types_h", "Particle types", 7, 0, 7);
  histo_list->Add(particle_types_h);  // 0

  TH1F* azimutal_angles_h =
      new TH1F("azimutal_angles_h", "Azimutal angles", 1e3, 0, TMath::Pi());
  histo_list->Add(azimutal_angles_h);  // 1

  TH1F* polar_angles_h =
      new TH1F("polar_angles_h", "Polar angles", 1e3, 0, TMath::Pi() * 2.);
  histo_list->Add(polar_angles_h);  // 2

  TH1F* momentum_h = new TH1F("momentum_h", "Momentum", 1e3, 0, 9);
  histo_list->Add(momentum_h);  // 3

  TH1F* momentum_xy_h = new TH1F("momentum_xy_h", "Momentum xy", 1e3, 0, 9);
  histo_list->Add(momentum_xy_h);  // 4

```

```
94  TH1F* energy_h = new TH1F("energy_h", "Energy", 1e4, 0, 4);
95  histo_list->Add(energy_h);  // 5
96
97  // invariant mass histograms
98  TH1F* invm_all_h =
99      new TH1F("invm_all_h", "Invariant mass, all particles", 1e4, 0, 9);
100 invm_all_h->Sumw2();
101 histo_list->Add(invm_all_h);  // 6
102
103 TH1F* invm_opposite_charge_h = new TH1F(
104     "invm_opposite_charge_h", "Invariant mass, opposite charge", 1e4, 0, 9);
105 invm_opposite_charge_h->Sumw2();
106 histo_list->Add(invm_opposite_charge_h);  // 7
107
108 TH1F* invm_same_charge_h =
109     new TH1F("invm_same_charge_h", "Invariant mass, same charge", 1e4, 0, 9);
110 invm_same_charge_h->Sumw2();
111 histo_list->Add(invm_same_charge_h);  // 8
112
113 TH1F* invm_pion_kaon_opposite_h =
114     new TH1F("invm_pion_kaon_opposite_h",
115              "Invariant mass, pion+ and kaon- or pion- and kaon+", 1e4, 0, 9);
116 invm_pion_kaon_opposite_h->Sumw2();
117 histo_list->Add(invm_pion_kaon_opposite_h);  // 9
118
119 TH1F* invm_pion_kaon_same_h =
120     new TH1F("invm_pion_kaon_same_h",
121              "Invariant mass, pion+ and kaon+ or pion- and kaon-", 1e4, 0, 9);
122 invm_pion_kaon_same_h->Sumw2();
123 histo_list->Add(invm_pion_kaon_same_h);  // 10
124
125 TH1F* invm_decayed_h =
126     new TH1F("invm_decayed_h", "Invariant mass, decayed particles from K*",
127              1e3, 0.6, 1.2);
128 invm_decayed_h->Sumw2();
129 histo_list->Add(invm_decayed_h);  // 11
130
131 for (int i{}; i < n_gen; ++i) {
132   std::vector<Particle> event_particles(100);
133
134   event_particles.reserve(150);
135
136   for (int j{}; j < 100; ++j) {
137     auto r = gRandom->Exp(1);  // GeV
138     auto theta = gRandom->Uniform(0, TMath::Pi());
139     auto phi = gRandom->Uniform(0, TMath::Pi() * 2.);
140
141     // convert polar to cartesian coordinates
142     event_particles[j].setMomentum(Momentum{PolarVector{r, theta, phi}});
143
144     auto x = gRandom->Uniform(0, 1);
145
146     if (x <= 0.4) {
147       event_particles[j].setIndex("pion+");
148     } else if (x <= 0.8) {
149       event_particles[j].setIndex("pion-");
150     } else if (x <= 0.85) {
151       event_particles[j].setIndex("kaon+");
152     } else if (x <= 0.9) {
153       event_particles[j].setIndex("kaon-");
154     } else if (x <= 0.945) {
155       event_particles[j].setIndex("proton+");
156     } else if (x <= 0.99) {
157       event_particles[j].setIndex("proton-");
158     } else {
159       event_particles[j].setIndex("k*");
160
161       auto decay_into = gRandom->Uniform(0, 1);
162
163       Particle decay_product_1{};
164       Particle decay_product_2{};
165
166       if (decay_into <= 0.5) {
```

```
          decay_product_1.setIndex("pion+");
          decay_product_2.setIndex("kaon-");
        } else {
          decay_product_1.setIndex("pion-");
          decay_product_2.setIndex("kaon+");
        }

        event_particles[j].decayToBody(decay_product_1, decay_product_2);

        // fill decay products invariant mass histogram
        auto invariant_mass_products =
            decay_product_1.getInvariantMass(decay_product_2);

        invm_decayed_h->Fill(invariant_mass_products);

        event_particles.push_back(decay_product_1);
        event_particles.push_back(decay_product_2);
      }

      // fill generation histograms
      auto const& new_particle = event_particles[j];

      // type
      particle_types_h->Fill(new_particle.getIndex().value());

      auto momentum = new_particle.getMomentum();
      auto polar_momentum = momentum.getPolar();

      // azimutal angle
      azimutal_angles_h->Fill(polar_momentum.theta);

      // polar angle
      polar_angles_h->Fill(polar_momentum.phi);

      // momentum
      momentum_h->Fill(std::sqrt(momentum * momentum));

      // momentum on xy plane
      momentum_xy_h->Fill(
          std::sqrt(momentum.x * momentum.x + momentum.y * momentum.y));

      // energy
      energy_h->Fill(new_particle.getEnergy());

      // fill invariant mass histograms. This loop improves performance because
      // it avoids unnecessary iterations in the loop after completing the event
      // generation
      if (new_particle.getName() != "k*") {
        for (auto invm_i = j - 1; invm_i >= 0; --invm_i) {
          auto const& invm_particle = event_particles[invm_i];

          if (invm_particle.getName() == "k*") {
            continue;
          }

          fillHistograms(new_particle, invm_particle, invm_all_h,
                         invm_opposite_charge_h, invm_same_charge_h,
                         invm_pion_kaon_opposite_h, invm_pion_kaon_same_h);
        }
      }
    }

    // fill invariant mass histograms with combinations including decayed
    // particles
    auto event_particles_begin = event_particles.begin();
    auto event_particles_end = event_particles.end();

    for (auto it = event_particles_begin + 100; it < event_particles_end;
         ++it) {
      auto const& decayed_particle = *it;

      for (auto invm_it = event_particles_begin; invm_it < it; ++invm_it) {
        auto const& invm_particle = *invm_it;
```

```
240
241         if (invm_particle.getName() == "k*") {
242           continue;
243         }
244
245         fillHistograms(decayed_particle, invm_particle, invm_all_h,
246                        invm_opposite_charge_h, invm_same_charge_h,
247                        invm_pion_kaon_opposite_h, invm_pion_kaon_same_h);
248       }
249     }
250   }
251
252   TFile* file = new TFile(file_name, "RECREATE");
253
254   histo_list->Write();
255
256   file->Close();
257
258   gBenchmark->Show("Benchmark");
259 }
```

analyse.cpp

```
1  #include <array>
2  #include <iostream>
3
4  #include "TCanvas.h"
5  #include "TF1.h"
6  #include "TFile.h"
7  #include "TH1.h"
8  #include "TKey.h"
9  #include "TMath.h"
10 #include "TROOT.h"
11 #include "TStyle.h"
12
13 const char* PARTICLE_NAMES[7]{"pion+",   "pion-",   "kaon+", "kaon-",
14                               "proton+", "proton-", "K*"};
15
16 // Labels for fit parameters
17 const char* HEIGHT_LABEL = "Height (p0)";
18 const char* MEAN_LABEL = "Mean (p1)";
19 const char* STDDEV_LABEL = "Std. dev. (p2)";
20
21 // Histogram limits for gaussian fits
22 const double H_LOW = 0.7;
23 const double H_HIGH = 1.1;
24
25 // Define fit functions
26 Double_t gauss(Double_t* xx, Double_t* par) {
27   Double_t x = xx[0];
28   Double_t val =
29       par[0] * TMath::Exp(-(x - par[1]) * (x - par[1]) / (2 * par[2] * par[2]));
30   return val;
31 }
32
33 Double_t exp(Double_t* xx, Double_t* par) {
34   Double_t x = xx[0];
35   Double_t val = par[0] * TMath::Exp(-x / par[1]);
36   return val;
37 }
38
39 Double_t uniform(Double_t* xx, Double_t* par) { return par[0]; }
40
41 void analyse(const char* file_name) {
42   // Set histogram options. Show entries, parameters, errors and chi square/DOF
43   gStyle->SetOptFit(001);
44   gStyle->SetOptStat("e");
45
46   TFile* file = new TFile(file_name, "READ");
47
48   std::array<TH1*, 12> histo_array;
49
50   // Read histograms from TList and put them inside an array
```

```cpp
51  auto key_list = file->GetListOfKeys();
52
53  for (int i{}; i < 12; ++i) {
54    auto key = (TKey*)key_list->At(i);
55    histo_array[i] = (TH1*)file->Get(key->GetName());
56  }
57
58  std::array<double, 12> expected_entries{
59      1e7, 1e7, 1e7, 1e7, 1e7, 1e7, 5e8, 2.5e8, 2.5e8, 4.46e7, 4.45e7, 1e5};
60
61  // Print expected and real entries
62  std::cout << "ENTRIES" << '\n';
63
64  for (int i{}; i < 12; ++i) {
65    auto title = histo_array[i]->GetTitle();
66    auto entries = histo_array[i]->GetEntries();
67
68    std::cout << title << ": expected " << expected_entries[i] << ", got "
69              << entries << '\n';
70  }
71
72  auto particle_types_histogram = histo_array[0];
73
74  // Print generated occurrences
75  std::cout << "\nPARTICLE OCCURRENCES" << '\n';
76
77  for (int i{}; i < 7; ++i) {
78    auto occurrences = particle_types_histogram->GetBinContent(i + 1);
79
80    std::cout << "Particle " << PARTICLE_NAMES[i] << ": " << occurrences
81              << " +- " << std::sqrt(occurrences) << '\n';
82  }
83
84  // Create first canvas, with particle types, angles and momentum
85  TCanvas* particles_canvas = new TCanvas();
86  particles_canvas->Divide(2, 2);
87
88  // Add first histogram with particle type occurrences
89  particles_canvas->cd(1);
90  auto x_axis = particle_types_histogram->GetXaxis();
91  x_axis->CenterLabels();
92  for (int i{0}; i < 7; ++i) {
93    x_axis->SetBinLabel(i + 1, PARTICLE_NAMES[i]);
94  }
95  particle_types_histogram->SetXTitle("Particle type");
96  particle_types_histogram->SetYTitle("Occurrences");
97  particle_types_histogram->Draw();
98
99  // Add and fit azimutal angles histogram
100 TF1* azimutal_fit = new TF1("azimutal_fit", uniform, 0., TMath::Pi(), 1);
101 azimutal_fit->SetParameter(0, 10000);
102 azimutal_fit->SetParName(0, HEIGHT_LABEL);
103
104 particles_canvas->cd(2);
105 auto azimutal_angles_histogram = histo_array[1];
106 azimutal_angles_histogram->SetXTitle("Azimutal angle (rad)");
107 azimutal_angles_histogram->SetYTitle("Occurrences");
108 azimutal_angles_histogram->Fit(azimutal_fit, "Q");
109
110 std::cout << "\nAZIMUTAL FIT" << '\n'
111           << HEIGHT_LABEL << ": " << azimutal_fit->GetParameter(0) << " +- "
112           << azimutal_fit->GetParError(0) << '\n'
113           << "Chi square/NDF: "
114           << azimutal_fit->GetChisquare() / azimutal_fit->GetNDF() << '\n'
115           << "Probability: " << azimutal_fit->GetProb() << '\n';
116
117 // Add and fit polar angles histogram
118 TF1* polar_fit = new TF1("polar_fit", uniform, 0., TMath::TwoPi(), 1);
119 polar_fit->SetParameter(0, 10000);
120 polar_fit->SetParName(0, HEIGHT_LABEL);
121
122 particles_canvas->cd(3);
123 auto polar_angles_histogram = histo_array[2];
```

```
124 polar_angles_histogram->SetXTitle("Polar angle (rad)");
125 polar_angles_histogram->SetYTitle("Occurrences");
126 polar_angles_histogram->Fit(polar_fit, "Q");
127
128 std::cout << "\nPOLAR FIT" << '\n'
129           << HEIGHT_LABEL << ": " << polar_fit->GetParameter(0) << " +- "
130           << polar_fit->GetParError(0) << '\n'
131           << "Chi square/NDF: "
132           << polar_fit->GetChisquare() / polar_fit->GetNDF() << '\n'
133           << "Probability: " << polar_fit->GetProb() << '\n';
134
135 // Add and fit momentum histogram
136 TF1* momentum_fit = new TF1("momentum_fit", exp, 0., 9., 2);
137 momentum_fit->SetParameter(0, 90000);
138 momentum_fit->SetParName(0, HEIGHT_LABEL);
139 momentum_fit->SetParameter(1, 1.);
140 momentum_fit->SetParName(1, MEAN_LABEL);
141
142 particles_canvas->cd(4);
143 auto momentum_histogram = histo_array[3];
144 momentum_histogram->SetXTitle("Momentum (GeV)");
145 momentum_histogram->SetYTitle("Occurrences");
146 momentum_histogram->Fit(momentum_fit, "Q");
147
148 std::cout << "\nMOMENTUM FIT" << '\n'
149           << HEIGHT_LABEL << ": " << momentum_fit->GetParameter(0) << " +- "
150           << momentum_fit->GetParError(0) << '\n'
151           << MEAN_LABEL << ": " << momentum_fit->GetParameter(1) << " +- "
152           << momentum_fit->GetParError(1) << '\n'
153           << "Chi square/NDF: "
154           << momentum_fit->GetChisquare() / momentum_fit->GetNDF() << '\n'
155           << "Probability: " << momentum_fit->GetProb() << '\n';
156
157 // Create second canvas, with invariant mass
158 TCanvas* inv_mass_canvas = new TCanvas();
159 inv_mass_canvas->Divide(2, 2);
160 inv_mass_canvas->cd(1);
161
162 // Add first histogram directly from the generation
163 TF1* k_star_fit = new TF1("k_star_fit", gauss, H_LOW, H_HIGH, 3);
164 k_star_fit->SetParameter(0, 500);
165 k_star_fit->SetParName(0, HEIGHT_LABEL);
166 k_star_fit->SetParameter(1, 0.9);
167 k_star_fit->SetParName(1, MEAN_LABEL);
168 k_star_fit->SetParameter(2, 0.05);
169 k_star_fit->SetParName(2, STDDEV_LABEL);
170 auto invm_decayed_h = histo_array[11];
171 // Rebin in order to have wider bins
172 invm_decayed_h->Rebin(5);
173 invm_decayed_h->SetAxisRange(H_LOW, H_HIGH);
174 invm_decayed_h->SetTitle("Decay products");
175 invm_decayed_h->SetXTitle("Invariant mass (GeV)");
176 invm_decayed_h->SetYTitle("Occurrences");
177 invm_decayed_h->Fit(k_star_fit, "Q");
178
179 // Create second histogram. Find the signal by subtracting same charge
180 // particles from opposite charge particles
181 auto invm_opposite_charge_h = histo_array[7];
182 invm_opposite_charge_h->Rebin(10);
183 auto invm_same_charge_h = histo_array[8];
184 invm_same_charge_h->Rebin(10);
185 TH1F* invm_subtraction_all = new TH1F(*(TH1F*)invm_opposite_charge_h);
186 // Cosmetics
187 invm_subtraction_all->SetTitle("Opposite charge - same charge");
188 invm_subtraction_all->SetName("invm_subtraction_all");
189 invm_subtraction_all->SetXTitle("Invariant mass (GeV)");
190 invm_subtraction_all->SetYTitle("Occurrences");
191 invm_subtraction_all->Add(invm_opposite_charge_h, invm_same_charge_h, 1, -1);
192 invm_subtraction_all->SetEntries(invm_opposite_charge_h->GetEntries());
193 invm_subtraction_all->SetAxisRange(H_LOW, H_HIGH);
194
195 TF1* invm_all_fit = new TF1("invm_all_fit", gauss, H_LOW, H_HIGH, 3);
196 invm_all_fit->SetParameter(0, 7.996);
```

```
197   invm_all_fit->SetParName(0, HEIGHT_LABEL);
198   invm_all_fit->SetParameter(1, 0.8919);
199   invm_all_fit->SetParName(1, MEAN_LABEL);
200   invm_all_fit->SetParameter(2, 0.04989);
201   invm_all_fit->SetParName(2, STDDEV_LABEL);
202
203   inv_mass_canvas->cd(2);
204   invm_subtraction_all->Fit(invm_all_fit, "Q");
205
206   std::cout << "\nINVARIANT MASS BETWEEN ALL PARTICLES (OPPOSITE CHARGE - SAME "
207                "CHARGE) FIT"
208             << '\n'
209             << HEIGHT_LABEL << ": " << invm_all_fit->GetParameter(0) << " +- "
210             << invm_all_fit->GetParError(0) << '\n'
211             << MEAN_LABEL << ": " << invm_all_fit->GetParameter(1) << " +- "
212             << invm_all_fit->GetParError(1) << '\n'
213             << STDDEV_LABEL << ": " << invm_all_fit->GetParameter(2) << " +- "
214             << invm_all_fit->GetParError(2) << '\n'
215             << "Chi square/NDF: "
216             << invm_all_fit->GetChisquare() / invm_all_fit->GetNDF() << '\n'
217             << "Probability: " << invm_all_fit->GetProb() << '\n'
218             << "K* mass: " << invm_all_fit->GetParameter(1) << " +- "
219             << invm_all_fit->GetParError(1) << '\n'
220             << "K* width: " << invm_all_fit->GetParameter(2) << " +- "
221             << invm_all_fit->GetParError(2) << '\n';
222
223   // Create the third histogram. Find the signal by subtracting kaon & pion with
224   // same charge from kaon & pion with opposite charge
225   auto invm_pion_kaon_opposite_h = histo_array[9];
226   invm_pion_kaon_opposite_h->Rebin(10);
227   auto invm_pion_kaon_same_h = histo_array[10];
228   invm_pion_kaon_same_h->Rebin(10);
229   TH1F* invm_subtraction_pion_kaon =
230       new TH1F(*(TH1F*)invm_pion_kaon_opposite_h);
231   // Cosmetics
232   invm_subtraction_pion_kaon->SetTitle(
233       "Opposite charge - same charge (kaon & pion)");
234   invm_subtraction_pion_kaon->SetName("invm_subtraction_pion_kaon");
235   invm_subtraction_pion_kaon->SetXTitle("Invariant mass (GeV)");
236   invm_subtraction_pion_kaon->SetYTitle("Occurrences");
237   invm_subtraction_pion_kaon->Add(invm_pion_kaon_opposite_h,
238                                   invm_pion_kaon_same_h, 1, -1);
239   invm_subtraction_pion_kaon->SetEntries(
240       invm_pion_kaon_opposite_h->GetEntries());
241   invm_subtraction_pion_kaon->SetAxisRange(H_LOW, H_HIGH);
242
243   TF1* invm_pion_kaon_fit =
244       new TF1("invm_pion_kaon_fit", gauss, H_LOW, H_HIGH, 3);
245   invm_pion_kaon_fit->SetParameter(0, 7.996);
246   invm_pion_kaon_fit->SetParName(0, HEIGHT_LABEL);
247   invm_pion_kaon_fit->SetParameter(1, 0.8919);
248   invm_pion_kaon_fit->SetParName(1, MEAN_LABEL);
249   invm_pion_kaon_fit->SetParameter(2, 0.04989);
250   invm_pion_kaon_fit->SetParName(2, STDDEV_LABEL);
251
252   inv_mass_canvas->cd(3);
253   invm_subtraction_pion_kaon->Fit(invm_pion_kaon_fit, "Q");
254
255   std::cout << "\nINVARIANT MASS BETWEEN KAON AND PION (OPPOSITE CHARGE - SAME "
256                "CHARGE) FIT"
257             << '\n'
258             << HEIGHT_LABEL << ": " << invm_pion_kaon_fit->GetParameter(0)
259             << " +- " << invm_pion_kaon_fit->GetParError(0) << '\n'
260             << MEAN_LABEL << ": " << invm_pion_kaon_fit->GetParameter(1)
261             << " +- " << invm_pion_kaon_fit->GetParError(1) << '\n'
262             << STDDEV_LABEL << ": " << invm_pion_kaon_fit->GetParameter(2)
263             << " +- " << invm_pion_kaon_fit->GetParError(2) << '\n'
264             << "Chi square/NDF: "
265             << invm_pion_kaon_fit->GetChisquare() / invm_pion_kaon_fit->GetNDF()
266             << '\n'
267             << "Probability: " << invm_pion_kaon_fit->GetProb() << '\n'
268             << "K* mass: " << invm_pion_kaon_fit->GetParameter(1) << " +- "
269             << invm_pion_kaon_fit->GetParError(1) << '\n'
```

```
270                 << "K* width: " << invm_pion_kaon_fit->GetParameter(2) << " +- "
271                 << invm_pion_kaon_fit->GetParError(2) << '\n';
272 }
```