Relazione ROOT 2025

Samuele Bellini, Francesco Caprara, Gabriele Introna, Niccolò Lugli

Dicembre 2024

1 Introduzione

Lo scopo di questo esperimento è studiare, attraverso la simulazione di un campione di particelle di vario tipo, le proprietà di una particella ignota e instabile che decade troppo velocemente per poter essere osservata direttamente. Poiché la particella iniziale è neutra, i prodotti del decadimento avranno carica discorde; inoltre, la massa invariante segue una distribuzione gaussiana con media pari alla massa della particella madre e deviazione standard pari alla sua larghezza di risonanza. Di conseguenza, si misurano le masse invarianti dal campione di particelle che contiene anche quelle derivanti dal decadimento separando per coppie di carica concorde e discorde, e si isola l'effetto delle sole particelle figlie rimuovendo il rumore di fondo dai dati. Per ridurre il fondo più efficientemente si può ripetere l'analisi considerando il campione composto dalle sole particelle ottenibili teoricamente dal decadimento.

2 Struttura del Codice

Il codice che gestisce le particelle è stato organizzato creando tre classi: ParticleType, ResonanceType e Particle. Come il nome suggerisce, le prime due vogliono rappresentare dei tipi di particelle (protone, antiprotone, kaone, pione, ...), mentre la terza rappresenta una particella di per sé. ParticleType descrive un tipo di particella che possiede una massa e una carica, e alla quale viene assegnato un nome. La classe, inoltre, espone anche due metodi virtuali, GetWidth e Print, che possono essere sovrascritti dalle classi figlie per implementare la larghezza di risonanza di una particella o per modificare la stampa a schermo delle proprietà della particella. ResonanceType fa uso di questa possibilità ereditando pubblicamente da ParticleType, in quanto vuole rappresentare un tipo di particella che può decadere e che, di conseguenza, oltre alle proprietà di una particella base, necessita anche della descrizione della sua larghezza di risonanza. La classe Particle, infine, funge da rappresentazione di una particella di un dato tipo con una quantità di moto. Sapendo che nel programma il numero di tipi di particelle sarà minuscolo rispetto al numero di particelle che verranno create si è preferito utilizzare, rispetto all'ereditarietà, che avrebbe portato a una ripetizione degli stessi dati (nome, massa, carica e larghezza di risonanza)

fra molte particelle, uno std::vector statico di puntatori a ParticleType (che di conseguenza, per il polimorfismo dinamico, possono puntare anche a ResonanceType). Quest'ultimo racchiude tutti i tipi di particelle che sono presenti nella simulazione, i quali vengono aggiunti a run-time: si è preferito questo approccio per accomodare anche l'eventualità in cui non si dovesse conoscere il numero di tipi di particelle a compile-time, come avverrebbe se questi volessero essere specificati in un file invece che nel codice. In questo modo ad ogni particella saranno associati solo tre float, per descrivere la quantità di moto, e un indicatore del tipo di particella, per il quale è stato scelto un int, che funge da indice nell'std::vector di tipi di particelle.

3 Generazione

Il codice prevede la generazione di 10^5 eventi, ciascuno dei quali considera molte particelle suddivise in un numero limitato di tipi. Nel nostro caso, in ogni evento vengono generate, una alla volta, 100 particelle che possono appartenere a sette tipi differenti, ognuno dei quali ha una fissa percentuale di probabilità di essere generato: le proporzioni prevedono 40% di pioni positivi, 40% di pioni negativi, 5% di kaoni positivi, 5% di kaoni negativi, 4.5% di protoni, 4.5% di antiprotoni e 1% di una particella instabile che chiamiamo K*, che decade in una coppia di carica discorde composta da un pione e un kaone, con uguale probabilità di ottenere coppie $\pi + K-$ o $\pi - K+$. Ad ogni particella generata sono associate sia proprietà di base (nome, massa, carica, larghezza di risonanza) sia proprietà cinematiche, cioè una quantità di moto sulle tre dimensioni, che può variare da particella a particella e da evento a evento. Per generare quest'impulso in maniera casuale abbiamo utilizzato i motodi Monte Carlo implementati da ROOT per generare una distribuzione esponenziale con media 1 per il modulo, due distribuzioni uniformi, rispettivamente tra 0 e 2π e tra 0 e π, per gli angoli phi (azimutale) e theta (polare), che definiscono la direzione dell'impulso nello spazio, e abbiamo poi convertito questi dati da coordinate sferiche a cartesiane, ottenendo le tre componenti Px, Py e Pz della quantità di moto.

4 Analisi

L'istogramma della distribuzione dei tipi delle particelle generate segue coerentemente la distribuzione a tratti costanti che ci si aspetta: la probabilità della creazione di ciascun tipo corrisponde alla proporzione in cui è effettivamente osservato (Tab. 1). La distribuzione degli impulsi è coerente con un'esponenziale, e al fit corrisponde correttamente un $\tilde{\chi}^2 \simeq 1$ (Tab. 2). Gli angoli polari e azimutali sono distribuiti uniformemente nei domini previsti, come confermano i fit a cui corrispondono nuovamente dei $\tilde{\chi}^2 \simeq 1$ (Tab. 2). Questi quattro grafici sono riportati in Fig. 1.

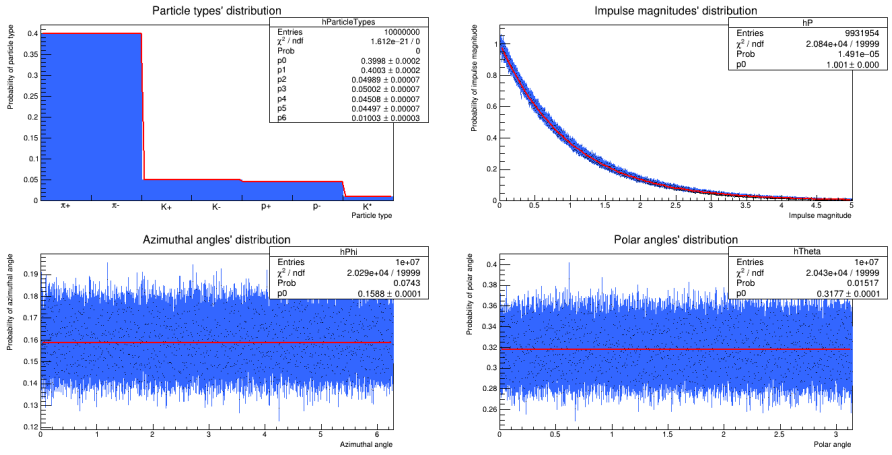

Figure 1: In alto a sx la distribuzione dei tipi di particelle generati, in alto a dx la distribuzione del modulo dell'impulso, in basso le distribuzioni degli angoli azimutale (ϕ) e polare (θ).

Per la distribuzione della massa invariante relativa alle coppie di figlie di K* si osserva, come ci si aspetta, una gaussiana con media e deviazione standard pari rispettivamente a massa e lunghezza di risonanza della K* (Tab. 3). Essendo la K* molto instabile e rara, per estrarre il suo segnale è occorso seguire un approccio per sottrazione: a partire dalla distribuzione della massa invariante delle particelle generate, rimuovendo il rumore di fondo si riesce a osservare il segnale della particella di risonanza. Ovvero, dalla distribuzione della massa invariante delle particelle di segno discorde, che comprendono le combinazioni accidentali (fondo) e quelle effettivamente generate a partire dal decadimento di K*, è stata rimossa quella della massa invariante delle particelle di segno concorde, che possono essere solo fondo (K* decade in $\pi+$ e K- o in $\pi-$ e K+). Per un'ulteriore distinzione più efficace è stata eseguita la stessa operazione, ma prendendo in considerazione solo i pioni π e i kaoni K, poiché K* può decadere solo in questi, e quindi riducendo il rumore di fondo (i protoni P sono circa il 9% delle particelle generate). Le distribuzioni risultanti, analogamente alla precedente (tutte in Fig. 2), corrispondono a delle gaussiane la cui media risulta essere la massa della K*, e la cui sigma la larghezza di risonanza: si è quindi riuscito ad estrarre il segnale dalle distribuzioni delle masse invarianti (Tab. 3).

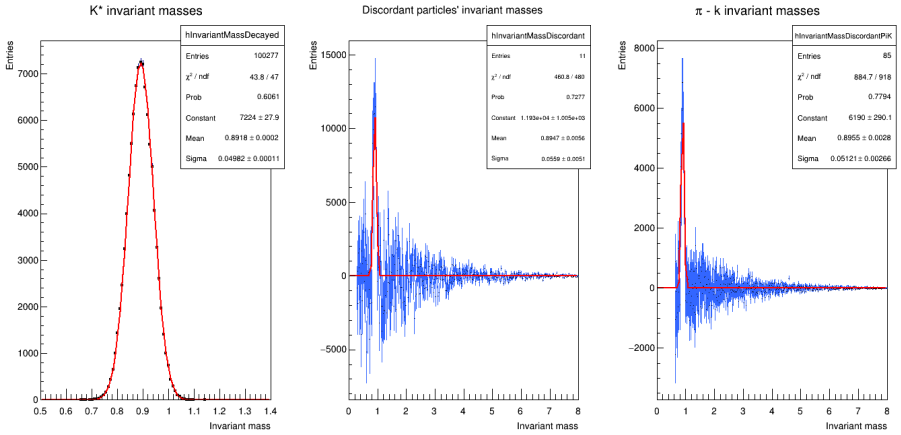

Figure 2: A sinistra la distribuzione delle masse invarianti solo fra particelle figlie, al centro la differenza degli istogrammi di masse invarianti fra particelle discordi e concordi, a destra come al centro ma solo considerando coppie πK

Specie	Occorrenze osservate (10^6)	Occorrenze attese (10^6)
π^+	3.998 ± 0.002	4.0000 ± 0.0015
π^-	4.003 ± 0.002	4.0000 ± 0.0015
K^+	0.4989 ± 0.0007	0.5000 ± 0.0007
K^-	0.5002 ± 0.0007	0.5000 ± 0.0007
p^+	0.4508 ± 0.0007	0.4500 ± 0.0007
p^-	0.4497 ± 0.0007	0.4500 ± 0.0007
K^*	0.1003 ± 0.0003	0.1000 ± 0.0003

Table 1: Tabella delle occorrenze osservate e attese dei diversi tipi di particelle.

Distribuzione	Parametro del fit	χ^2	DOF	$\tilde{\chi}^2$
Fit a distribuzione angolo ϕ (azi)	0.15883 ± 0.00005	20288.6	19999	1.01
Fit a distribuzione angolo θ (pol)	0.31766 ± 0.00010	20434.6	19999	1.02
Fit a distribuzione modulo impulso (expo)	1.0007 ± 0.0002	20844.9	19999	1.04

Table 2: Risultati dei fit per diverse distribuzioni.

Distribuzione e fit	μ	σ	Ampiezza	$\tilde{\chi}^2$
Massa invariante vere K^*	0.89185 ± 0.00016	0.04982 ± 0.00011	7220 ± 30	0.932
Massa invariante ottenuta da differenza delle combinazioni di carica discorde e concorde	0.895 ± 0.006	0.056 ± 0.005	$(11.9 \pm 1.0) \cdot 10^3$	0.960
Massa invariante ottenuta da differenza delle combinazioni di carica discorde e concorde πK	0.051 ± 0.003	0.051 ± 0.003	$(6.2 \pm 0.3) \cdot 10^3$	0.964

Table 3: Parametri di fit per le masse invarianti in diverse combinazioni.

Selected files

11 printable files

ParticleType.hpp
ParticleType.cpp
ResonanceType.hpp
ResonanceType.cpp
Particle.hpp
Particle.cpp
compile.C
ParticleGenerator.cpp
Analysis.cpp
testParticleType.cpp
testParticle.cpp

ParticleType.hpp

```cpp
#ifndef PARTICLE_TYPE_HPP
#define PARTICLE_TYPE_HPP

namespace kape {
class ParticleType
{
 public:
  explicit ParticleType(const char* name, double mass, int charge);
  explicit ParticleType();
  virtual ~ParticleType();

  const char* GetName() const;
  double GetMass() const;
  int GetCharge() const;

  virtual double GetWidth() const;
  virtual void Print() const;

 private:
  const char* fName;
  const double fMass;
  const int fCharge;
};
} // namespace kape

#endif
```

ParticleType.cpp

```cpp
#include "ParticleType.hpp"
#include <iostream>
#include <stdexcept>

namespace kape {
ParticleType::ParticleType(const char* name, double mass, int charge)
    : fName{name}
```

```
    , fMass{mass}
    , fCharge{charge}
{
  if (mass <= 0.) {
    throw std::invalid_argument{"mass can't be negative or null"};
  }

  if (name == nullptr) {
    throw std::invalid_argument{"name can't point to nullptr"};
  }
}

ParticleType::ParticleType()
    : ParticleType("", 1, 0)
{}

ParticleType::~ParticleType()
{}

const char* ParticleType::GetName() const
{
  return fName;
}

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

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

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


void ParticleType::Print() const
{
  std::cout << "Name:\t" << fName << '\n';
  std::cout << "Mass:\t" << fMass << '\n';
  std::cout << "Charge:\t" << fCharge << '\n';
}
} // namespace kape
```

ResonanceType.hpp

```
#ifndef RESONANCE_TYPE_HPP
#define RESONANCE_TYPE_HPP
```

```
#include "ParticleType.hpp"

namespace kape {
class ResonanceType : public ParticleType
{
 public:
  explicit ResonanceType(const char* name, double mass, int charge,
                         double width);
  explicit ResonanceType();

  virtual double GetWidth() const override;
  void Print() const override;

 private:
  const double fWidth;
};
} // namespace kape

#endif
```

ResonanceType.cpp

```
#include "ResonanceType.hpp"
#include <iostream>
#include <stdexcept>

namespace kape {
ResonanceType::ResonanceType(const char* name, double mass, int charge,
                             double width)
    : ParticleType(name, mass, charge)
    , fWidth{width}
{
  if (width <= 0) {
    throw std::invalid_argument{"width can't be negative or null"};
  }
}

ResonanceType::ResonanceType()
    : ParticleType()
    , fWidth{}
{}

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

void ResonanceType::Print() const
{
  ParticleType::Print();
  std::cout << "Width:\t" << fWidth << '\n';
}
```

```
} // namespace kape

```

Particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include "ParticleType.hpp"
#include "ResonanceType.hpp"
#include <vector>

namespace kape {
class Particle
{
 public:
  static int GetNParticleType();
  static void AddParticleType(const char* name, double mass, int charge,
                              double width = 0);
  static void PrintParticleType();

  Particle(const char* name = DEFAULT_NAME, double px = 0., double py = 0.,
           double pz = 0.);
  int Decay2body(Particle& dau1, Particle& dau2) const;
  int GetIndex() const;
  double GetPx() const;
  double GetPy() const;
  double GetPz() const;
  double GetMass() const;
  double GetEnergy() const;
  double GetCharge() const;
  void SetIndex(int index);
  void SetIndex(const char* name);
  void SetP(double px, double py, double pz);
  double InvMass(Particle const& p) const;

  void Print() const;

 private:
  static inline const char* DEFAULT_NAME{"DEFAULT_NAME"};
  static inline std::vector<ParticleType*> fParticleType{};
  //returns the index of the first particle type named "name"
  //if not found it returns the number of particle types
  static int FindParticle(const char* name);
  void Boost(double bx, double by, double bz);

  int fIndex;
  double fPx;
  double fPy;
  double fPz;
};

```

```
48 } // namespace kape
49
50 #endif
```

Particle.cpp

```
 1 #include "Particle.hpp"
 2 #include <cmath>
 3 #include <cstdlib> //for RAND_MAX
 4 #include <cstring> //for strcmp
 5 #include <iostream>
 6 #include <stdexcept>
 7
 8 namespace kape {
 9 int Particle::GetNParticleType()
10 {
11   return static_cast<int>(fParticleType.size());
12 }
13
14 Particle::Particle(const char* name, double px, double py, double pz)
15     : fPx{px}
16     , fPy{py}
17     , fPz{pz}
18 {
19   fIndex = FindParticle(name);
20   // not found
21   if (fIndex == static_cast<int>(fParticleType.size())
22       && name != DEFAULT_NAME) {
23     std::cout << name << " is not a defined type of particle\n";
24     throw std::runtime_error{"it is not a defined type of particle. Check "
25                              "terminal output for the name."};
26   }
27 }
28
29 int Particle::Decay2body(Particle& dau1, Particle& dau2) const
30 {
31   if (GetMass() == 0.0) {
32     printf("Decayment cannot be preformed if mass is zero\n");
33     return 1;
34   }
35
36   double massMot  = GetMass();
37   double massDau1 = dau1.GetMass();
38   double massDau2 = dau2.GetMass();
39
40   if (fIndex > -1) { // add width effect
41
42     // gaussian random numbers
43
44     float x1, x2, w, y1;
45
46     double invnum = 1. / RAND_MAX;
```

```
47      do {
48        x1 = 2.0 * rand() * invnum - 1.0;
49        x2 = 2.0 * rand() * invnum - 1.0;
50        w  = x1 * x1 + x2 * x2;
51      } while (w >= 1.0);
52
53      w  = sqrt((-2.0 * log(w)) / w);
54      y1 = x1 * w;
55
56      massMot += fParticleType[fIndex]->GetWidth() * y1;
57    }
58
59    if (massMot < massDau1 + massDau2) {
60      printf("Decayment cannot be preformed because mass is too low in this "
61             "channel\n");
62      return 2;
63    }
64
65    double pout =
66        sqrt(
67            (massMot * massMot - (massDau1 + massDau2) * (massDau1 + massDau2))
68            * (massMot * massMot - (massDau1 - massDau2) * (massDau1 - massDau2)))
69        / massMot * 0.5;
70
71    double norm = 2 * M_PI / RAND_MAX;
72
73    double phi   = rand() * norm;
74    double theta = rand() * norm * 0.5 - M_PI / 2.;
75    dau1.SetP(pout * sin(theta) * cos(phi), pout * sin(theta) * sin(phi),
76              pout * cos(theta));
77    dau2.SetP(-pout * sin(theta) * cos(phi), -pout * sin(theta) * sin(phi),
78              -pout * cos(theta));
79
80    double energy = sqrt(fPx * fPx + fPy * fPy + fPz * fPz + massMot * massMot);
81
82    double bx = fPx / energy;
83    double by = fPy / energy;
84    double bz = fPz / energy;
85
86    dau1.Boost(bx, by, bz);
87    dau2.Boost(bx, by, bz);
88
89    return 0;
90  }
91
92  void Particle::AddParticleType(const char* name, double mass, int charge,
93                                 double width)
94  {
95    int index = FindParticle(name);
96    if (index == GetNParticleType()) // it's a new Particle Type
97    {
98      if (width == 0.) { // it's a ParticleType
99        fParticleType.push_back(new ParticleType(name, mass, charge));
```

```
100     } else {
101       fParticleType.push_back(new ResonanceType(name, mass, charge, width));
102     }
103   } else // we're updating a Particle Type
104   {
105     delete fParticleType[index];
106     if (width == 0.) { // it's a ParticleType
107       fParticleType[index] = new ParticleType(name, mass, charge);
108     } else {
109       fParticleType[index] = new ResonanceType(name, mass, charge, width);
110     }
111   }
112 }
113
114 void Particle::PrintParticleType()
115 {
116   std::cout << "Particle types:\n";
117   for (auto const& p : fParticleType) {
118     p->Print();
119   }
120 }
121
122 int Particle::GetIndex() const
123 {
124   return fIndex;
125 }
126 double Particle::GetPx() const
127 {
128   return fPx;
129 }
130 double Particle::GetPy() const
131 {
132   return fPy;
133 }
134 double Particle::GetPz() const
135 {
136   return fPz;
137 }
138
139 double Particle::GetMass() const
140 {
141   return fParticleType[fIndex]->GetMass();
142 }
143
144 double Particle::GetEnergy() const
145 {
146   double m{fParticleType[fIndex]->GetMass()};
147   return std::sqrt(m * m + fPx * fPx + fPy * fPy + fPz * fPz);
148 }
149
150 double Particle::GetCharge() const
151 {
152   return fParticleType[fIndex]->GetCharge();
```

```
153 }
154
155 void Particle::SetIndex(int index)
156 {
157   if (index >= GetNParticleType() || index < 0) {
158     std::cout << "it is not a defined type of particle\n";
159     throw std::runtime_error{"it is not a defined type of particle."};
160   }
161
162   fIndex = index;
163 }
164
165 void Particle::SetIndex(const char* name)
166 {
167   SetIndex(FindParticle(name));
168 }
169
170 void Particle::SetP(double px, double py, double pz)
171 {
172   fPx = px;
173   fPy = py;
174   fPz = pz;
175 }
176
177 double Particle::InvMass(Particle const& p) const
178 {
179   return sqrt(std::pow(GetEnergy() + p.GetEnergy(), 2)
180               - (std::pow(fPx + p.fPx, 2) + std::pow(fPy + p.fPy, 2)
181                  + std::pow(fPz + p.fPz, 2)));
182 }
183
184 void Particle::Print() const
185 {
186   std::cout << "Index: " << fIndex << '\n';
187   std::cout << "Name: " << fParticleType[fIndex]->GetName() << '\n';
188   std::cout << "P = (" << fPx << ", " << fPy << ", " << fPz << ") " << '\n';
189 }
190
191 int Particle::FindParticle(const char* name)
192 {
193   int i{0};
194   for (; i != static_cast<int>(fParticleType.size()); ++i) {
195     if (std::strcmp(fParticleType[i]->GetName(), name) == 0) {
196       break;
197     }
198   }
199   return i;
200 }
201
202 void Particle::Boost(double bx, double by, double bz)
203 {
204   double energy = GetEnergy();
205
```

```
206   // Boost this Lorentz vector
207   double b2     = bx * bx + by * by + bz * bz;
208   double gamma  = 1.0 / sqrt(1.0 - b2);
209   double bp     = bx * fPx + by * fPy + bz * fPz;
210   double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;
211
212   fPx += gamma2 * bp * bx + gamma * bx * energy;
213   fPy += gamma2 * bp * by + gamma * by * energy;
214   fPz += gamma2 * bp * bz + gamma * bz * energy;
215 }
216 } // namespace kape
```

compile.C

```
1 void compile(){
2     gROOT->LoadMacro("ParticleType.cpp+");
3     gROOT->LoadMacro("ResonanceType.cpp+");
4     gROOT->LoadMacro("Particle.cpp+");
5     gROOT->LoadMacro("ParticleGenerator.cpp+");
6 }
```

ParticleGenerator.cpp

```
 1 #include "Particle.hpp"
 2 #include "TFile.h"
 3 #include "TH1.h"
 4 #include "TRandom.h"
 5 #include "TRandom3.h"
 6 #include <array>
 7 #include <cmath> //for M_PI
 8 #include <iostream>
 9
10 enum ParticlesIndexes
11 {
12   PI_PLUS = 0,
13   PI_MINUS,
14   K_PLUS,
15   K_MINUS,
16   P_PLUS,
17   P_MINUS,
18   K_STAR
19 };
20
21 void RunSimulation()
22 {
23   kape::Particle::AddParticleType("pi+", 0.13957, +1);      // pione +
24   kape::Particle::AddParticleType("pi-", 0.13957, -1);      // pione -
25   kape::Particle::AddParticleType("K+", 0.49367, +1);       // kaone +
26   kape::Particle::AddParticleType("K-", 0.49367, -1);       // kaone -
27   kape::Particle::AddParticleType("p+", 0.93827, +1);       // protone
28   kape::Particle::AddParticleType("p-", 0.93827, -1);       // antiprotone
29   kape::Particle::AddParticleType("K*", 0.89166, 0, 0.050); // K*
```

```
30
31  // for a longer period of the random number generator
32  delete gRandom;
33  gRandom = new TRandom3();
34  gRandom->SetSeed(
35      136279841); // it's the exponent of the biggest mersenne
36                  // prime found to this day :D (from GIMPS on 21/10/2024)
37
38  // chose 300 because all 100 particles could (in principle) be a k* and decay
39  // in two more particles
40  std::array<kape::Particle, 300> eventParticles;
41
42  // creating histograms
43  TH1F* hParticleTypes =
44      new TH1F("hParticleTypes", "Generated particle types", 7, -0.5, 6.5);
45  TH1F* hPhi   = new TH1F("hPhi", "Generated phi angles", 1e5, 0., 2. * M_PI);
46  TH1F* hTheta = new TH1F("hTheta", "Generated theta angles", 1e5, 0., M_PI);
47  TH1F* hP     = new TH1F("hP", "Generated p magnitudes", 1e5, 0., 5.);
48  TH1F* hPTrasverse =
49      new TH1F("hPTrasverse", "Generated p trasverses", 1e5, 0., 5.);
50  TH1F* hEnergy =
51      new TH1F("hEnergy", "Generated particle energies", 1000, 0., 6.);
52  TH1F* hInvariantMass =
53      new TH1F("hInvariantMass", "Generated invariant masses", 1e5, 0., 8.);
54  TH1F* hInvariantMassDiscordant =
55      new TH1F("hInvariantMassDiscordant",
56               "Generated discordant invariant masses", 1e4, 0., 8.);
57  TH1F* hInvariantMassConcordant =
58      new TH1F("hInvariantMassConcordant",
59               "Generated concordant invariant masses", 1e4, 0., 8.);
60  TH1F* hInvariantMassDiscordantPiK =
61      new TH1F("hInvariantMassDiscordantPiK",
62               "Generated discordant invariant masses pi/K", 1e4, 0., 8.);
63  TH1F* hInvariantMassConcordantPiK =
64      new TH1F("hInvariantMassConcordantPiK",
65               "Generated concordant invariant masses pi/K", 1e4, 0., 8.);
66  TH1F* hInvariantMassDecayed =
67      new TH1F("hInvariantMassDecayed", "Generated decayed invariant masses",
68               100, 0.5, 1.4);
69
70  // sumw2 for correct errors
71  hParticleTypes->Sumw2();
72  hPhi->Sumw2();
73  hTheta->Sumw2();
74  hP->Sumw2();
75  hPTrasverse->Sumw2();
76  hEnergy->Sumw2();
77  hInvariantMass->Sumw2();
78  hInvariantMassDiscordant->Sumw2();
79  hInvariantMassConcordant->Sumw2();
80  hInvariantMassDiscordantPiK->Sumw2();
81  hInvariantMassConcordantPiK->Sumw2();
82  hInvariantMassDecayed->Sumw2();
```

```
83
84    // generating 10^5 events
85    for (int eventIndex = 0; eventIndex != 1e5; ++eventIndex) {
86      // index for the next free space where decayed particles can be placed
87      int arrayEnd = 100;
88      // generating the 100 particles of each event
89      for (int arrayIndex = 0; arrayIndex < 100; ++arrayIndex) {
90        Double_t phi   = gRandom->Uniform(0., 2. * M_PI);
91        Double_t theta = gRandom->Uniform(0., M_PI);
92        Double_t p     = gRandom->Exp(1.);
93
94        eventParticles[arrayIndex].SetP(p * std::sin(theta) * std::cos(phi),
95                                        p * std::sin(theta) * std::sin(phi),
96                                        p * std::cos(theta));
97
98        // choose particle type following proportions
99        Double_t randomChoice = gRandom->Rndm();
100       if (randomChoice < 0.40) {
101         // pi+
102         eventParticles[arrayIndex].SetIndex(PI_PLUS);
103       } else if (randomChoice < 0.80) {
104         // pi-
105         eventParticles[arrayIndex].SetIndex(PI_MINUS);
106       } else if (randomChoice < 0.85) {
107         // k+
108         eventParticles[arrayIndex].SetIndex(K_PLUS);
109       } else if (randomChoice < 0.90) {
110         // k-
111         eventParticles[arrayIndex].SetIndex(K_MINUS);
112       } else if (randomChoice < 0.945) {
113         // p+
114         eventParticles[arrayIndex].SetIndex(P_PLUS);
115       } else if (randomChoice < 0.99) {
116         // p-
117         eventParticles[arrayIndex].SetIndex(P_MINUS);
118       } else {
119         // k* -> decays
120         eventParticles[arrayIndex].SetIndex(K_STAR);
121
122         // choose decayed particle types randomly
123         if (gRandom->Rndm() <= 0.5) {
124           eventParticles[arrayEnd].SetIndex(PI_PLUS);
125           eventParticles[arrayEnd + 1].SetIndex(K_MINUS);
126         } else {
127           eventParticles[arrayEnd].SetIndex(PI_MINUS);
128           eventParticles[arrayEnd + 1].SetIndex(K_PLUS);
129         }
130
131         int error = eventParticles[arrayIndex].Decay2body(
132             eventParticles[arrayEnd], eventParticles[arrayEnd + 1]);
133         if (error != 0) {
134           std::cout << "decayed to body failed with error " << error << '\n';
135           throw std::runtime_error{
```

```
136               "decayed to body failed, check terminal for more info.\n"};
137         }
138
139         // the next free space is two places after the last free space (2
140         // daughters)
141         arrayEnd += 2;
142       }
143
144       // filling histograms
145       hParticleTypes->Fill(eventParticles[arrayIndex].GetIndex());
146       hPhi->Fill(phi);
147       hTheta->Fill(theta);
148       hP->Fill(p);
149       hPTrasverse->Fill(
150           std::sqrt(std::pow(eventParticles[arrayIndex].GetPx(), 2)
151                     + std::pow(eventParticles[arrayIndex].GetPy(), 2)));
152       hEnergy->Fill(eventParticles[arrayIndex].GetEnergy());
153     }
154
155     // calculating invariant masses between all pairs of particles
156     for (int i = 0; i != arrayEnd - 1; ++i) {
157       if (eventParticles[i].GetIndex() == K_STAR) { // ignore k_star
158         continue;
159       }
160       for (int j = i + 1; j != arrayEnd; ++j) {
161         if (eventParticles[j].GetIndex() == K_STAR) { // ignore k_star
162           continue;
163         }
164
165         // calculate invariant mass of the pair
166         Double_t invMass = eventParticles[i].InvMass(eventParticles[j]);
167         hInvariantMass->Fill(invMass);
168
169         int i_type = eventParticles[i].GetIndex();
170         int j_type = eventParticles[j].GetIndex();
171
172         // discordant
173         if (eventParticles[i].GetCharge() * eventParticles[j].GetCharge() < 0) {
174           hInvariantMassDiscordant->Fill(invMass);
175
176           // PiK pair
177           if ((i_type == PI_PLUS && j_type == K_MINUS)
178               || (i_type == PI_MINUS && j_type == K_PLUS)
179               || (j_type == PI_PLUS && i_type == K_MINUS)
180               || (j_type == PI_MINUS && i_type == K_PLUS)) {
181             hInvariantMassDiscordantPiK->Fill(invMass);
182           }
183         } else { // concordant
184           hInvariantMassConcordant->Fill(invMass);
185
186           // PiK pair
187           if ((i_type == PI_PLUS && j_type == K_PLUS)
188               || (i_type == PI_MINUS && j_type == K_MINUS)
```

```
              || (j_type == PI_PLUS && i_type == K_PLUS)
              || (j_type == PI_MINUS && i_type == K_MINUS)) {
            hInvariantMassConcordantPiK->Fill(invMass);
          }
        }
      }
    }

    // filling the histogram with the invariant masses from only pairs of
    // decayed particles
    int i = 100;
    while (i < arrayEnd) {
      hInvariantMassDecayed->Fill(
          eventParticles[i].InvMass(eventParticles[i + 1]));
      i += 2;
    }
  }

  //save to file
  TFile* file = new TFile("histo.root", "RECREATE");

  hParticleTypes->Write();
  hPhi->Write();
  hTheta->Write();
  hP->Write();
  hPTrasverse->Write();
  hEnergy->Write();
  hInvariantMass->Write();
  hInvariantMassDiscordant->Write();
  hInvariantMassConcordant->Write();
  hInvariantMassDiscordantPiK->Write();
  hInvariantMassConcordantPiK->Write();
  hInvariantMassDecayed->Write();

  file->Close();
}
```

Analysis.cpp

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

std::string ExpectedWithError(Int_t nTot, Double_t probability)
{
  return std::string{
      std::to_string(nTot * probability) + " ± "
      + std::to_string(std::sqrt((1. - probability) * (nTot)*probability))};
```

```
14 }
15
16 enum ParticlesIndexesAnalysis
17 {
18   PI_PLUS = 0,
19   PI_MINUS,
20   K_PLUS,
21   K_MINUS,
22   P_PLUS,
23   P_MINUS,
24   K_STAR
25 };
26
27 void Analysis()
28 {
29   TFile* file = new TFile("histo.root");
30
31   // read the data from file
32   TH1F* hParticleTypes           = (TH1F*)file->Get("hParticleTypes");
33   TH1F* hPhi                     = (TH1F*)file->Get("hPhi");
34   TH1F* hTheta                   = (TH1F*)file->Get("hTheta");
35   TH1F* hP                       = (TH1F*)file->Get("hP");
36   TH1F* hPTrasverse              = (TH1F*)file->Get("hPTrasverse");
37   TH1F* hEnergy                  = (TH1F*)file->Get("hEnergy");
38   TH1F* hInvariantMass           = (TH1F*)file->Get("hInvariantMass");
39   TH1F* hInvariantMassDiscordant = (TH1F*)file->Get("hInvariantMassDiscordant");
40   TH1F* hInvariantMassConcordant = (TH1F*)file->Get("hInvariantMassConcordant");
41   TH1F* hInvariantMassDiscordantPiK =
42       (TH1F*)file->Get("hInvariantMassDiscordantPiK");
43   TH1F* hInvariantMassConcordantPiK =
44       (TH1F*)file->Get("hInvariantMassConcordantPiK");
45   TH1F* hInvariantMassDecayed = (TH1F*)file->Get("hInvariantMassDecayed");
46
47   // rebinning
48   hPhi->Rebin(5);
49   hTheta->Rebin(5);
50   hP->Rebin(5);
51   hInvariantMassDiscordant->Rebin(20);
52   hInvariantMassConcordant->Rebin(20);
53   hInvariantMassDiscordantPiK->Rebin(10);
54   hInvariantMassConcordantPiK->Rebin(10);
55
56   // check histo entries
57   if (hParticleTypes->GetEntries() != 1e7)
58     std::cout << "hParticleTypes has the wrong number of entries \n";
59   if (hPhi->GetEntries() != 1e7)
60     std::cout << "hPhi has the wrong number of entries \n";
61   if (hTheta->GetEntries() != 1e7)
62     std::cout << "hTheta has the wrong number of entries \n";
63   if (hP->GetEntries() != 1e7)
64     std::cout << "hP has the wrong number of entries \n";
65   if (hPTrasverse->GetEntries() != 1e7)
66     std::cout << "hPTrasverse has the wrong number of entries \n";
```

```
67    if (hEnergy->GetEntries() != 1e7)
68      std::cout << "hEnergy has the wrong number of entries \n";
69
70    // particle types proportions
71    std::cout
72        << "check that the expected number of particles generated for each type "
73           "corresponds to the number of generated particles of that type within "
74           "errors:\n";
75    std::cout << "|Particle Type\t| Expected\t\t\t\t| Generated\t\t|\n";
76    std::cout << "-------------------------------------------------------------"
77                 "------------------\n";
78    std::cout << "|pi+\t\t|" << ExpectedWithError(1e7, 0.400) << "\t\t|"
79              << hParticleTypes->GetBinContent(PI_PLUS + 1) << " ± "
80              << hParticleTypes->GetBinError(PI_PLUS + 1) << "\t|\n";
81    std::cout << "|pi-\t\t|" << ExpectedWithError(1e7, 0.400) << "\t\t|"
82              << hParticleTypes->GetBinContent(PI_MINUS + 1) << " ± "
83              << hParticleTypes->GetBinError(PI_MINUS + 1) << "\t|\n";
84    std::cout << "|K+ \t\t|" << ExpectedWithError(1e7, 0.050) << "\t\t|"
85              << hParticleTypes->GetBinContent(K_PLUS + 1) << " ± "
86              << hParticleTypes->GetBinError(K_PLUS + 1) << "\t|\n";
87    std::cout << "|K- \t\t|" << ExpectedWithError(1e7, 0.050) << "\t\t|"
88              << hParticleTypes->GetBinContent(K_MINUS + 1) << " ± "
89              << hParticleTypes->GetBinError(K_MINUS + 1) << "\t|\n";
90    std::cout << "|p+ \t\t|" << ExpectedWithError(1e7, 0.045) << "\t\t|"
91              << hParticleTypes->GetBinContent(P_PLUS + 1) << " ± "
92              << hParticleTypes->GetBinError(P_PLUS + 1) << "\t|\n";
93    std::cout << "|p- \t\t|" << ExpectedWithError(1e7, 0.045) << "\t\t|"
94              << hParticleTypes->GetBinContent(P_MINUS + 1) << " ± "
95              << hParticleTypes->GetBinError(P_MINUS + 1) << "\t|\n";
96    std::cout << "|K* \t\t|" << ExpectedWithError(1e7, 0.010) << "\t\t|"
97              << hParticleTypes->GetBinContent(K_STAR + 1) << " ± "
98              << hParticleTypes->GetBinError(K_STAR + 1) << "\t|\n";
99
100   // add all parameters to the output in the figures
101   gStyle->SetOptStat(11);
102   gStyle->SetOptFit(1111);
103
104   // Figure 1: particle types, p, phi,
105   // theta---------------------------------------
106   TCanvas* Figure1 = new TCanvas("Figure1", "Figure1", 0, 0, 800, 600);
107   Figure1->Divide(2, 2);
108
109   // particle types-------------------------------------------------------
110   Figure1->cd(1);
111
112   // normalize
113   hParticleTypes->Scale(1. / hParticleTypes->Integral(), "width");
114
115   // fitting
116   TF1* particleTypesDistr = new TF1("particleTypesDistr",
117                                     "[0]*(x<0.5) + "
118                                     "[1]*(0.5<x && x<1.5) +"
119                                     "[2]*(1.5<x && x<2.5) + "
```

```
                                    "[3]*(2.5<x && x<3.5) + "
                                    "[4]*(3.5<x && x<4.5) + "
                                    "[5]*(4.5<x && x<5.5) + "
                                    "[6]*(5.5<x && x<6.5)",
                                    -0.5, 6.5);

    hParticleTypes->Fit(particleTypesDistr);

    // fit output
    std::cout << "\nParticle Types Distribution Fit: \n"
                 "y = \t A if (x<0.5) \n"
                 "\t B if (0.5<x and x<1.5)\n "
                 "\t C if (1.5<x and x<2.5)\n "
                 "\t D if (2.5<x and x<3.5)\n "
                 "\t E if (3.5<x and x<4.5)\n "
                 "\t F if (4.5<x and x<5.5)\n "
                 "\t G if (5.5<x and x<6.5)\n";

    std::cout << "Parameter A: " << particleTypesDistr->GetParameter(0) << " ± "
              << particleTypesDistr->GetParError(0) << '\n';
    std::cout << "Parameter B: " << particleTypesDistr->GetParameter(1) << " ± "
              << particleTypesDistr->GetParError(1) << '\n';
    std::cout << "Parameter C: " << particleTypesDistr->GetParameter(2) << " ± "
              << particleTypesDistr->GetParError(2) << '\n';
    std::cout << "Parameter D: " << particleTypesDistr->GetParameter(3) << " ± "
              << particleTypesDistr->GetParError(3) << '\n';
    std::cout << "Parameter E: " << particleTypesDistr->GetParameter(4) << " ± "
              << particleTypesDistr->GetParError(4) << '\n';
    std::cout << "Parameter F: " << particleTypesDistr->GetParameter(5) << " ± "
              << particleTypesDistr->GetParError(5) << '\n';
    std::cout << "Parameter G: " << particleTypesDistr->GetParameter(6) << " ± "
              << particleTypesDistr->GetParError(6) << '\n';

    std::cout << "Reduced Chi Square: "
              << particleTypesDistr->GetChisquare() / particleTypesDistr->GetNDF()
              << "\n";
    std::cout << "Chi Square Probability: " << particleTypesDistr->GetProb()
              << "\n\n";

    // graphics
    hParticleTypes->SetTitle("Particle types' distribution");
    hParticleTypes->GetXaxis()->SetBinLabel(1, "#pi+");
    hParticleTypes->GetXaxis()->SetBinLabel(2, "#pi-");
    hParticleTypes->GetXaxis()->SetBinLabel(3, "K+");
    hParticleTypes->GetXaxis()->SetBinLabel(4, "K-");
    hParticleTypes->GetXaxis()->SetBinLabel(5, "p+");
    hParticleTypes->GetXaxis()->SetBinLabel(6, "p-");
    hParticleTypes->GetXaxis()->SetBinLabel(7, "K*");
    hParticleTypes->GetXaxis()->SetLabelSize(0.065);
    hParticleTypes->GetXaxis()->SetTitleOffset(1.2);
    hParticleTypes->GetXaxis()->SetTitle("Particle type");
    hParticleTypes->GetYaxis()->SetTitle("Probability of particle type");
    hParticleTypes->SetFillColor(kAzure - 2);
```

```
173    hParticleTypes->SetLineColor(kAzure - 2);
174    hParticleTypes->SetBarWidth(0.2);
175    hParticleTypes->SetBarOffset(0.8);
176    // if sumw2 is set to true the histogram doesn't get filled in
177    hParticleTypes->Sumw2(kFALSE);
178    hParticleTypes->Draw("b same");
179
180    // p-------------------------------------------------------------
181    Figure1->cd(2);
182
183    // normalize
184    hP->Scale(1. / hP->Integral(), "width");
185
186    // fitting
187    TF1* pDistr = new TF1("pDistr", "TMath::Exp(-x/[0])", 0., 5.);
188    pDistr->SetParameter(0, 1);
189    pDistr->SetParameter(1, 1);
190    hP->Fit(pDistr);
191
192    // fit output
193    std::cout << "\nP Distribution Fit: y = e^(-x/A)\n";
194
195    std::cout << "Parameter A: " << pDistr->GetParameter(0) << " ± "
196              << pDistr->GetParError(0) << "\n";
197
198    std::cout << "Reduced Chi Square: "
199              << pDistr->GetChisquare() / pDistr->GetNDF() << "\n";
200    std::cout << "Chi Square Probability: " << pDistr->GetProb() << "\n\n";
201
202    // graphics
203    hP->SetTitle("Impulse magnitudes' distribution");
204    hP->GetXaxis()->SetTitle("Impulse magnitude");
205    hP->GetYaxis()->SetTitle("Probability of impulse magnitude");
206    hP->GetXaxis()->SetTitleOffset(1.2);
207    hP->SetLineColor(kAzure - 2);
208    hP->Draw();
209
210    // phi-----------------------------------------------------------------
211    Figure1->cd(3);
212
213    // normalize
214    hPhi->Scale(1. / hPhi->Integral(), "width");
215
216    // fitting
217    TF1* phiDistr = new TF1("phiDistr", "[0]", 0, 2 * TMath::Pi());
218
219    hPhi->Fit(phiDistr);
220
221    // fit output
222    std::cout << "\nPhi Distribution Fit: y = A\n";
223    std::cout << "Parameter A: " << phiDistr->GetParameter(0) << " ± "
224              << phiDistr->GetParError(0) << "\n";
225
```

```
226   std::cout << "Reduced Chi Square: "
227             << phiDistr->GetChisquare() / phiDistr->GetNDF() << "\n";
228   std::cout << "Chi Square Probability: " << phiDistr->GetProb() << "\n\n";
229
230   // graphics
231   hPhi->SetTitle("Azimuthal angles' distribution");
232   hPhi->GetXaxis()->SetTitle("Azimuthal angle");
233   hPhi->GetYaxis()->SetTitle("Probability of azimuthal angle");
234   hPhi->GetXaxis()->SetTitleOffset(1.2);
235   hPhi->SetLineColor(kAzure - 2);
236   hPhi->Draw();
237
238   // theta-----------------------------------------------------------------
239   Figure1->cd(4);
240
241   // normalize
242   hTheta->Scale(1. / hTheta->Integral(), "width");
243
244   // fitting
245   TF1* thetaDistr = new TF1("thetaDistr", "[0]", 0, TMath::Pi());
246   hTheta->Fit(thetaDistr);
247
248   // fit output
249   std::cout << "\nTheta Distribution Fit: y = A\n";
250   std::cout << "Parameter A: " << thetaDistr->GetParameter(0) << " ± "
251             << thetaDistr->GetParError(0) << "\n";
252
253   std::cout << "Reduced Chi Square: "
254             << thetaDistr->GetChisquare() / thetaDistr->GetNDF() << "\n";
255   std::cout << "Chi Square Probability: " << thetaDistr->GetProb() << "\n\n";
256
257   // graphics
258   hTheta->SetTitle("Polar angles' distribution");
259   hTheta->GetXaxis()->SetTitle("Polar angle");
260   hTheta->GetYaxis()->SetTitle("Probability of polar angle");
261   hTheta->GetXaxis()->SetTitleOffset(1.2);
262   hTheta->SetFillColor(kAzure - 2);
263   hTheta->SetLineColor(kAzure - 2);
264   hTheta->Draw();
265
266   // Figure 2: invariant masses graphs -------------------------------------
267   TCanvas* Figure2 = new TCanvas("Figure2", "Figure2", 0, 0, 800, 600);
268   Figure2->Divide(3, 1);
269
270   // Only K*---------------------------------------------------------------
271   Figure2->cd(1);
272
273   // fitting
274   TF1* invariantMassDecayedDistr =
275       new TF1("invariantMassDecayedDistr", "gaus(0)", 0., 8.);
276   hInvariantMassDecayed->Fit(invariantMassDecayedDistr);
277
278   // fit output
```

```
279   std::cout << "\nK* Invariant mass fit: y = A*exp(-0.5*((x-M)/D)**2)\n";
280   std::cout << "Parameter A: " << invariantMassDecayedDistr->GetParameter(0)
281             << " ± " << invariantMassDecayedDistr->GetParError(0) << "\n";
282   std::cout << "Parameter M: " << invariantMassDecayedDistr->GetParameter(1)
283             << " ± " << invariantMassDecayedDistr->GetParError(1) << "\n";
284   std::cout << "Parameter D: " << invariantMassDecayedDistr->GetParameter(2)
285             << " ± " << invariantMassDecayedDistr->GetParError(2) << "\n";
286   std::cout << "Reduced Chi Square: "
287             << invariantMassDecayedDistr->GetChisquare()
288                    / invariantMassDecayedDistr->GetNDF()
289             << "\n";
290   std::cout << "Chi Square Probability: "
291             << invariantMassDecayedDistr->GetProb() << "\n\n";
292
293   // graphics
294   hInvariantMassDecayed->SetTitle("K* invariant masses");
295   hInvariantMassDecayed->GetXaxis()->SetTitle("Invariant mass");
296   hInvariantMassDecayed->GetYaxis()->SetTitle("Entries");
297   hInvariantMassDecayed->SetMarkerStyle(kFullSquare);
298   hInvariantMassDecayed->SetMarkerSize(0.5f);
299   hInvariantMassDecayed->Draw();
300
301   // Difference of discordant particles' invariant masses----------
302   // subtract
303   TH1F* hDiffMass = new TH1F(*hInvariantMassDiscordant);
304   hDiffMass->Add(hInvariantMassConcordant, -1);
305
306   Figure2->cd(2);
307
308   // fitting
309   TF1* diffMassDistr = new TF1("diffMassDistr", "gaus(0)", 0., 8.);
310   hDiffMass->Fit(diffMassDistr);
311
312   // fit output
313   std::cout
314       << "\nK* discordant invariant mass fit: y = A*exp(-0.5*((x-M)/D)**2)\n";
315   std::cout << "Parameter A: " << diffMassDistr->GetParameter(0) << " ± "
316             << diffMassDistr->GetParError(0) << "\n";
317   std::cout << "Parameter M: " << diffMassDistr->GetParameter(1) << " ± "
318             << diffMassDistr->GetParError(1) << "\n";
319   std::cout << "Parameter D: " << diffMassDistr->GetParameter(2) << " ± "
320             << diffMassDistr->GetParError(2) << "\n";
321   std::cout << "Reduced Chi Square: "
322             << diffMassDistr->GetChisquare() / diffMassDistr->GetNDF() << "\n";
323   std::cout << "Chi Square Probability: " << diffMassDistr->GetProb() << "\n\n";
324
325   // graphics
326   hDiffMass->SetTitle("Discordant particles' invariant masses");
327   hDiffMass->GetXaxis()->SetTitle("Invariant mass");
328   hDiffMass->GetYaxis()->SetTitle("Entries");
329   hDiffMass->SetLineColor(kAzure - 2);
330   hDiffMass->Draw();
331
```

```
  // Difference of discordant PiK particles' invariant masses-------
  // subtract
  TH1F* hDiffMassPiK = new TH1F(*hInvariantMassDiscordantPiK);
  hDiffMassPiK->Add(hInvariantMassConcordantPiK, -1);

  Figure2->cd(3);

  // fitting
  TF1* diffMassPiKDistr = new TF1("diffMassPiKDistr", "gaus(0)", 0., 8.);
  hDiffMassPiK->Fit(diffMassPiKDistr);

  // fit output
  std::cout << "\nK* discordant PiK invariant mass fit: y = "
               "A*exp(-0.5*((x-M)/D)**2)\n";
  std::cout << "Parameter A: " << diffMassPiKDistr->GetParameter(0) << " ± "
            << diffMassPiKDistr->GetParError(0) << "\n";
  std::cout << "Parameter M: " << diffMassPiKDistr->GetParameter(1) << " ± "
            << diffMassPiKDistr->GetParError(1) << "\n";
  std::cout << "Parameter D: " << diffMassPiKDistr->GetParameter(2) << " ± "
            << diffMassPiKDistr->GetParError(2) << "\n";
  std::cout << "Reduced Chi Square: "
            << diffMassPiKDistr->GetChisquare() / diffMassPiKDistr->GetNDF()
            << "\n";
  std::cout << "Chi Square Probability: " << diffMassPiKDistr->GetProb()
            << "\n\n";

  // graphics
  hDiffMassPiK->SetTitle("#pi - k invariant masses");
  hDiffMassPiK->GetXaxis()->SetTitle("Invariant mass");
  hDiffMassPiK->GetYaxis()->SetTitle("Entries");
  hDiffMassPiK->SetLineColor(kAzure - 2);
  hDiffMassPiK->Draw();
}
```

testParticleType.cpp

```
#define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
#include "ParticleType.hpp"
#include "ResonanceType.hpp"
#include "doctest.h"
#include <cstring> //for strcmp()
#include <iostream>

TEST_CASE("Testing ParticleType")
{
  SUBCASE("Testing Getter methods")
  {
    kape::ParticleType a{"a", 0.5, 1};
    CHECK(std::strcmp(a.GetName(), "a") == 0);
    CHECK(a.GetMass() == 0.5);
    CHECK(a.GetCharge() == 1);
  }
```

```
17
18    SUBCASE("Testing Print method")
19    {
20      kape::ParticleType a{"a", 0.5, 1};
21
22      std::cout << "\nPlease check that the 2 printed outputs are the same: \n\n";
23      std::cout << "Name:\ta\n";
24      std::cout << "Mass:\t0.5\n";
25      std::cout << "Charge:\t1\n\n";
26      a.Print();
27    }
28  }
29
30  TEST_CASE("Testing ResonanceType")
31  {
32    SUBCASE("Testing Getter methods")
33    {
34      kape::ResonanceType b{"b", 0.5, 1, 1.};
35      CHECK(std::strcmp(b.GetName(), "b") == 0);
36      CHECK(b.GetMass() == 0.5);
37      CHECK(b.GetCharge() == 1);
38      CHECK(b.GetWidth() == 1.);
39    }
40
41    SUBCASE("Testing Print method")
42    {
43      kape::ResonanceType b{"b", 0.5, 1, 1.};
44
45      std::cout << "\nPlease check that the 2 printed outputs are the same: \n\n";
46      std::cout << "Name:\tb\n";
47      std::cout << "Mass:\t0.5\n";
48      std::cout << "Charge:\t1\n";
49      std::cout << "Width:\t1\n\n";
50      b.Print();
51    }
52
53    SUBCASE("Testing Print override")
54    {
55      kape::ParticleType* particles[2];
56      particles[0] = new kape::ParticleType("ParticleType", 0.5, 1);
57      particles[1] = new kape::ResonanceType("ResonanceType", 1., -1, 1.);
58
59      std::cout << "\nPlease check that the 2 printed outputs are the same: \n\n";
60      std::cout << "Name:\tParticleType\n";
61      std::cout << "Mass:\t0.5\n";
62      std::cout << "Charge:\t1\n";
63
64      std::cout << "Name:\tResonanceType\n";
65      std::cout << "Mass:\t1\n";
66      std::cout << "Charge:\t-1\n";
67      std::cout << "Width:\t1\n\n";
68
69      for (int i = 0; i < 2; i++) {
```

```
70        particles[i]->Print();
71      }
72    }
73  }
```

testParticle.cpp

```
1   #define DOCTEST_CONFIG_IMPLEMENT_WITH_MAIN
2   #include "Particle.hpp"
3
4   #include "doctest.h"
5
6   TEST_CASE("Testing Particle class")
7   {
8     SUBCASE("Testing AddParticleType()")
9     {
10      CHECK(kape::Particle::GetNParticleType() == 0);
11      kape::Particle::AddParticleType("kape", 70., -1);
12      CHECK(kape::Particle::GetNParticleType() == 1);
13      kape::Particle::AddParticleType("samu", 63., 4, 1.);
14      CHECK(kape::Particle::GetNParticleType() == 2);
15      kape::Particle::AddParticleType("lele", 56., 18);
16      CHECK(kape::Particle::GetNParticleType() == 3);
17      kape::Particle::AddParticleType("nick", 79., -100, 0.40);
18      CHECK(kape::Particle::GetNParticleType() == 4);
19
20      kape::Particle gebbi{"kape", 10., -3., 0.};
21      CHECK(gebbi.GetIndex() == 0);
22      CHECK(gebbi.GetMass() == 70.);
23      gebbi.Print();
24      gebbi.SetIndex(1);
25      CHECK(gebbi.GetIndex() == 1);
26      CHECK(gebbi.GetMass() == 63.);
27      gebbi.Print();
28
29      kape::Particle::AddParticleType("nick", 79., +100, 0.40);
30      CHECK(kape::Particle::GetNParticleType() == 4);
31
32      kape::Particle::PrintParticleType();
33    }
34
35    SUBCASE("testing with array")
36    {
37      kape::Particle::AddParticleType("pi+", 0.13957, +1);      // pione +
38      kape::Particle::AddParticleType("pi-", 0.13957, -1);      // pione -
39      kape::Particle::AddParticleType("K+", 0.49367, +1);       // kaone +
40      kape::Particle::AddParticleType("K-", 0.49367, -1);       // kaone -
41      kape::Particle::AddParticleType("p+", 0.93827, +1);       // protone +
42      kape::Particle::AddParticleType("p-", 0.93827, -1);       // protone -
43      kape::Particle::AddParticleType("K*", 0.89166, 0, 0.050); // K*
44
45      std::array<kape::Particle, 300> eventParticles;
```

```
46      }
47  }
```