

Simulation of ALICE Project

Coli Simone

November 24th, 2021

1 Introduction

ALICE_Simulation is a computer program that simulates the ALICE experiment that has been conducted at CERN since 1993, which consists of analyzing the collisions occurring between particles at very high energies⁽¹⁾. Those particles would create different particles following a probability distribution (Table1).

Particle Type	Probability
π^+	40%
π^-	40%
k^+	5.0%
k^-	5.0%
P^+	4.5%
P^-	4.5%
k^*	1.0%

Table 1: *The table shows the probability of obtaining a specific particle from a collision of high energy particles*

In the simulation, we generated a finite amount of particles from the collision of the flux in the particle accelerator, each with a proper momentum, mass, and resulting energy, to maintain the conservation of those properties. The goal of the experiment is to detect the existence of Kaon 0 (k^*), a very rare particle that decays into either a Positive Pion (π^+) and Negative Kaon (k^-) or a Negative Pion (π^-) and Positive Kaon (k^+), after only $5.2 \times 10^{-8}s$. We stored the data of momentum, energy, charge, and invariant mass of all the particles to detect the presence of differences that could indicate the existence of the Kaon.

The program we implemented stores the information about the particles and generates histograms from which we studied the system.

2 Code Structure

The code's division into different files and folders has the background idea of making it more orderly. There are eight files for the simulation program (one main file: `mainE.cpp`, one libraries file: `library.hpp`, three header files: `ParticleType.hpp`, `ResonanceType.hpp`, `Particle.hpp`, and three source files: `ParticleType.cpp`, `ResonanceType.cpp`, `Particle.cpp`) and one for the data analysis (`analysis.C`). The header

files contain the classes implemented for the proper functioning of the simulation, whereas the source files contain the implementation of the methods defined in the headers.

ParticleType Class

The class `ParticleType` creates a homonymous type that contains the name, the mass, and the charge of a particular particle, respectively as a `std::string`, a `double`, and an `integer`. This class has five methods, two of which are virtual.

ResonanceType Class

The `ResonanceType` class is a derived class from `ParticleType` and, in addition to the base class items, contains the information about the width of the particle as a `double`. The type defined with the name of this class creates a particle with a width, contrary to what happens to a `ParticleType` object, in which the width of the particle is always zero. This class has two methods, both of them are the override of the already existing virtual methods in the base class.

Particle Class

The class `Particle` is the one that allows the creation of a particle giving it a random momentum and making it decay into other particles if necessary. It also creates a set of particle type, each with a proper index as an identifier. The variables in this class are three momentum components (`fPx`, `fPy`, `fPz`), an array of `ParticleType` and its dimension (`fParticleType`, `fMaxNumParticleType`), an index of particle type (`fNParticleType`), and a numeric code proper of each particle type (`fIndex`). This class has, several methods, including some static, meaning they are accessible from the main without defining an object.

3 Generation

In the simulation, there had been 100000 collision events, each using a set of 100 particles. The particles resulting from the collisions were Positive Pions (π^+), Negative Pions (π^-), Positive Kaons (k^+), Negative Kaons (k^-), Protons (p^+), Antiprotons (p^-), and Resonance Kaons (k^*), generated randomly using a uniform distribution and the probability shown in Table 1. The module of the momentum of the particles comes from an exponential distribution with a mean of 1. Its direction drives from the cartesian components:

$$\begin{cases} p_x = |p| \cdot \cos \theta \cdot \cos \phi \\ p_y = |p| \cdot \cos \theta \cdot \sin \phi \\ p_z = |p| \cdot \sin \theta \end{cases} \quad (1)$$

Where the azimuthal angle theta (θ) and the polar angle phi (ϕ) are generated using a uniform random distribution respectively from 0 to π and the second from 0 to 2π . In the case that a Resonance Kaon is created from the collision of particles, it decays into either a Positive Pion and a Negative Kaon or a Negative Pion and a Positive Kaon with the same probability. The momentum of these new two particles comes from a normal distribution.

4 Analysis

The generation of particle types is compatible with the theoretical calculation as shown in Table. 2

Particle Type	Entries	Error	Theo. Ent.
π^+	4000250	2000	$4.0 \cdot 10^6$
π^-	3998440	2000	$4.0 \cdot 10^6$
k^+	500727	707	$5.0 \cdot 10^5$
k^-	499365	707	$5.0 \cdot 10^5$
P^+	450417	671	$4.5 \cdot 10^5$
P^-	450536	671	$4.5 \cdot 10^5$
k^*	100264	317	$1.0 \cdot 10^5$

Table 2: The table shows the entries from the simulation with the respective error calculated using root, and the theoretical entries calculated taking the percentage of each particle type from the total amount of particles.

The angles and momentum distributions are fitting to the relative uniform and exponential distribution as shown in table 3.

Distribution	Fit's Parameters	χ^2	D.O.F.	Reduced χ^2
Polar Angles (pol0)	9999 ± 3	973.8	999	0.9748
Azimutal Angols (pol0)	9999 ± 3	902.8	999	0.9037
Momentum (expo)	$(9939 \pm 3) \cdot 10^{-4}$	959.2	998	0.9961

Table 3: The table shows the results of the fits from the histograms of the angles and the momentum.

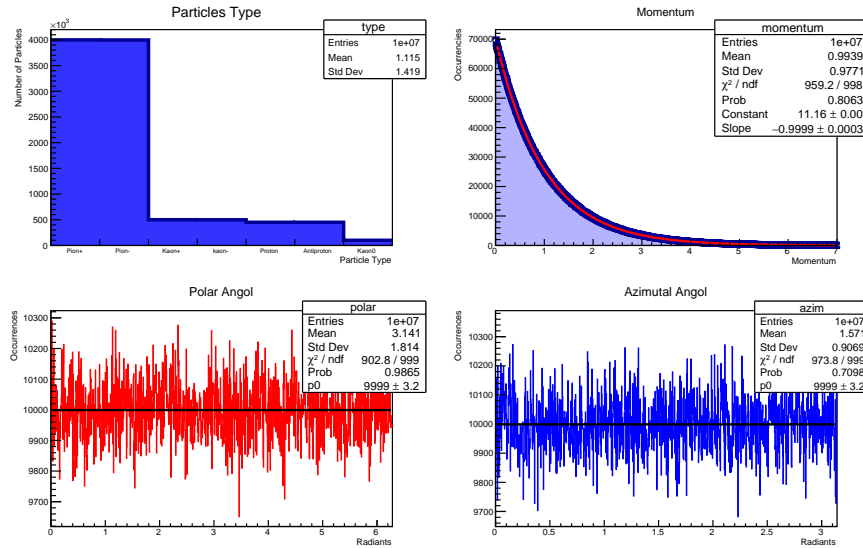


Figure 1: The image shows the fit between the uniform distribution compared with the histograms of the angles randomly generated and the histogram of the momentum with the exponential distribution.

It is possible to analyze these phenomena by looking at the histograms of the invariant mass, because its definition contains both the momentum and the energy of the particles, it is, therefore, conserved in the collision. The construction of the invariant mass histogram consist of counting the number of particles per mass invariant value, and to detect the resonance Kaon. We considered the histogram only for Pion and Kaon with different charges and we compared it with the histogram of the Pion and Kaon with the same charges. Because the resonance Kaon decays very quickly into those two particles with opposite charges, there should be a little "bump" in the first of those two histograms. A second approach consists of comparing the two histograms of the invariant mass of all the particles with the same charge and opposite charges.

Subtracting the two pairs of histograms gives, in both cases, a bell shape, fitting a normal distribution with mean being the mass of the resonance Kaon and sigma its width, should appear in the canvas. Comparing the results with the testing histogram of the decay, filled during the simulation, fitting a normal distribution, gives a value that stands inside the error.

In conclusion, from the previous considerations, we had been able to detect the presence

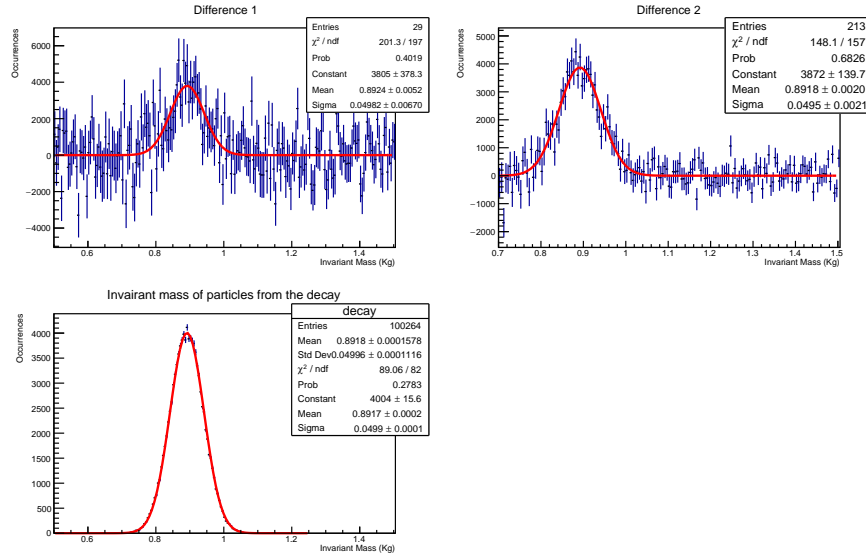


Figure 2: The image shows the histograms of the invariant mass calculated in the three different ways, all of them fitted using a normal distribution.

of a resonance Kaon and the result of the simulated experiment has been summarized in table 4.

Distribution	Mean	Sigma	Amplitude	χ^2
Inv. Mass. Decay	0.8917 ± 0.0002	4004 ± 15	999	1.086
Inv. Mass. Charges	0.8924 ± 0.0052	3805 ± 378	999	1.021
Inv. Mass. Kaon-Pion	0.8918 ± 0.0020	3872 ± 140	998	0.9433

Table 4: The table summarize the result of the fit of all the histograms of the invariant mass.

Word Citation

1. Alice Experiment. CERN. <https://home.cern/science/experiments/alice> .
November 29th, 2021.

Appendix

Code

4.1 ParticleType.hpp

```
1
2 #ifndef PARTICLETYPE_HPP
3 #define PARTICLETYPE_HPP
4
5 #include <string>
6
7 class ParticleType {
8 private:
9     const std::string fName;
10    const double fMass;
11    const int fCharge;
12 public:
13
14    //Constructor
15    ParticleType(std::string Name, double Mass, int Charge);
16
17    //Getters
18    std::string GetParticleName() const;
19    double GetParticleMass() const;
20    int GetParticleCharge() const;
21    virtual double GetParticleWidth() const;
22
23    //Printer
24    virtual void Print() const;
25 };
26
27 #endif
28
```

4.2 ParticleType.cpp

```
1
2 #include <iostream>
3 #include <string>
4 #include "ParticleType.hpp"
5
6 //Class ParticleType
7 //-----
8 //Constructor
9 ParticleType::ParticleType(std::string Name, double Mass, int Charge):
10 fName(Name),
11 fMass(Mass),
12 fCharge(Charge)
13 {}
14
15 //Getters
16 std::string ParticleType::GetParticleName() const { return fName; }
17 double ParticleType::GetParticleMass() const { return fMass; }
18 int ParticleType::GetParticleCharge() const { return fCharge; }
19 double ParticleType::GetParticleWidth() const { return 0; }
```

```

20
21
22 //Printer
23 void ParticleType::Print() const {
24     std::cout << "Particle's Name: "<<fName << '\n';
25     std::cout << "Particle's Mass: "<<fMass << '\n';
26     std::cout << "Particle's Charge: "<<fCharge << '\n';
27     std::cout << "-----" << '\n';
28 }
29 //-----
30

```

4.3 ResonanceType.hpp

```

1
2 #ifndef RESONANCETYPE_HPP
3 #define RESONANCETYPE_HPP
4
5 #include <string>
6 #include "../ParticleType/ParticleType.hpp"
7
8 class ResonanceType: public ParticleType {
9 private:
10     const double fWidth;
11 public:
12
13     //Constructor
14     ResonanceType(std::string Name, double Mass, int Charge, double Width
15 );
16
17     //Printer
18     void Print() const;
19
20     //Getter
21     double GetParticleWidth() const;
22 };
23
24 #endif
25

```

4.4 ResonanceType.cpp

```

1
2 #include <iostream>
3
4 #include <string>
5 #include "ResonanceType.hpp"
6
7 //Class ResonanceType
8 //-----
9 //Constructor
10 ResonanceType::ResonanceType(std::string Name, double Mass, int Charge,
11     double Width):
12     ParticleType(Name, Mass, Charge),
13     fWidth(Width)
14 {}
15

```

```

15 //Getter
16 double ResonanceType::GetParticleWidth() const { return fWidth; }
17
18 //Printer
19 void ResonanceType::Print() const {
20     ParticleType::Print();
21     std::cout << "Width of the Particle's Resonance: " << fWidth << '\n';
22 }
23 //-----
24

```

4.5 Particle.hpp

```

1
2 #ifndef PARTICLE_HPP
3 #define PARTICLE_HPP
4
5 #include <string>
6 #include <vector>
7 #include "../ParticleType/ParticleType.hpp"
8
9 enum class PL{Electron, Positron, Proton, Antiproton, PionPlus,
10     PionMinus, Pion0, KaonPlus, KaonMinus, Kaon0};
11
12 class Particle {
13 private:
14     double fPx;
15     double fPy;
16     double fPz;
17
18     static const int fMaxNumParticleType = 10;
19     static ParticleType* fParticleType[fMaxNumParticleType];
20     static int fNParticleType;
21     int fIndex;
22
23     static int FindParticle(std::string PTBF); //Particle To Be Found
24
25     void Boost(double bx, double by, double bz);
26
27 public:
28     Particle(std::string name, double Px, double Py, double Pz);
29     Particle() = default;
30     static void AddParticle(std::string name, double mass, int charge,
31         double with=0);
32     static void AddParticle(PL particle);
33
34     int GetIndex() const;
35     int GetCharge() const;
36     std::string GetName() const;
37     double GetPx() const;
38     double GetPy() const;
39     double GetPz() const;
40     double GetMass() const;
41     double GetMomentum() const;
42     double GetTotalEnergy() const;
43     double GetInvMass(Particle& p) const;
44     double GetTrasMomentum() const;

```



```

44     void SetIndex(std::string);
45     void SetIndex(int index);
46     void SetMomentum(double x, double y, double z);
47
48     static void PrintTable();
49     void PrintParticle() const;
50
51     static void ParticleFeatures(PL& particle, int const N);
52
53     int Decay2body(Particle &dau1, Particle &dau2) const;
54
55
56 };
57
58 #endif
59
60

```

4.6 Particle.cpp

```

1
2 #include "../libraries/library.hpp"
3
4 #include <string>
5 #include <iostream>
6 #include <cstdlib>
7 #include <cmath>
8 #include <random>
9
10 int Particle::fNParticleType = 0;
11 ParticleType* Particle::fParticleType[fMaxNumParticleType];
12
13 // Public Methods //////////////////////////////////////
14 int Particle::FindParticle(std::string PTBF) {
15     int i = 0;
16     for(; i < fNParticleType; ++i) {
17         std::string ParticleName = fParticleType[i]->GetParticleName();
18         if(ParticleName == PTBF) { return i; }
19         else if (fNParticleType == 0) { return 0; }
20     }
21     return 10;
22 }
23
24 void Particle::AddParticle(std::string name, double mass, int charge,
25 double width) {
26     int const N = fNParticleType;
27     int chack = FindParticle(name);
28     if(N < fMaxNumParticleType && chack==10) {
29         if(width!=0) {
30             fParticleType[N] = new ResonanceType(name, mass, charge, width);
31             ++fNParticleType;
32         } else {
33             fParticleType[N] = new ParticleType (name, mass, charge);
34             ++fNParticleType;
35         }
36     } else {
37         std::cout << "!! -- The particle Does already exist -- !! " << '\n';
38     }
39 }

```

```

37     }
38 }
39
40 int Particle::Decay2body(Particle &dau1, Particle &dau2) const {
41     if(GetMass() == 0.0){
42         printf("Decayment cannot be preformed if mass is zero\n");
43         return 1;
44     }
45
46     double massMot = GetMass();
47     double massDau1 = dau1.GetMass();
48     double massDau2 = dau2.GetMass();
49
50     if(fIndex < 10){ // add width effect
51
52         // gaussian random numbers
53
54         float x1, x2, w, y1, y2;
55
56         double invnum = 1./RAND_MAX;
57         do {
58             x1 = 2.0 * rand()*invnum - 1.0;
59             x2 = 2.0 * rand()*invnum - 1.0;
60             w = x1 * x1 + x2 * x2;
61         } while ( w >= 1.0 );
62
63         w = sqrt( (-2.0 * log( w ) ) / w );
64         y1 = x1 * w;
65         y2 = x2 * w;
66         massMot += fParticleType[fIndex]->GetParticleWidth() * y1;
67     }
68     if(massMot < massDau1 + massDau2){
69         printf("Decayment cannot be preformed because mass is too low in
70 this channel\n");
71         return 2;
72     }
73     double pout = sqrt((massMot*massMot - (massDau1+massDau2)*(massDau1+
74 massDau2))*(massMot*massMot - (massDau1-massDau2)*(massDau1-massDau2))
75 )/massMot*0.5;
76     double norm = 2*M_PI/RAND_MAX;
77     double phi = rand()*norm;
78     double theta = rand()*norm*0.5 - M_PI/2.;
79     dau1.SetMomentum(pout*sin(theta)*cos(phi), pout*sin(theta)*sin(phi),
80 pout*cos(theta));
81     dau2.SetMomentum(-pout*sin(theta)*cos(phi), -pout*sin(theta)*sin(phi),
82 -pout*cos(theta));
83     double energy = sqrt(fPx*fPx + fPy*fPy + fPz*fPz + massMot*massMot);
84     double bx = fPx/energy;
85     double by = fPy/energy;
86     double bz = fPz/energy;
87     dau1.Boost(bx, by, bz);
88     dau2.Boost(bx, by, bz);
89     return 0;
90 }
91
92 void Particle::Boost(double bx, double by, double bz)
93 {
94     double energy = GetTotalEnergy();
95     //Boost this Lorentz vector

```

```

91     double b2 = bx*bx + by*by + bz*bz;
92     double gamma = 1.0 / sqrt(1.0 - b2);
93     double bp = bx*fPx + by*fPy + bz*fPz;
94     double gamma2 = b2 > 0 ? (gamma - 1.0)/b2 : 0.0;
95
96     fPx += gamma2*bp*bx + gamma*bx*energy;
97     fPy += gamma2*bp*by + gamma*by*energy;
98     fPz += gamma2*bp*bz + gamma*bz*energy;
99 }
100
101 // Constructor //////////////////////////////////////
102 Particle::Particle(std::string name, double Px = 0, double Py = 0,
103     double Pz = 0):
104     fPx(Px),
105     fPy(Py),
106     fPz(Pz),
107     fIndex (FindParticle(name))
108 { if(fIndex == 10) { std::cout << "!! -- This Type of Particle does not
109     Exist -- !" << '\n'; } }
110
111 // Getter Methods //////////////////////////////////////
112 int Particle::GetIndex() const { return fIndex; }
113
114 int Particle::GetCharge() const {return fParticleType[fIndex]->
115     GetParticleCharge(); }
116
117 std::string Particle::GetName() const {return fParticleType[fIndex]->
118     GetParticleName();}
119
120 double Particle::GetPx() const { return fPx; }
121
122 double Particle::GetPy() const { return fPy; }
123
124 double Particle::GetPz() const { return fPz; }
125
126 double Particle::GetMass() const {
127     return (fParticleType[fIndex]->GetParticleMass());
128 }
129
130 double Particle::GetMomentum() const {
131     return (fPx*fPx + fPy*fPy + fPz*fPz);
132 }
133
134 double Particle::GetTrasMomentum() const {
135     return fPx*fPx + fPy*fPy;
136 }
137
138 double Particle::GetTotalEnergy() const {
139     double m = GetMass();
140     double p2 = GetMomentum();
141     return sqrt(m*m + p2);
142 }
143
144 double Particle::GetInvMass(Particle& p) const {
145     double E1 = GetTotalEnergy();
146     double E2 = p.GetTotalEnergy();
147     double Psum = (fPx+p.GetPx())*(fPx+p.GetPx())+(fPy+p.GetPy())*(fPy+p.
148     GetPy())+(fPz+p.GetPz())*(fPz+p.GetPz());
149     double M = sqrt((E1+ E2)*(E1+ E2) - Psum);

```

```

145     return M;
146 }
147
148 // Setter Methods //////////////////////////////////////
149 void Particle::SetIndex(std::string type) {
150     const int index = FindParticle(type);
151     if(10 != index) {
152         fIndex = index;
153     }
154 }
155
156 void Particle::SetIndex(int index) {
157     if(index < fNParticleType) {
158         fIndex = index;
159     }
160 }
161
162 void Particle::SetMomentum(double x, double y, double z) {
163     fPx = x;
164     fPy = y;
165     fPz = z;
166 }
167
168 // Printer //////////////////////////////////////
169 void Particle::PrintTable() {
170     for(int i = 0; i < fNParticleType; ++i) {
171         fParticleType[i]->Print();
172     }
173 }
174
175 void Particle::PrintParticle() const {
176     std::cout << "Particle index: " << fIndex << '\n';
177     std::cout << "Particle name: " << fParticleType[fIndex]->
GetParticleName() << '\n';
178     std::cout << "Px: " << fPx << '\n';
179     std::cout << "Py: " << fPy << '\n';
180     std::cout << "Pz: " << fPz << '\n';
181     std::cout << "-----" << '\n';
182 }
183
184 // Particles List //////////////////////////////////////
185 void Particle::ParticleFeatures(PL& particle, int const N) {
186     int check;
187     switch (particle) {
188         case (PL::Electron):
189             check = FindParticle("Electron");
190             if (check == 10) {
191                 fParticleType[N] = new ParticleType ("Electron", 0.0005109, -1)
;
192                 ++fNParticleType;
193             } else {
194                 std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
195             }
196             break;
197         case (PL::Proton) :
198             check = FindParticle("Proton");
199             if (check == 10) {
200                 fParticleType[N] = new ParticleType ("Proton", 0.938327, +1);

```

```

201         ++fNParticleType;
202     } else {
203         std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
204     }
205     break;
206     case (PL::Positron) :
207         check = FindParticle("Positron");
208         if(check == 10) {
209             fParticleType[N] = new ParticleType ("Positron", 0.0005109, +1)
;
210             ++fNParticleType;
211         } else {
212             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
213         }
214         break;
215     case (PL::PionMinus):
216         check = FindParticle("Pion-");
217         if(check == 10) {
218             fParticleType[N] = new ParticleType ("Pion-", 0.13957, +1);
219             ++fNParticleType;
220         } else {
221             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
222         }
223         break;
224     case (PL::PionPlus) :
225         check = FindParticle("Pion+");
226         if(check == 10) {
227             fParticleType[N] = new ParticleType ("Pion+", 0.13957, -1);
228             ++fNParticleType;
229         } else {
230             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
231         }
232         break;
233     case (PL::Pion0) :
234         check = FindParticle("Pion0");
235         if(check == 10) {
236             fParticleType[N] = new ParticleType ("Pion0", 0.1350, 0);
237             ++fNParticleType;
238         } else {
239             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
240         }
241         break;
242     case (PL::KaonPlus) :
243         check = FindParticle("Kaon+");
244         if(check == 10) {
245             fParticleType[N] = new ParticleType ("Kaon+", 0.49367, +1);
246             ++fNParticleType;
247         } else {
248             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
249         }
250         break;
251     case (PL::KaonMinus) :
252         check = FindParticle("Kaon-");

```

```

253         if(check == 10) {
254             fParticleType[N] = new ParticleType ("Kaon-", 0.49367, -1);
255             ++fNParticleType;
256         } else {
257             std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
258         }
259         break;
260         case (PL::Kaon0) :
261             check = FindParticle("Kaon0");
262             if(check == 10) {
263                 fParticleType[N] = new ResonanceType ("Kaon0", 0.89166, 0,
0.05);
264                 ++fNParticleType;
265             } else {
266                 std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
267             }
268             break;
269         case (PL::Antiproton) :
270             check = FindParticle("Antiproton");
271             if(check == 10) {
272                 fParticleType[N] = new ParticleType ("Antiproton", 0.93827, -1)
;
273                 ++fNParticleType;
274             } else {
275                 std::cout << "!! -- The particle Does already exist -- !! " <<
'\n';
276             }
277             break;
278         default:
279             std::cout << "!! -- Particle not in the list, add it using the
standard AddParticle -- !!" << '\n';
280         }
281     }
282
283     void Particle::AddParticle(PL particle) {
284         int const N = fNParticleType;
285         ParticleFeatures(particle, N);
286     }
287

```

4.7 mainE.cpp

```

1
2     #include "../libraries/library.hpp"
3     #include <iostream>
4     #include <cmath>
5
6     void ProgressionBar(int Progression) {
7         double n = (Progression/5.22E8);
8         std::cout << "[";
9         int pos = 70 * n;
10        for (int i = 0; i < 70; ++i) {
11            if (i < pos) std::cout << "=";
12            else if (i == pos) std::cout << ">";
13            else std::cout << " ";
14        }

```

```

15     std::cout << "]" << int(n * 100.0) << " %\r";
16     std::cout.flush();
17 }
18
19 int main() {
20     double const pi = 3.1415926535;
21     int progression = 0;
22     TRandom* Random = new TRandom();
23     Random->SetSeed(0);
24
25     Particle::AddParticle(PL::PionPlus);
26     Particle::AddParticle(PL::PionMinus);
27     Particle::AddParticle(PL::KaonPlus);
28     Particle::AddParticle(PL::KaonMinus);
29     Particle::AddParticle(PL::Proton);
30     Particle::AddParticle(PL::Antiproton);
31     Particle::AddParticle(PL::Kaon0);
32
33     TH1F* type = new TH1F("type", "Particles Type", 7, 0, 7);
34     type->GetXaxis()->TAxis::SetBinLabel(1, "Pion+");
35     type->GetXaxis()->TAxis::SetBinLabel(2, "Pion-");
36     type->GetXaxis()->TAxis::SetBinLabel(3, "Kaon+");
37     type->GetXaxis()->TAxis::SetBinLabel(4, "kaon-");
38     type->GetXaxis()->TAxis::SetBinLabel(5, "Proton");
39     type->GetXaxis()->TAxis::SetBinLabel(6, "Antiproton");
40     type->GetXaxis()->TAxis::SetBinLabel(7, "Kaon0");
41
42     TH1F* momentum = new TH1F("momentum", "Momentum", 1000, 0, 7);
43     TH1F* tmomentum = new TH1F("tmomentum", "Transversal Momentum", 1000,
44         0, 7);
45     TH1F* invmass = new TH1F("invmass", "Invariant Mass", 1000, 0, 5);
46     TH1F* energy = new TH1F("energy", "Energy", 1000, 0, 7);
47     TH1F* azim = new TH1F("azim", "Azimutal Angol", 1000, 0, pi);
48     TH1F* polar = new TH1F("polar", "Polar Angol", 1000, 0, 2*pi);
49     TH1F* invmassdis = new TH1F("invmassdis", "Invariant Mass opposite
50         charges", 1000, 0, 5);
51     TH1F* invmasscon = new TH1F("invmasscon", "Invariant Mass same charges"
52         , 1000, 0, 5);
53     TH1F* invppkmpmkp = new TH1F("invppkmpmkp", "Invariant Mass pion+/kaon-
54         & pion-/kaon+", 1000, 0, 5);
55     //invariant (i) mass (m) of pion (p) plus (p) and kaon (k) minus (m) or
56     //pion (p) minus (m) and kaon (k) plus (p)
57     //i-m-p-p-k-m-p-m-k-p
58     invppkmpmkp->SetLineColor(kRed);
59     TH1F* invppkppmkm = new TH1F("invppkppmkm", "Invariant Mass pion+/kaon+
60         & pion-/kaon-", 1000, 0, 5);
61     //invariant (i) mass (m) of pion (p) minus (m) and kaon (k) plus (p) or
62     //pion (p) minus (m) and kaon (k) minus (k)
63     //i-m-p-p-k-p-p-m-k-m
64     invppkppmkm->SetLineColor(kBlue);
65     TH1F* decay = new TH1F("decay", "Invairant mass of particles from the
66         decay", 1000, 0, 5);
67
68     invmass->Sumw2();
69     invmassdis->Sumw2();
70     invmasscon->Sumw2();
71     invppkmpmkp->Sumw2();
72     invppkppmkm->Sumw2();
73     decay->Sumw2();

```

```

66
67 int const N = 100;
68 int const extra = 20;
69 Particle Particella[N+extra];
70
71 for(int i = 0; i < 1E5; ++i) {
72     int ExtraCounter = 0;
73     for(int j = 0; j < (N); ++j) {
74
75         double phi = 2*pi*Random->Uniform(0.0,1.0);
76         double theta = pi*Random->Uniform(0.0,1.0);
77
78         azim->Fill(theta);
79         polar->Fill(phi);
80
81         double P = Random->Exp(1.0);
82
83         double Px = P*cos(phi)*cos(theta);
84         double Py = P*cos(phi)*sin(theta);
85         double Pz = P*sin(phi);
86
87         Particella[j].SetMomentum(Px, Py, Pz);
88
89         //Setting the type //////////////////////////////////////
90         double prob = Random->Uniform(0.0,100.0);
91
92         if(prob<40) {
93             Particella[j].SetIndex("Pion+");
94         } else if (prob<80) {
95             Particella[j].SetIndex("Pion-");
96         } else if (prob<85) {
97             Particella[j].SetIndex("Kaon+");
98         } else if (prob<90) {
99             Particella[j].SetIndex("Kaon-");
100        } else if (prob<94.5) {
101            Particella[j].SetIndex("Proton");
102        } else if (prob<99) {
103            Particella[j].SetIndex("Antiproton");
104        } else {
105            Particella[j].SetIndex("Kaon0");
106            int c = ExtraCounter + N;
107            double chance = Random->Uniform(0.0,1.0);
108            if(chance < 0.5) {
109                Particella[c].SetIndex("Pion+");
110                Particella[c+1].SetIndex("Kaon-");
111            } else {
112                Particella[c].SetIndex("Pion-");
113                Particella[c+1].SetIndex("Kaon+");
114            }
115            int check = Particella[j].Decay2body(Particella[c], Particella[c
+1]);
116            if (check != 0) return check;
117            double MassInvCondition = Particella[c].GetInvMass(Particella[c
+1]);
118            decay->Fill(MassInvCondition);
119            ExtraCounter = ExtraCounter +2;
120        }
121        int index = Particella[j].GetIndex();
122        type->Fill(index);

```



```

123     double PTModule = sqrt(Particella[j].GetTrasMomentum());
124     tmomentum->Fill(PTModule);
125     double PModule = sqrt(Particella[j].GetMomentum());
126     momentum->Fill(PModule);
127     double Energy = Particella[j].GetTotalEnergy();
128     energy->Fill(Energy);
129     ++progression;
130 }
131 double MassInv;
132 for(int k = 0; k < N+ExtraCounter; ++k) {
133     for(int h = k+1 ; h<N+ExtraCounter; ++h){
134         MassInv = Particella[k].GetInvMass(Particella[h]);
135         invmass->Fill(MassInv);
136         if((Particella[k].GetCharge() * Particella[h].GetCharge()) < 0) {
137             double MassInvCondition = Particella[k].GetInvMass(Particella[h
138 ]);
139             invmassdis->Fill(MassInvCondition);
140         } else if((Particella[k].GetCharge() * Particella[h].GetCharge())
141 > 0) {
142             double MassInvCondition = Particella[k].GetInvMass(Particella[h
143 ]);
144             invmasscon->Fill(MassInvCondition);
145         }
146         if (Particella[k].GetName() == "Pion+" && Particella[h].GetName()
147 == "Kaon-") {
148             double MassInvCondition = Particella[k].GetInvMass(Particella[h
149 ]);
150             invppkmpmkp->Fill(MassInvCondition);
151         } else if(Particella[k].GetName() == "Pion-" && Particella[h].
152 GetName() == "Kaon+") {
153             double MassInvCondition = Particella[k].GetInvMass(Particella[h
154 ]);
155             invppkmpmkp->Fill(MassInvCondition);
156         } else if (Particella[k].GetName() == "Pion+" && Particella[h].
157 GetName() == "Kaon+") {
158             double MassInvCondition = Particella[k].GetInvMass(Particella[h
159 ]);
160             invppkppmkp->Fill(MassInvCondition);
161         } else if(Particella[k].GetName() == "Pion-" && Particella[h].
162 GetName() == "Kaon-") {
163             double MassInvCondition = Particella[k].GetInvMass(Particella[h
164 ]);
165             invppkppmkp->Fill(MassInvCondition);
166         }
167     }
168     ++progression;
169 }
170 ProgressionBar(progression);
171 }
172
173 TCanvas *canv1 = new TCanvas("canv1", "Type");
174 type->Draw();
175 TCanvas *canv2 = new TCanvas("canv2", "Momentum");
176 canv2->Divide(1,2);
177 canv2->cd(1);
178 momentum->Draw();
179 canv2->cd(2);
180 tmomentum->Draw();
181 TCanvas *canv12 = new TCanvas("canv12", "energy");

```

```

171 energy->Draw();
172 TCanvas *canv4 = new TCanvas("canv4", "invMass");
173 invmass->Draw();
174 TCanvas *canv5 = new TCanvas("canv5", "polarAngle");
175 polar->Draw();
176 TCanvas *canv6 = new TCanvas("canv6", "azimutalAngle");
177 azim->Draw();
178 TCanvas *canv7 = new TCanvas("canv7", "Invariant Mass opposite charges"
    );
179 invmassdis->Draw("histo");
180 TCanvas *canv8 = new TCanvas("canv8", "invariant Mass same charges");
181 invmasscon->Draw("histo");
182 TCanvas *canv9 = new TCanvas("canv9", "Invariant Mass of pion and kaon"
    );
183 canv9->Divide(1,2);
184 canv9->cd(1);
185 invppkppmkm->Draw("histo");
186 canv9->cd(2);
187 invppkmpmkp->Draw("histo");
188 TCanvas *canv10 = new TCanvas("canv10", "Invariant Mass pion+/kaon+ &
    pion-/kaon-");
189 invppkppmkm->Draw("histo");
190 invppkmpmkp->Draw("histo same");
191 TCanvas *canv11 = new TCanvas("canv11", "Decay");
192 decay->Draw("histo");
193
194 canv1->Print("../histograms/type.pdf");
195 canv2->Print("../histograms/Momentums.pdf");
196 canv12->Print("../histograms/energy.pdf");
197 canv4->Print("../histograms/invMass.pdf");
198 canv5->Print("../histograms/polarAngle.pdf");
199 canv6->Print("../histograms/azimutalAngle.pdf");
200 canv7->Print("../histograms/invmassdis.pdf");
201 canv8->Print("../histograms/invmasscon.pdf");
202 canv9->Print("../histograms/invppkmpmkp.pdf");
203 canv10->Print("../histograms/invppkppmkm.pdf");
204 canv11->Print("../histograms/decay.pdf");
205
206 TFile* RFile = new TFile("ALICE_Simulation.root", "RECREATE");
207 RFile->cd();
208 type->Write();
209 momentum->Write();
210 tmomentum->Write();
211 energy->Write();
212 invmass->Write();
213 polar->Write();
214 azim->Write();
215 invmassdis->Write();
216 invmasscon->Write();
217 invppkmpmkp->Write();
218 invppkppmkm->Write();
219 decay->Write();
220 RFile->ls();
221 RFile->Close();
222
223
224 return 0;
225 }
226

```

4.8 library.hpp

```
1
2 #ifndef LIBRARY_HPP
3 #define LIBRARY_HPP
4
5 #include "../script/ParticleType/ParticleType.hpp"
6 #include "../script/ResonanceType/ResonanceType.hpp"
7 #include "../script/Particle/Particle.hpp"
8 #include "TRandom.h"
9 #include "TAxis.h"
10 #include "TH1.h"
11 #include "TCanvas.h"
12 #include "TFile.h"
13
14 #endif
15
```

4.9 analysis.C

```
1
2 void setStyle() {
3     gROOT->SetStyle("Modern");
4     gStyle->SetPalette(56);
5     gStyle->SetOptFit(111);
6 }
7
8 // Function that checks the generation of the values //////////
9
10 void Checks() {
11
12     int optarg = 1111;
13
14     // Check of the genercation of particles //////////
15     double particleProb[7];
16
17     particleProb[0]= 0.4;
18     particleProb[1]= 0.4;
19     particleProb[2]= 0.05;
20     particleProb[3]= 0.05;
21     particleProb[4]= 0.045;
22     particleProb[5]= 0.045;
23     particleProb[6]= 0.01;
24
25     TFile* c = new TFile("Alice_Simulation.root","read");
26     TH1F* type = (TH1F*)c->Get("type");
27     cout<<"Bin Content - Error - Theo. Val.\n";
28     cout<<"-----\n";
29     for(int i=1; i<8; ++i) {
30         cout<<type->GetBinContent(i)<<" - "<<type->GetBinError(i)<<" - "<<
31         particleProb[i-1]*1E7<<"\n";
32         cout<<"-----\n";
33     }
34
35     // CCheck of the shape of the momentum //////////
36     TCanvas* c1 = new TCanvas("c1", "Momentum");
37     TH1F* Mom = (TH1F*)c->Get("momentum");
38     Mom->Fit("expo", "", "");
39
40 }
```

```

38 Mom->SetFillColorAlpha(kBlue, 0.3);
39 Mom->SetLineWidth(6);
40 Mom->Draw();
41 c1->Print("../fit/Momentumfit.pdf");
42 gPad->Update();
43 TPaveStats* ft = (TPaveStats*)Mom->FindObject("stats");
44 ft->SetOptFit(optarg);
45 cout<<"-----\n";
46 cout<<Mom->GetMean()<<" - "<<Mom->GetRMS()<<" - "<< 1<<'\\n';
47 cout<<"-----\n";
48
49 // Cheack of the angles //////////////////////////////////////
50 TCanvas* c2 = new TCanvas("c2", "Angoli");
51 c2->Divide(1,2);
52
53 c2->cd(1);
54 TH1F* Pol = (TH1F*)c->Get("polar");
55 Pol->Fit("pol0");
56 TF1* f1 = Pol->GetFunction("pol0");
57 f1->SetLineColor(kBlack);
58 Pol->SetLineColor(kRed);
59 Pol->Draw();
60 gPad->Update();
61 TPaveStats* ft1 = (TPaveStats*)Pol->GetListOfFunctions()->FindObject(
"stats");
62 ft1->SetOptFit(optarg);
63 gStyle->SetStatW(.2);
64 gStyle->SetStatH(.4);
65
66 c2->cd(2);
67 TH1F* azim = (TH1F*)c->Get("azim");
68 azim->Fit("pol0");
69 TF1* f2 = azim->GetFunction("pol0");
70 f2->SetLineColor(kBlack);
71 azim->SetLineColor(kBlue);
72 azim->Draw();
73 gPad->Update();
74 TPaveStats* ft5 = (TPaveStats*)azim->FindObject("stats");
75 ft5->SetOptFit(optarg);
76 c2->Print("../fit/anglesfit.pdf");
77
78 // Check the entrance for each histogram
79
80 TH1F* tmom = (TH1F*)c->Get("tmomentum");
81 TH1F* imass = (TH1F*)c->Get("invmass");
82 TH1F* en = (TH1F*)c->Get("energy");
83 TH1F* imd = (TH1F*)c->Get("invmassdis");
84 TH1F* imc = (TH1F*)c->Get("invmasscon");
85 TH1F* inv1 = (TH1F*)c->Get("invppkmpmkp");
86 TH1F* inv2 = (TH1F*)c->Get("invppkppmkm");
87 TH1F* dec = (TH1F*)c->Get("decay");
88
89 cout<<"-----\n";
90 cout<<"Entries Momentum: "<< Mom->GetEntries()<<" - "<< 1E7<<'\\n';
91
92 cout<<"-----\n";
93 cout<<"Entries tMoment: "<< tmom->GetEntries()<<" - "<<1E7<<'\\n';
94
95 cout<<"-----\n";

```

```

96     cout<<"Entries invariant mass: "<< imass->GetEntries()<<" - "<<1E5
    *101*100/2<<'\n';
97
98     cout<<"-----\n";
99     cout<<"Entries energy: "<< en->GetEntries()<<" - "<<1E7<<'\n';
100
101     cout<<"-----\n";
102     cout<<"Entries invariant mass opposite: "<< imd->GetEntries()<<" - "
    <<1E5*101*100*0.495/2<<'\n';
103
104     cout<<"-----\n";
105     cout<<"Entries invariant mass concord: "<< imc->GetEntries()<<" - "
    <<1E5*101*100*0.495/2<<'\n';
106
107     cout<<"-----\n";
108     cout<<"Entries invariant mass 1: "<< inv1->GetEntries()<<" - "<<1E5
    *101*100<<'\n';
109
110     cout<<"-----\n";
111     cout<<"Entries invariant mass 2: "<< inv2->GetEntries()<<" - "<<1E5
    *101*100<<'\n';
112
113     cout<<"-----\n";
114     cout<<"Entries decay: "<< dec->GetEntries()<<" - "<<1E7*0.01<<'\n';
115
116     cout<<"-----\n";
117     cout<<"Entries azimuthal angles: "<< azimuth->GetEntries()<<" - "<<1E7<<'\n';
118
119     cout<<"-----\n";
120     cout<<"Entries polar angles: "<< Pol->GetEntries()<<" - " << 1E7<<'\n';
121     cout<<"-----\n";
122
123 }
124
125
126 // Function that analyze the function for the detection of the
    Resonance Kaon ///////////
127
128 void analysis() {
129     int optarg = 1111;
130     TColor* myYellow = gROOT->GetColor(10);
131     TFile* f = new TFile("Alice_Simulation.root","read");
132
133     TCanvas* c3 = new TCanvas("c3", "Cariche");
134     TH1F* Im1 = (TH1F*)f->Get("invmassdis");
135     TH1F* Im2 = (TH1F*)f->Get("invmasscon");
136     TH1F* diff1 = new TH1F("diff1","Difference 1", 1000, 0, 5);
137     diff1->Add(Im1,Im2, -1, 1);
138     diff1->Fit("gaus","","",0.5, 1.5);
139     diff1->SetAxisRange(0.5, 1.5);
140     diff1->Draw();
141     gPad->Update();
142     TPaveStats* ft2 = (TPaveStats*)diff1->FindObject("stats");
143     c3->Print("../fit/Cariche_fit.pdf");
144     ft2->SetOptFit(optarg);
145     ft2->SetOptStat(10);
146

```

```

147 TCanvas* c4 = new TCanvas("c4", "Pioni e Kaoni");
148 TH1F* Im3 = (TH1F*)f->Get("invppkmpmkp");
149 TH1F* Im4 = (TH1F*)f->Get("invppkppmkm");
150 TH1F* diff2 = new TH1F("diff2", "Difference 2", 1000, 0, 5);
151 diff2->Add(Im3, Im4, 1, -1);
152 diff2->Fit("gaus", "", "", 0.7, 1.5);
153 diff2->SetAxisRange(0.7, 1.5);
154 diff2->Draw();
155 gPad->Update();
156 TPaveStats* ft3 = (TPaveStats*)diff2->FindObject("stats");
157 c4->Print("../fit/pion_kaon_fit.pdf");
158 ft3->SetOptFit(optarg);
159 ft3->SetOptStat(10);
160
161 TCanvas* c5 = new TCanvas("c5", "Decadimento Kaoni");
162 TH1F* kaon = (TH1F*)f->Get("decay");
163 kaon->Fit("gaus", "", "", 0.5, 1.25);
164 kaon->SetAxisRange(0.5, 1.5);
165 kaon->Draw();
166 gPad->Update();
167 TPaveStats* ft4 = (TPaveStats*)kaon->FindObject("stats");
168 ft4->SetOptFit(optarg);
169 c5->Print("../fit/decay_fit.pdf");
170 ft4->SetOptStat(2211);
171 }
172

```