Simulazione del Decadimento della Particella K^{0*}

Malagoli Tommaso Andrea , Mazzi Davide , Pivi Riccardo , Schirripa Mattia

Novembre 2024

1 Introduzione

Il codice, scritto con il linguaggio di programazzione C++, simula eventi fisici risultanti da collisioni di particelle elementari. Per la rappresentazione delle particelle risultanti dagli urti sono state implementate tre classi, ciascuna con il proprio omonimo header e source file. L'analisi dati è stata svolta grazie all'utilizzo del pacchetto di analisi dati Root, in modo tale da poter creare grafici sulle generazioni.

2 Struttura del codice

Sono state implementate tre classi:

- ParticleType - descrive le proprietà di base delle particelle elementari: la massa, la carica e il nome.
- ResonanceType - descrive, oltre le proprietà di base, anche la proprietà specifica della risonanza: la larghezza.
- Particle - descrive le proprietà di base e la quantità di moto.

La gerarchia delle classi è strutturata nel seguente modo:

- ResonanceType eredita (relazione is-a) dalla madre ParticleType tutte le sue funzioni e sovrascrive la funzione *void Print()*.
- Particle non eredita nulla dalle due classi precedenti, ma include un *array* di puntatori a ParticleType (relazione has-a), che verrà usata nel codice come lista di inizializzazione di particelle da genereare stocasticamente.

Ogni classe ha caratteristiche comuni e caratteristiche proprie: ciascuna di esse possiede *get functions* da usare in vari metodi, costruttori (alcuni iniziaizzati di default e *ad hoc* per poter conferire il polimorfismo dinamico alle classi) insieme ad altre minori

componenti.

Inoltre, si descrivono di seguito le principali funzioni delle classi utlizzate durante la simulazione:

- *void FindParticle*: un metodo static di Particle che ritorna l'indice (numero che contraddistingue il tipo della particella) di un particolare oggetto di tipo Particle.
- *void PrintParticle*: un metodo di Particle che stampa a terminale le informazioni principali di una particella attraverso l'uso della libreria standard *iostream*.
- *double TotalEnergy* : un metodo di Particle che restituisce l'energia della particella grazie a formule relativistiche.
- *double InvMass* : un metodo di Particle che ritorna il valore dell'invariante di massa.
- *virtual GetWidht*: questa *get-function* è dichiarata *virtual* in ParticleType e sovrascritta in ResonanceType. Nel primo caso ritorna sempre zero, mentre nel secondo caso ritorna l'effettiva larghezza.
- *void Print*: questo metodo è dichiarato *virtual* in ParticleType e sovrascritto in ResonanceType. Esso stampa a schermo le variabili private dei vari oggetti.
- *int Decay2body*: metodo di Particle che assegna alle particelle figlie del decadimento di K^{0^*}(qualora esso sia effettivamente possibile) la corretta quantità di moto.

In seguito sono stati implementati due codici, main_module e macro_analisys che, rispettivamente, creano la generazione delle particelle secondo le modalità successivamente spiegate e analizzano la generazione precedentemente avvenuta.

3 Generazione

L'array statico di puntatori a ParticleType presente nella classe Particle è inizializzato grazie all'utilizzo del metodo statico AddParticleType, inserendo nell'array i seguenti tipi di particelle: $\pi+$, $\pi-$, $k+$, $k-$, $p+$, $p-$, K^{0^*}. La generazione delle particelle è stata gestita tramite due cicli annidati:

1) *ciclo esterno*: genera un numero pari a 10^5 eventi.

2) *ciclo interno*: genera ad ogni evento un array di dimensione *size-safe* riempiendolo con almeno 100 particelle. L'utilizzo di *size-safe* - inizializzato a 120 - serve a garantire che non si verifichi un *segmentation fault* causato dal decadimento delle particelle K^{0^*}.

Ognuna di queste particelle è generata - utilizzando i metodi di generazione Monte Carlo di Root - secondo le probabilità di generazione definite nella Tabella[1].

Nome della particella	Probabilità di generazione
$\pi+$	40%
$\pi-$	40%
$k+$	5%
$k-$	5%
$p+$	4.5%
$p-$	4.5%
K^{0^*}	1%

Tabella 1: Tabella riassuntiva delle probabilità di generazione di ogni tipo di particella: per ogni tipo di particella esiste la controparte di carica opposta eccetto che per particella K^{0^*}.

Ogni istanza della classe Particle (e quindi ogni particella) possiede predefinite caratterstiche che devono essere inizializzate per il corretto funzionamento del codice di generazione; in particolare ogni particella possiede:

- Massa (*Mass***), Carica (*Charge***), Larghezza(*Width***, caratteristica solo delle ResonanceType), Nome (*Name***): queste informazioni vengono identificate grazie ad un indice che permette di individuare uno specifico tipo di particella.
- Quantità di moto (*fP***): viene generato a *run time* il modulo della quantità di moto secondo una distribuzione esponenziale di media 1. Le sue componenti dipendono dall'angolo azimutale e polare.
- Angolo azimutale (*phi***): viene generato a *run time* secondo una distribuzione di probabilità uniforme nell'intervallo [0 ; 2π].
- Angolo polare (*theta***): viene generato a *run time* secondo una distribuzione di probabilità uniforme nell'intervallo [0; π].

4 Analisi

Tramite le classi di Root sono stati creati degli istogrammi che permettono di visualizzare graficamente la generazione e di analizzarne le caratteristiche.
Per quanto concerne la probabilità di generazione dei tipi di particelle le occorrenze osservate risultano essere tutte, eccetto per la particella K^{0^*}, compatibili con quelle

[1](*) viene specificato nella sez. 5.1 *Legenda*
[2](**) viene specificato nella sez. 5.1 *Legenda*

attese dalla teoria: [Tabella 2] e [Figura 1].

Nome della particella	Occorrenze Osservate	Occorrenze Attese
$\pi+$	(4.000 +/- 0.002) x 10^6	4 x 10^6
$\pi-$	(4.000 +/- 0.002) x 10^6	4 x 10^6
$k+$	(0.5007 +/- 0.0007) x 10^6	0.5 x 10^6
$k-$	(0.5007 +/- 0.0007) x 10^6	0.5 x 10^6
$p+$	(0.4500 +/- 0.0007) x 10^6	0.45 x 10^6
$p-$	(0.4501 +/- 0.0007) x 10^6	0.45 x 10^6
K^{0^*}	(0.1008 +/- 0.0003) x 10^6	0.1 x 10^6

Tabella 2: Tabella riassuntiva della generazione dei 10^5 eventi: nella colonna a sinistra il tipo di particella preso in considerazione nella generazione; nella coloonna centrale il numero di occorrenze osservate(numero di particella di tale tipo effettivamente creata dal programma); nella colonna di destra il numero atteso di particelle secondo la tabella delle probabilità Tabella 1.

Oltre al dato sopracitato, il programma esamina anche l'impulso delle N particelle, generato attraverso una distribuzione esponenziale decrescente creata nel programma con una media di $\tau = 1$ GeV. I dati [Tabella 3] e il grafico [in Figura 1, con il titolo impulso] ottenuti durante la generazione sono compatibili con quelli previsti dalla teoria. Analogamente gli angoli indicanti le direzioni delle particelle generatesi in seguito a un decadimento, ovvero l'angolo azimutale ϕ e quello polare θ sono stati generati ambedue attraverso una distribuzione uniforme (fra 0 e 2π il primo e fra 0 e π il secondo), ottenendo risultati compatibili con quelli attesi [Tabella 3] e [Figura 1, sotto il titolo angolo polare e angolo azimutale].
In [Figura 2] sono mostrati 3 istogrammi, tutti relativi a una specifica distribuzione di massa invariante: il primo in alto contiene i soli decadimenti di K^{0^*}, mentre il successivo la differenza fra l'istogramma relativo alle particelle con carica discorde e quello concernente le particelle con carica concorde; e infine quello più in basso riporta la differenza fra l'istogramma attinente alle combinazioni di pioni e kaoni di segno opposto con quello inerente alle combinazioni di pioni e kaoni di segno concorde.
In particolare si può osservare un picco in corrispondenza della massa della risonanza K^{0^*} in entrambi gli ultimi due istogrammi, consistente con il segnale raffigurato nell'istogramma più in alto.

Distribuzione	Parametri Fit	χ^2	DOF	χ^2/DOF
Fit alla distribuzione dell'angolo polare (pol[0])	(5.000 +/- 0.002) x 10^4	222	199	1.16
Fit alla distribuzione dell'angolo azimutale (pol[0])	(3.333 +/- 0.001) x 10^4	325.5	299	1.09
Fit alla distribuzione del modulo dell'impulso (expo)	-1.0005 +/- 0.0005	97.29	98	0.99

Tabella 3: Statistiche dei fit per tre distribuzioni: angolo polare, angolo azimutale e modulo dell'impulso, con i relativi parametri del fit, chi-quadro (χ^2), gradi di libertà (DOF), e chi-quadro ridotto (χ^2/DOF).

Distribuzione Masse Invariante e Fit	Media	Sigma	Ampiezza	χ^2/DOF
K^{0^*} vere (gauss)	(0.8918 ± 0.0002)	(0.05026 ± 0.00011)	$(1.2 \times 10^4 \pm 5 \times 10)$	1.1
Differenza delle combinazioni di carica discorde e concorde (gauss)	(0.894 ± 0.006)	(0.059 ± 0.006)	$(1.1 \times 10^4 \pm 1 \times 10^3)$	1.06
Differenza delle combinazioni π e k di carica discorde e concorde (gauss)	(0.891 ± 0.003)	(0.048 ± 0.002)	$(1.25 \times 10^4 \pm 6 \times 10^2)$	1.02

Tabella 4: Tabella delle distribuzioni della massa invariante di K^{0^*} che fornisce, in ordine, media, sigma, ampiezza e chi-quadro ridotto (χ^2/DOF) dei tre grafici supponendone la loro natura gaussiana.

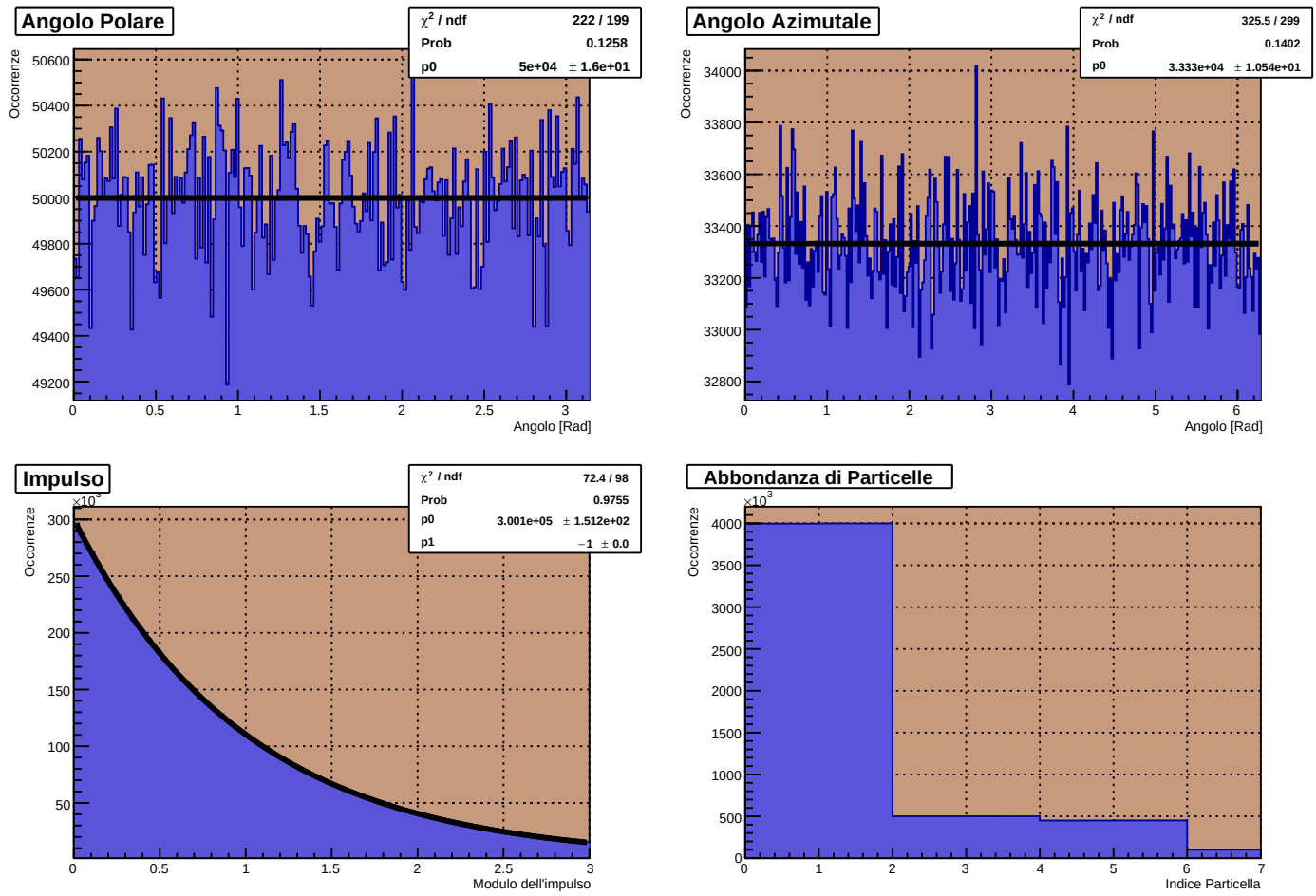

Figura 1: Grafici realizzati con Root raffiguranti l'abbondanza dei tipi di particelle, le distrubuzioni degli angoli polari e azimutali con fit uniforme, la distribuzione esponenziale dell'impulso. Legenda dei bin nell'abbondanza di particelle: $\pi+ = 0$, $\pi- = 1$, $k+ = 2$, $k- = 3$, $p+ = 4$, $p- = 5$, $K^{0^*} = 6$.

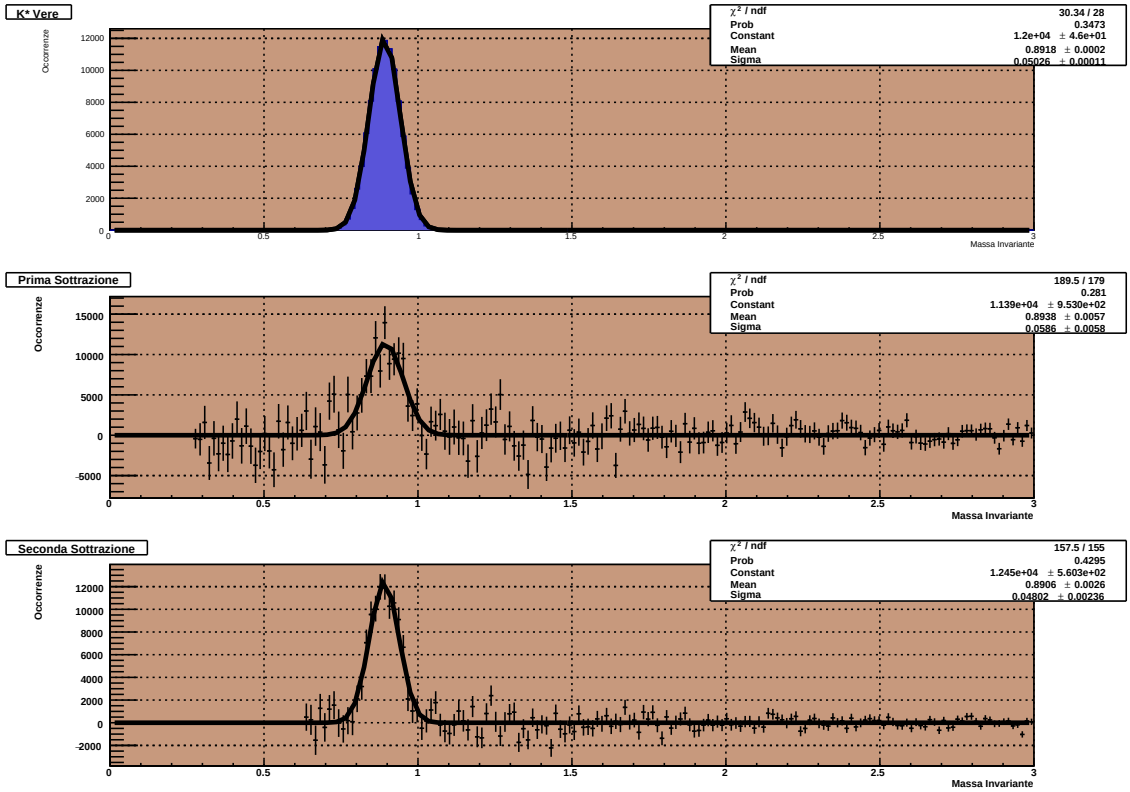

Figura 2: Grafico realizzato con Root raffigurante le masse invarianti con relativi fit gaussiani, tra cui gli ultimi due ottenuti attraverso il metodo della sottrazione della massa invariante di fondo della particella K^{0*}.

5 Appendice

5.1 Legenda

1 (*) = accanto a K^{0*}, è parte del nome della particella K^{0*}.

2 (**) = quando un nome ha vicino due asterischi significa che l'oggetto precedentemente definito viene chiamato in quel modo nel codice.

3 Il listato del codice presenta vari colori che identificano parti specifiche del codice: verde per i commenti, blu per le keyword di C++, rosso per le stringhe.

5.2 Codice

5.2.1 ParticleType.hpp

```
1  #ifndef PARTICLETYPE\_HPP
2
3  #define PARTICLETYPE\_HPP
4
5  #include <iostream>
6
7  class ParticleType
8  {
9  private:
10   char *const fName;
11   double const fMass;
12   int const fCharge;
13
14 public:
15   ParticleType(char *const Name, double const Mass,
16   int const Charge);
17
18   inline char *getName() const
19   {
20     return fName;
21   }
22
23   inline double getMass() const
24   {
25     return fMass;
26   }
27
28   inline int getCharge() const
29   {
30     return fCharge;
31   }
32
33   virtual void Print() const;
34
35   inline virtual double getWidth() const
36   {
37     return 0.;
38   };
39 };
40
41 #endif
```

5.2.2 ParticleType.cpp

```
1
2
3  #include "ParticleType.hpp"
4
5  ParticleType::ParticleType(char* const Name,
6                             double const Mass, int const Charge)
7      : fName{Name}
8      , fMass{Mass}
9      , fCharge{Charge} {};
10
11 void ParticleType::Print() const
12 {
13   std::cout << "\n Charge is: " << fCharge << "\n "
14             << "Mass is: " << fMass << "\n "
15             << "Name is: " << *fName << "\n";
16 };
```

5.2.3 ResonanceType.hpp

```
1  #ifndef RESONANCETYPE_HPP
2  #define RESONANCETYPE_HPP
3
4  #include "ParticleType.hpp"
5
6  class ResonanceType : public ParticleType
7  {
8  private:
9    double const fWidth;
10
11 public:
12   ResonanceType(char *const Name, double const Mass,
13   int const Charge, double const Width);
14
15   double getWidth() const override
16   {
17     return fWidth;
18   }
19
20   void Print() const override;
21 };
```

```
22
23 #endif
```

5.2.4 ResonceType.cpp

```
1  #include "ResonanceType.hpp"
2  #include <iostream>
3
4  void ResonanceType::Print() const
5  {
6    ParticleType::Print();
7    std::cout << "Width: " <<fWidth<<"\n";
8  }
9
10 ResonanceType::ResonanceType(char* const Name,
11 double const Mass, int const Charge, double const Width)
12      : ParticleType(Name, Mass, Charge)
13      , fWidth{Width} {};
```

5.2.5 Particle.hpp

```
1  #ifndef PARTICLE_HPP
2  #define PARTICLE_HPP
3  #include "ParticleType.hpp"
4  #include "ResonanceType.hpp"
5  #include <algorithm>
6  #include <array>
7  #include <iterator>
8  #include <cstring>
9
10 class Particle
11 {
12 private:
13   std::array<double, 3> fP{0., 0., 0.};
14   int fIndex{0};
15   static int fNParticleType;
16   static const int fMaxNumParticleType{10};
17   static std::array<ParticleType*,fMaxNumParticleType>
18   fParticleType;
19
20   static int FindParticle(char *Name)
```

```
21    //it searches for the index of a specific ParticleType
22    //inside fParticleType array
23    //used in several method
24    {
25      if (fNParticleType == 0)
26      {
27        return -1;
28      }
29      else
30      {
31        for (int i = 0; i < fNParticleType; ++i)
32        {
33          if ((fParticleType[i]->getName()) == Name)
34          {
35            return i;
36          }
37        }
38        return -1;
39      }
40    }
41
42    void Boost(double bx, double by, double bz);
43    //used in decay2body
44
45  public:
46    Particle() = default;
47
48    Particle(char *Name, std::array<double, 3> P);
49
50    int getIndex()
51    {
52      return fIndex;
53    }
54
55    static void AddParticleType(char *const Name,
56    double const Mass, int const Charge, double const Width = 0.);
57
58    void setIndex(int codex);
59    void setIndex(char *Name);
60
61    void PrintParticleType() const;
62
63    void PrintParticle() const;
```

```
64
65    inline double getPx() const
66    {
67      return fP[0];
68    };
69
70    inline double getPy() const
71    {
72      return fP[1];
73    };
74
75    inline double getPz() const
76    {
77      return fP[2];
78    };
79
80    inline double getMass() const
81    {
82      return fParticleType[fIndex]->getMass();
83    };
84
85    inline int getCharge() const
86    {
87      return fParticleType[fIndex]->getCharge();
88    };
89
90    inline static int getNparticleType()
91    {
92      return fNParticleType;
93    }
94
95    double NormP() const;
96
97    double TotalEnergy() const;
98
99    double InvMass(Particle &p);
100
101   void setP(double px, double py, double pz);
102
103   static void ClearParticleTypes();
104
105   int Decay2body(Particle &dau1, Particle &dau2) const;
106 };
```

```
107 #endif
```

5.2.6 Particle.cpp

```
1  #include "Particle.hpp"
2  #include <cmath>
3  #include <iostream>
4  #include <array>
5
6  int Particle::fNParticleType{0};
7
8  std::array<ParticleType *, 10> Particle::fParticleType;
9
10 Particle::Particle(char *Name, std::array<double, 3> P)
11     : fIndex{FindParticle(Name)}, fP(P)
12 {
13   if (fIndex == -1)
14   {
15     std::cout << "No particle with this name" << '\n';
16   }
17 }
18
19 void Particle::AddParticleType(char *const Name,
20 double const Mass, int const Charge, double const Width)
21 //it adds a new ParticleType to the array fParticleType
22 {
23   if ((fNParticleType < fMaxNumParticleType) &&
24   (FindParticle(Name) == -1))
25   {
26     if (Width > 0)
27     {
28       ResonanceType *r = new ResonanceType(Name, Mass,
29       Charge, Width);
30       fParticleType[fNParticleType] = r;
31       ++fNParticleType;
32     }
33     else
34     {
35       ParticleType *p = new ParticleType(Name, Mass,
36       Charge); // dynamic allocation
37       fParticleType[fNParticleType] = p;
38       ++fNParticleType;
```

```
39      }
40    }
41  }
42
43  void Particle::setIndex(int Index)  //overloaded
44  {
45    if (Index < fNParticleType)
46    {
47      fIndex = Index;
48    }
49    else
50    {
51      std::cout << "There is no particle in that Index \n";
52    }
53  }
54
55  void Particle::setIndex(char *Name) //overloaded
56  {
57    fIndex = Particle::FindParticle(Name);
58  }
59
60  void Particle::PrintParticleType() const
61  {
62    for (int i = 0; i < fNParticleType; ++i)
63    {
64      fParticleType[i]->Print();
65      std::cout << "\n";
66    }
67  }
68
69  void Particle::PrintParticle() const
70  {
71    if (fIndex != -1 && fIndex < fNParticleType)
72    {
73      std::cout << "Index: " << fIndex << "\n";
74      std::cout << "Name: "
75      << *(fParticleType[fIndex]->getName()) << "\n";
76      std::cout << "Momentum x: " << fP[0] << "\n"
77                << "Momentum y: " << fP[1] << "\n"
78                << "Momentum z: " << fP[2] << "\n";
79    }
80    else
81    {
```

```
82      std::cout << "\n The Particle has no
83      data about Mass, Charge, Width!";
84    }
85  }
86
87  double Particle::NormP() const
88  {
89    double NormP = std::sqrt(std::pow(this->getPx(), 2) +
90    std::pow(this->getPy(), 2) + std::pow(this->getPz(), 2));
91    return NormP;
92  }
93
94  double Particle::TotalEnergy() const
95  {
96
97    if (fIndex != -1 && fIndex < fNParticleType)
98    {
99      double E = std::sqrt(std::pow(this->getMass(), 2) +
100     std::pow(this->NormP(), 2));
101     return E;
102   }
103   else
104   {
105     std::cout << "\n The Particle has no data about
106     Mass, Charge, Width! TotalEnergy return -1";
107     return -1;
108   }
109 }
110
111 double Particle::InvMass(Particle &p)
112 {
113   if ((fIndex != -1) && (fIndex < fNParticleType) &&
114   (p.fIndex != -1) && (p.fIndex < fNParticleType))
115   {
116     double InvM =
117     std::sqrt(std::pow(this->TotalEnergy() + p.TotalEnergy(), 2)
118     - (std::pow(this->getPz() + p.getPz(), 2)  +
119     std::pow( this->getPy() + p.getPy(), 2) +
120     std::pow(p.getPx() + this->getPx(), 2)));
121     return InvM;
122   }
123   else
124   {
```

```
125       std::cout << "\n The Particle has no data about
126       Mass, Charge, Width! InvMass return -1";
127       return -1;
128     }
129   }
130
131   void Particle::setP(double px, double py, double pz)
132   {
133     Particle::fP = {px, py, pz};
134   }
135
136   void Particle::ClearParticleTypes()
137   {
138     for (size_t i = 0; i < size_t(fNParticleType); ++i)
139     {
140       delete fParticleType[i];
141       // deallocation of memory for each pointer
142       fParticleType[i] = nullptr;
143       // sets pointer to nullptr in order to avoid dangling pointers
144     }
145     fNParticleType = 0;
146     // resets the number of elemts
147   }
148
149   int Particle::Decay2body(Particle &dau1, Particle &dau2) const
150   {
151     if (getMass() == 0.0)
152     {
153       std::cout << "Decayment cannot be preformed
154       if mass is zero" << '\n';
155       return 1;
156     }
157
158     double massMot = getMass();
159     double massDau1 = dau1.getMass();
160     double massDau2 = dau2.getMass();
161
162     if (fIndex > -1)
163     { // add width effect
164
165       // gaussian random numbers
166
167       float x1, x2, w, y1;
```

```
168
169       double invnum = 1. / RAND_MAX;
170       do
171       {
172         x1 = 2.0 * rand() * invnum - 1.0;
173         x2 = 2.0 * rand() * invnum - 1.0;
174         w = x1 * x1 + x2 * x2;
175       } while (w >= 1.0);
176
177       w = sqrt((-2.0 * log(w)) / w);
178       y1 = x1 * w;
179
180       massMot += fParticleType[fIndex]->getWidth() * y1;
181     }
182
183     if (massMot < massDau1 + massDau2)
184     {
185       std::cout << "Decayment cannot be preformed because
186       mass is too low in this channel"
187                 << '\n';
188       return 2;
189     }
190
191     double pout =
192         sqrt((massMot * massMot -
193         (massDau1 + massDau2) * (massDau1 + massDau2)) *
194         (massMot * massMot - (massDau1 - massDau2) *
195         (massDau1 - massDau2))) / massMot * 0.5;
196
197     double norm = 2 * M_PI / RAND_MAX;
198
199     double phi = rand() * norm;
200     double theta = rand() * norm * 0.5 - M_PI / 2.;
201     dau1.setP(pout * sin(theta) * cos(phi),
202     pout * sin(theta) * sin(phi), pout * cos(theta));
203     dau2.setP(-pout * sin(theta) * cos(phi),
204     -pout * sin(theta) * sin(phi), -pout * cos(theta));
205
206     double energy = sqrt(std::pow((this->NormP()), 2) +
207     massMot * massMot);
208
209     double bx = fP[0] / energy;
210     double by = fP[1] / energy;
```

```
211     double bz = fP[2] / energy;
212 
213     dau1.Boost(bx, by, bz);
214     dau2.Boost(bx, by, bz);
215 
216     return 0;
217 }
218 
219 void Particle::Boost(double bx, double by, double bz)
220 {
221     double energy = TotalEnergy();
222 
223     // Boost this Lorentz vector
224     double b2 = bx * bx + by * by + bz * bz;
225     double gamma = 1.0 / sqrt(1.0 - b2);
226     double bp = bx * fP[0] + by * fP[1] + bz * fP[2];
227     double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;
228 
229     fP[0] += gamma2 * bp * bx + gamma * bx * energy;
230     fP[1] += gamma2 * bp * by + gamma * by * energy;
231     fP[2] += gamma2 * bp * bz + gamma * bz * energy;
232 }
```

5.2.7 main_module.cpp

```
1  #include "Particle.hpp"
2  #include "TMath.h"
3  #include "TStyle.h"
4  #include "TH1F.h"
5  #include "TF1.h"
6  #include "TCanvas.h"
7  #include "TLegend.h"
8  #include "TFile.h"
9  #include "TMatrixD.h"
10 #include "TFitResult.h"
11 #include "TROOT.h"
12 #include "TRandom.h"
13 #include "TApplication.h"
14 #include <cmath>
15 
16 int main_module()
17 {
```

```
18
19    // setting the seed for ghRandom
20    gRandom->SetSeed();
21    // Recreate the file
22    TFile *file = new TFile("Particles.root", "Recreate");
23
24    // the particles type of the main module
25    Particle::AddParticleType(const_cast<char *>("pion+"),
26    0.13957, 1, 0);      // Index 0
27    Particle::AddParticleType(const_cast<char *>("pion-"),
28    0.13957, -1, 0);     //Index 1
29    Particle::AddParticleType(const_cast<char *>("kaon+"),
30    0.49367, 1, 0);      //Index 2
31    Particle::AddParticleType(const_cast<char *>("kaon-"),
32    0.49367, -1, 0);     //Index 3
33    Particle::AddParticleType(const_cast<char *>("proton+"),
34    0.93827, 1, 0);    //Index 4
35    Particle::AddParticleType(const_cast<char *>("proton-"),
36    0.93827, -1, 0);   //Index 5
37    Particle::AddParticleType(const_cast<char *>("kaon*"),
38    0.89166, 0, 0.050); //Index 6
39
40    // histograms of the events
41    //and generation of the events
42    TH1F *h = new TH1F("h", "histo_particletypes",
43    Particle::getNparticleType(), 0, Particle::getNparticleType());
44    TH1F *h1 = new TH1F("h1", "histo_polar_angle", 200, 0, M_PI);
45    TH1F *h2 = new TH1F("h2", "histo_azimuthal_angle",
46    300, 0, 2 * M_PI);
47    TH1F *h3 = new TH1F("h3", "histo_impulse", 100, 0., 3.);
48    TH1F *h4 = new TH1F("h4", "histo_trasversal_impulse",
49    100, 0., 3.);
50    TH1F *h5 = new TH1F("h5", "histo_energy", 100, 0, 5);
51    TH1F *h6 = new TH1F("h6", "histo_invariant_mass",
52    200, 0., 3.);
53    TH1F *h7 = new TH1F("h7", "histo_invariant_mass_discordant",
54    200, 0., 3.);
55    TH1F *h8 = new TH1F("h8", "histo_invariant_mass_concordant",
56    200, 0., 3.);
57    TH1F *h9 = new TH1F("h9", "histo_invariant_mass_disc_pi_ka",
58    200, 0., 3.);
59    TH1F *h10 = new TH1F("h10", "histo_invariant_mass_conc_pi_ka",
60    200, 0., 3.);
```

```
61    TH1F *h11 = new TH1F("h11", "histo_invariant_mass_decay",
62    200, 0., 3.);
63
64    std::array<TH1F *, 12> hist{h, h1, h2, h3, h4, h5, h6,
65    h7, h8, h9, h10, h11};
66
67    for (int i = 7; i < 11; ++i) //for the analysis
68    {
69      hist[i]->Sumw2();
70    }
71
72    for (int i = 0; i < 12; ++i) //cosmetic
73    {
74      hist[i]->SetFillColor(9);
75    }
76
77    int const Nsafe = 120;
78    int const Nparticles = 100;
79    double x = 0; // used for Rndm
80    for (int i = 0; i < 100000; ++i) //10e5 event
81    {
82      int counter_decay{0};
83      Particle EventParticles[Nsafe];
84      // default constructor used here
85      for (int j = 0; j < Nparticles; ++j)
86      //100 particles for each event
87      {
88        double phi = gRandom->Uniform(0., 2. * M_PI);
89        double theta = gRandom->Uniform(0., M_PI);
90        double modp = gRandom->Exp(1);
91        std::array<double, 3> cartesian_momentum{modp *
92        sin(theta) * cos(phi), modp * sin(theta) * sin(phi),
93        modp * cos(theta)};
94        EventParticles[j].setP(cartesian_momentum[0],
95        cartesian_momentum[1], cartesian_momentum[2]);
96
97        h1->Fill(theta);
98        h2->Fill(phi);
99        h3->Fill(modp);
100       h4->Fill(std::sqrt(std::pow(cartesian_momentum[0], 2) +
101       std::pow(cartesian_momentum[1], 2)));
102
103       x = gRandom->Rndm();
```

```
104         if (x < 0.4)
105         {
106           h->Fill(0);
107           EventParticles[j].setIndex(0);
108         }
109         else if (x < 0.8)
110         {
111           h->Fill(1);
112           EventParticles[j].setIndex(1);
113         }
114         else if (x < 0.85)
115         {
116           h->Fill(2);
117           EventParticles[j].setIndex(2);
118         }
119         else if (x < 0.9)
120         {
121           h->Fill(3);
122           EventParticles[j].setIndex(3);
123         }
124         else if (x < 0.945)
125         {
126           h->Fill(4);
127           EventParticles[j].setIndex(4);
128         }
129         else if (x < 0.99)
130         {
131           h->Fill(5);
132           EventParticles[j].setIndex(5);
133         }
134         else
135         {
136           h->Fill(6);
137           EventParticles[j].setIndex(6);
138           double y = gRandom->Rndm();
139           if (y < 0.5)
140           {
141             EventParticles[100 + 2 * counter_decay].setIndex(0);
142             EventParticles[101 + 2 * counter_decay].setIndex(3);
143             EventParticles[j].Decay2body(EventParticles[100 +
144             2 * counter_decay], EventParticles[101 + 2 * counter_decay]);
145             // filling the histo of the invariant mass of the decay
146             h11->Fill(
```

```
147             EventParticles[100 + 2 * counter_decay].InvMass(EventParticles[
148             101 + 2 * counter_decay]));
149           }
150           else
151           {
152             EventParticles[100 + 2 * counter_decay].setIndex(1);
153             EventParticles[101 + 2 * counter_decay].setIndex(2);
154             EventParticles[j].Decay2body(EventParticles[100 +
155             2 * counter_decay], EventParticles[101 + 2 * counter_decay]);
156             // filling the histo of invariant mass of the decay
157             h11->Fill(
158             EventParticles[100 + 2 * counter_decay].InvMass(EventParticles[
159             101 + 2 * counter_decay]));
160           }
161           ++counter_decay;
162         }
163         // filling the histo of the Energy
164         h5->Fill(EventParticles[j].TotalEnergy());
165       } // out of the generation for of each event
166
167       for (int k = 0; k < Nparticles + 2 * counter_decay; ++k)
168       {
169         if (EventParticles[k].getIndex() == 6)
170         {
171           k++;
172         }
173         for (int j = k + 1; j < Nparticles + 2 * counter_decay; ++j)
174         {
175           if (EventParticles[j].getIndex() == 6)
176           {
177             j++;
178           }
179           // filling the histo of invariant mass
180           //(all the particles of the event)
181           h6->Fill(EventParticles[k].InvMass(EventParticles[j]));
182           if (EventParticles[j].getCharge() *
183           EventParticles[k].getCharge() < 0)
184           {
185             // filling the histo of invariant mass (discordant)
186             h7->Fill(EventParticles[k].InvMass(EventParticles[j]));
187           }
188           if (EventParticles[j].getCharge() *
189           EventParticles[k].getCharge() > 0)
```

```
190           {
191             // filling the histo of invariant mass (concordant)
192             h8->Fill(EventParticles[k].InvMass(EventParticles[j]));
193           }
194           if ((EventParticles[k].getIndex() == 0 &&
195                EventParticles[j].getIndex() == 3) ||
196               (EventParticles[k].getIndex() == 1 &&
197               EventParticles[j].getIndex() == 2) ||
198               (EventParticles[k].getIndex() == 3 &&
199               EventParticles[j].getIndex() == 0) ||
200               (EventParticles[k].getIndex() == 2 &&
201               EventParticles[j].getIndex() == 1))
202           {
203             // filling the histo of invariant mass (discordant pi ka)
204             h9->Fill(EventParticles[k].InvMass(EventParticles[j]));
205           }
206           if ((EventParticles[k].getIndex() == 0 &&
207                EventParticles[j].getIndex() == 2) ||
208               (EventParticles[k].getIndex() == 1 &&
209               EventParticles[j].getIndex() == 3) ||
210               (EventParticles[k].getIndex() == 2 &&
211               EventParticles[j].getIndex() == 0) ||
212               (EventParticles[k].getIndex() == 3 &&
213               EventParticles[j].getIndex() == 1))
214           {
215             // filling the histo of invariant mass (concordant pi ka)
216             h10->Fill(EventParticles[k].InvMass(EventParticles[j]));
217           }
218         }
219       }
220     } // out of the for of the 10e5 events
221     // Writing all the histograms and than closing the file
222     h->Write();
223     h1->Write();
224     h2->Write();
225     h3->Write();
226     h4->Write();
227     h5->Write();
228     h6->Write();
229     h7->Write();
230     h8->Write();
231     h9->Write();
232     h10->Write();
```

```
233     h11->Write();
234
235     file->Close();
236
237     // destruction of the ParticleType
238     Particle::ClearParticleTypes();
239
240     return 0;
241 }
```

5.2.8 macro_analysis.C

```
1  #include "TCanvas.h"
2  #include "TF1.h"
3  #include "TFile.h"
4  #include "TFitResult.h"
5  #include "TH1F.h"
6  #include "TLegend.h"
7  #include "TMath.h"
8  #include "TMatrixD.h"
9  #include "TROOT.h"
10 #include "TRandom.h"
11 #include "TStyle.h"
12 #include <array>
13 #include <cmath>
14 #include <iostream>
15
16 void analysis()
17 {
18
19    // cosmetic
20    gROOT->SetStyle("Plain");
21    gStyle->SetPalette(57);
22    gStyle->SetOptFit(1111);
23    gStyle->SetOptStat(0);
24
25    // reading from the data of the simulation
26    TFile *file = new TFile("Particles.root", "READ");
27    TH1F *h = (TH1F *)file->Get("h");
28    TH1F *h1 = (TH1F *)file->Get("h1");
29    TH1F *h2 = (TH1F *)file->Get("h2");
30    TH1F *h3 = (TH1F *)file->Get("h3");
```

```
31    TH1F *h4 = (TH1F *)file->Get("h4");
32    TH1F *h5 = (TH1F *)file->Get("h5");
33    TH1F *h6 = (TH1F *)file->Get("h6");
34    TH1F *h7 = (TH1F *)file->Get("h7");
35    TH1F *h8 = (TH1F *)file->Get("h8");
36    TH1F *h9 = (TH1F *)file->Get("h9");
37    TH1F *h10 = (TH1F *)file->Get("h10");
38    TH1F *h11 = (TH1F *)file->Get("h11");
39
40    // the entris of each histo
41    std::array<TH1F *, 12> hist{h, h1,
42    h2, h3, h4, h5, h6, h7, h8, h9, h10, h11};
43    for (int i = 0; i < 12; ++i)
44    {
45      std::cout << "For the " << i
46      << "th histogram the number of entries is: "
47                << hist[i]->GetEntries() << "\n";
48    }
49
50    // the content of each bin of h, the histo of the partycles types
51    for (int i = 1; i < 8; ++i)
52    {
53      std::cout << "For the " << i
54      << "th bin, of the numbers of particles,the number of entries is:"
55                << h->GetBinContent(i) << " " << "+/-" << " "
56                << h->GetBinError(i) << '\n';
57    }
58
59    // fitting the Azimutal and Polar Angles with a costant fuction
60
61    TF1 *f1 = new TF1("f1", "[0]", 0, M_PI);
62    h1->Fit(f1);
63    std::cout << "\n First Fit: \n The parameter is:"
64    << f1->GetParameter(0) << " " << "+/-"
65              << " " << f1->GetParError(0)
66              << " , the probability is: " << f1->GetProb()
67              << " and the reduced chisquare:"
68              << f1->GetChisquare() / f1->GetNDF() << '\n';
69
70    TF1 *f2 = new TF1("f2", "[0]", 2 * M_PI);
71    h2->Fit(f2);
72    std::cout << "\n Second Fit: \n The parameter is:"
73    << f2->GetParameter(0) << " " << "+/-" << " "
```

```
74              << f2->GetParError(0) <<
75              " , the probability is: " << f2->GetProb()
76              << " and the reduced chisquare:" <<
77              f2->GetChisquare() / f2->GetNDF() << '\n';
78
79
80    // fitting the Impulse with an exponenitial
81
82    TF1 *f3 = new TF1("f3", "[0]*exp([1]*x)", 0, 5);
83    h3->Fit(f3);
84    std::cout << "\n Third Fit: \n The mean is:"
85    << h3->GetMean() << " +/- " << h3->GetMeanError()
86    << "\n the parameter 0:" << f3->GetParameter(0) << " "
87              << "\n the parameter 1:" << f3->GetParameter(1) << " "
88              << ", the probability is: " << f3->GetProb()
89              << " and the reduced chisquare:"
90              << f3->GetChisquare() / f3->GetNDF() << '\n';
91
92
93     // creating the histograms of the subtractions:
94     // discordant & concordant, pion+ kaon- & pion- kaon+
95    TH1F *hSub1 = new TH1F("hSub1", "histo_subtraction_1", 200, 0., 3.);
96    TH1F *hSub2 = new TH1F("hSub2", "histo_subtraction_2", 200, 0., 3.);
97
98    hSub1->Sumw2();
99    hSub2->Sumw2();
100
101   hSub1->Add(h7, h8, 1, -1);
102   hSub2->Add(h9, h10, 1, -1);
103
104   // setting the correct number of entris
105   hSub1->SetEntries(std::abs(h7->GetEntries() - h8->GetEntries()));
106   hSub2->SetEntries(std::abs(h7->GetEntries() - h8->GetEntries()));
107
108   // fitting with a gaussian
109   TF1 *f4 = new TF1("f4", "gaus", 0., 3.);
110   TF1 *f5 = new TF1("f5", "gaus", 0., 3.);
111
112   hSub1->Fit(f4);
113   std::cout << "\n Fourth Fit: \n Mean is "
114   << f4->GetParameter(1) << "+/-" << f4->GetParError(1)
115             << " and the width is "
116             << f4->GetParameter(2) << "+/-" << f4->GetParError(2)
```

```
117             << " , the probability is: " << f4->GetProb()
118             << " and chisquare reduced is: "
119             << f4->GetChisquare() / f4->GetNDF() << '\n';
120
121   hSub2->Fit(f5);
122   std::cout << "\n Fifth Fit: \n Mean is "
123   << f5->GetParameter(1) << "+/-" << f5->GetParError(1)
124             << " and the width is "
125             << f5->GetParameter(2) << "+/-" << f5->GetParError(2)
126             << " , the probability is: " << f5->GetProb()
127             << " and chisquare reduced is: "
128             << f5->GetChisquare() / f5->GetNDF() << '\n';
129
130   TCanvas *cCanvas = new TCanvas("Histograms", "Histo");
131   cCanvas->Divide(2, 2);
132
133   cCanvas->cd(1);
134
135   h1->SetTitle("Angolo Polare");
136   h1->GetXaxis()->SetTitle("Angolo [Rad]");
137   h1->GetYaxis()->SetTitle("Occorrenze");
138   h1->SetFillColor(9);
139   gPad->SetGrid();
140   gPad->SetFrameFillColor(44);
141   h1->Draw();
142   f1->Draw("same");
143
144   cCanvas->cd(2);
145   h2->SetTitle("Angolo Azimutale");
146   h2->GetXaxis()->SetTitle("Angolo [Rad]");
147   h2->GetYaxis()->SetTitle("Occorrenze");
148   h2->SetFillColor(9);
149   gPad->SetGrid();
150   gPad->SetFrameFillColor(44);
151   h2->Draw();
152   f2->Draw("same");
153
154   cCanvas->cd(3);
155   h3->SetTitle("Impulso");
156   h3->GetXaxis()->SetTitle("Modulo dell'impulso");
157   h3->GetYaxis()->SetTitle("Occorrenze");
158   h3->SetFillColor(9);
159   gPad->SetGrid();
```

```
160     gPad->SetFrameFillColor(44);
161     h3->Draw();
162     f3->Draw("same");
163 
164     cCanvas->cd(4);
165     h->SetTitle("Abbondanza di Particelle");
166     h->GetXaxis()->SetTitle("Indice Particella");
167     h->GetYaxis()->SetTitle("Occorrenze");
168     h->SetFillColor(9);
169     gPad->SetGrid();
170     gPad->SetFrameFillColor(44);
171     h->Draw();
172 
173     TCanvas *cSubtraction = new TCanvas("Analysis", "analysis");
174     cSubtraction->Divide(1, 3);
175 
176     cSubtraction->cd(1);
177     h11->Fit("gaus", "", "", 0, 3);
178     h11->SetTitle("K* Vere");
179     h11->GetXaxis()->SetTitle("Massa Invariante");
180     h11->GetYaxis()->SetTitle("Occorrenze");
181     h11->SetFillColor(9);
182     gPad->SetGrid();
183     gPad->SetFrameFillColor(44);
184     h11->Draw("same");
185     cSubtraction->cd(2);
186     hSub1->SetTitle("Prima Sottrazione");
187     hSub1->GetXaxis()->SetTitle("Massa Invariante");
188     hSub1->GetYaxis()->SetTitle("Occorrenze");
189     gPad->SetGrid();
190     gPad->SetFrameFillColor(44);
191     hSub1->SetFillColor(9);
192     hSub1->Draw("same");
193     cSubtraction->cd(3);
194     hSub2->SetTitle("Seconda Sottrazione");
195     hSub2->GetXaxis()->SetTitle("Massa Invariante");
196     hSub2->GetYaxis()->SetTitle("Occorrenze");
197     hSub2->SetFillColor(9);
198     gPad->SetGrid();
199     gPad->SetFrameFillColor(44);
200     hSub2->Draw("same");
201     cSubtraction->Update();
202 
```

```
203     // Printing and closing the file of the data of the simulation
204     cSubtraction->Print("Subtraction.gif");
205     cCanvas->Print("Angles.gif");
206 
207     cSubtraction->Print("Subtraction.pdf");
208     cCanvas->Print("Canvas.pdf");
209     file->Close();
210 }
```