GENERAZIONE MONTECARLO DI PARTICELLE E ANALISI DATI

Laboratorio di programmazione ROOT/C++ 2018/2019

Marco Uguccioni

Introduzione

L'esperienza di laboratorio consiste nella realizzazione di un programma di generazione Monte Carlo con l'utilizzo di ROOT, il framework di analisi dati "object oriented" in C++.
Il programma genera 10^5 eventi, ciascuno costituito da più di 100 particelle in numero limitato di tipi secondo determinate proporzioni: Pioni, Kaoni, Protoni e K*. A quest'ultima è riservato un trattamento particolare: poiché si tratta di una particella instabile, essa dispone di una larghezza di risonanza (legata alla vita media) ed è inoltre necessario farla decadere qualora sia proprio questa ad essere generata dal programma. Il decadimento della K* produce un Pione e un Kaone in combinazione di carica opposta. Successivamente alla generazione si effettua un'analisi degli istogrammi ottenuti considerando varie grandezze fisiche: quantità di moto delle particelle, angoli polari e azimutali, energie, masse invarianti. Particolarmente importanti sono queste ultime distribuzioni: da un'opportuna sottrazione di istogrammi è possibile separare il segnale della K* dal fondo di altre particelle, in modo da estrarre i parametri della risonanza per verificare la consistenza dei dati ottenuti dal programma con quelli impostati in fase di generazione.

Struttura del codice

Il codice si compone di tre classi, ciascuna suddivisa in moduli indipendenti di intestazione (.h) ed implementazione (.C), ed un "main module", nonché la parte più consistente del programma, dove si effettuano i cicli di generazione.
La prima classe, ParticleType, descrive le tre proprietà di base di una particella stabile, cioè nome, massa e carica, inserite nella classe come attributi const, in modo che, una volta creata l'istanza, questi attributi siano intrinseci al tipo di particella e non possano più essere modificati. Sono stati inoltre implementati dei getters per gli attributi ed il metodo Print() che permette di stampare in output tali proprietà.
La seconda classe, ResonanceType, è una specializzazione della prima, in quanto aggiunge alle proprietà di base di ParticleType, la proprietà di una particella instabile, cioè la larghezza di risonanza. Proprio per questo motivo si è scelto di utilizzare una relazione di tipo "is a", facendo ereditare pubblicamente la seconda classe dalla prima, aggiungendo il getter opportuno al nuovo attributo e ridefinendo il metodo Print() grazie al polimorfismo del C++ (virtual).
La terza ed ultima classe, Particle, è un'ulteriore specializzazione in quanto aggiunge alle precedenti proprietà le componenti della quantità di moto delle particelle (proprietà cinematiche) .
La strada scelta è quella del rimpiego di codice per composizione, utilizzando quindi un array di puntatori static di tipo ParticleType, comune ad ogni istanza e con la funzione di tabella, cioè contenente le informazioni sui tipi di particelle che verranno generate nel main module: in questo modo si evita di duplicare, visto l'elevato numero di oggetti, le stesse proprietà di base, minimizzando la dispendiosità di memoria del programma. Il meccanismo della classe è il seguente: ciascuna istanza di Particle recupera le sue informazioni di base attraverso un indice (index) che corrisponde alla posizione nella tabella, cioè l'indice i-esimo dell'array. Vengono dunque implementati i metodi statici AddParticleType(), che permette di aggiungere tipi di particelle alla tabella, e FindParticle(), che in base al nome ricevuto nel costruttore, ricerca nella tabella l'indice corrispondente a quello della particella ricevuta per argomento, assegnandolo all'istanza index.
Infine, oltre ai getter che restituiscono l'energia della particella e la massa invariante, è presente il

metodo Decay2Body() che permette di realizzare il decadimento della K* in un Pione e un Kaone. Nel main module, dopo aver creato i tipi di particelle interessate, si crea un array di tipo Particle, inizialmente vuoto, con capacità di 120 particelle (100 di base più le eventuali figlie della K*), popolato casualmente secondo definite proporzioni e sovrascritto ad ogni evento. Per la generazione sono stati utilizzati gli opportuni metodi della classe TRandom di ROOT, e la stessa main function è infine utilizzata per la creazione di opportuni istogrammi per l'analisi dei dati.

Generazione

La generazione consiste in 10^5 eventi, ciascuno costituito da 100 particelle. I tipi generati, secondo definite proporzioni, sono i seguenti: Pioni (+/-) 80%, Kaoni (+/-) 10%, Protoni (+/-) 9%, (prodotte in egual proporzione di carica) e K* (1%).
Anche le proprietà cinematiche da assegnare alle varie particelle dell'array sono create casualmente. Per ogni particella è stata infatti generata una coordinata azimutale phi ed una polare theta, distribuite rispettivamente tra 0-2π e tra 0-π con probabilità uniforme; ed il modulo dell'impulso p, secondo una distribuzione esponenziale di media 1 GeV. Da queste ultime grandezze fisiche è semplice settare le componenti p_x, p_y, p_z dell'impulso, utilizzando la formula del passaggio in coordinate polari sferiche.
Nel particolare caso in cui la particella generata sia proprio la K*, è necessario far decadere la risonanza in un Kaone e un Pione, da aggiungere all'array di particelle, con la proporzione 50% Kaone+, Pione- e 50% Kaone-, Pione+.
Inoltre, durante la generazione, vengono riempiti gli istogrammi relativi alle proprietà delle particelle, ovvero: modulo dell'impulso e impulso trasverso, angolo polare e azimutale, energia e massa invariante (combinazione a due a due). Per quest'ultima si hanno diverse varianti: prima fra tutte le particelle dell'evento, poi fra tutte quelle di stessa carica (e opposta) e fra le combinazioni Pione-Kaone di carica concorde (e discorde). Si popola inoltre, come istogramma di controllo, quello della massa invariante fra i "veri" prodotti della K*.

Analisi

In tabella 1 vengono riportate le abbondanze di particelle generate, ottenute con il metodo TH1F::GetBinContent(int nBin), su un totale di 10^7 ingressi:

Specie	Occorrenze Osservate	Occorrenze Attese
π+	$(4.001 \pm 0.002) \cdot 10^6$	$4.0 \cdot 10^6$
π+	$(3.998 \pm 0.002) \cdot 10^6$	$4.0 \cdot 10^6$
K+	$(5.003 \pm 0.007) \cdot 10^5$	$5.0 \cdot 10^5$
K-	$(5.002 \pm 0.007) \cdot 10^5$	$5.0 \cdot 10^5$
P+	$(4.507 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
P-	$(4.498 \pm 0.007) \cdot 10^5$	$4.5 \cdot 10^5$
K*	$(0.999 \pm 0.003) \cdot 10^5$	$1.0 \cdot 10^5$

Tabella 1- Abbondanze dei tipi di particelle generate. Si nota che il programma genera le varie specie secondo le proporzioni richieste entro gli errori statistici. (Per il grafico si veda Figura 1).

In Tabella 2 vengono invece presentati i risultati dei fit alle distribuzioni degli angoli polari, azimutali e del modulo dell'impulso, con i corrispondenti grafici in Figura 1:

Distribuzione	Parametri del Fit	χ^2	D.O.F.	χ^2/D.O.F
Angolo azimutale phi	(9999 ± 3) p0	1033	999	1.034
Angolo polare theta	(9999 ± 3) p0	1003	999	1.004
Modulo dell'impulso	(1.0 ± 0.0) media EXP	953	998	0.955

Tabella 2- Risultati dei fit alle distribuzioni degli angoli polari, azimutali e del modulo dell'impulso. I dati ottenuti sugli angoli sono consistenti con una distribuzione uniforme (fit con "pol0"), mentre la distribuzione del modulo dell'impulso ha un comportamento compatibile con un esponenziale di media 1 GeV (fit con "expo").

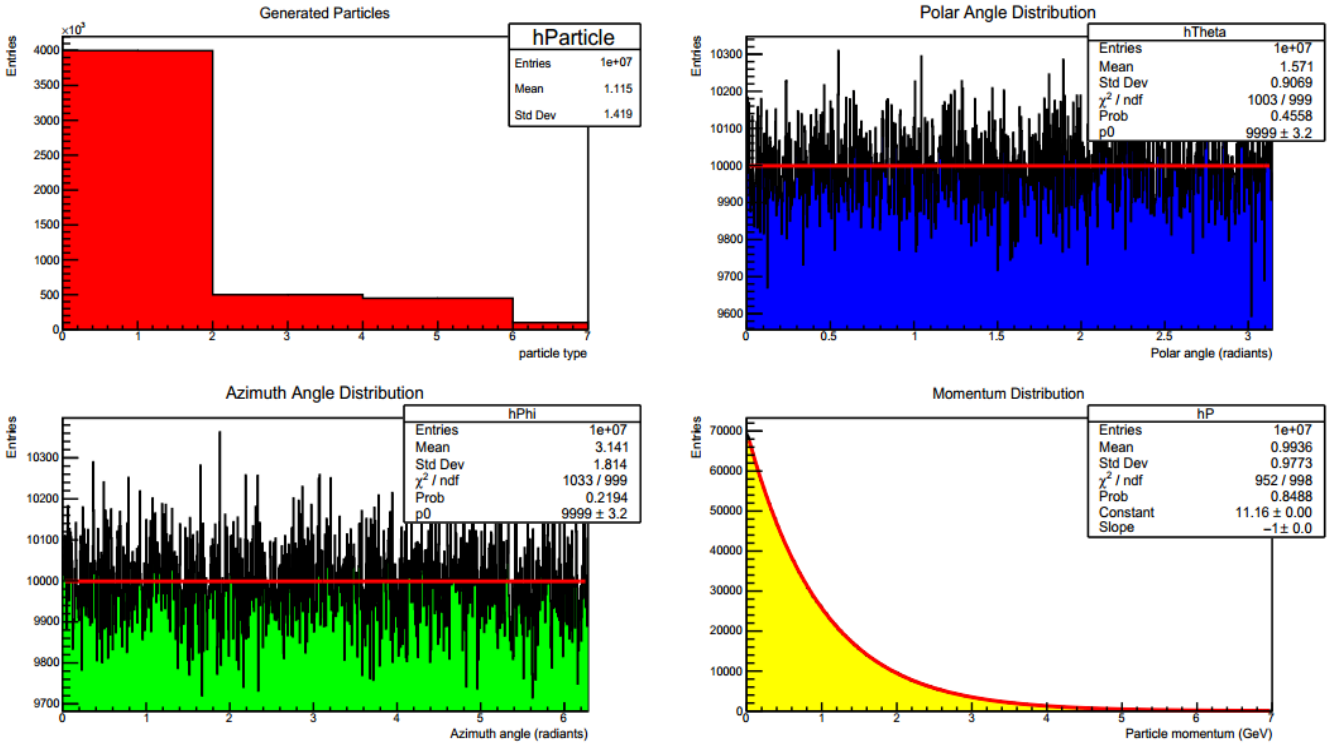

Figura 1- Distribuzioni di abbondanza di particelle (alto a sx), del modulo dell'impulso (basso a dx), angolo polare (alto a dx) e azimutale (basso a sx). Si nota dal box della statistica che i dati ottenuti sono compatibili con le distribuzioni teoriche impostate in fase di generazione. Per l'abbondanza di particelle i bin rappresentano rispettivamente (da sinistra): Pione+, Pione-, Kaone+, Kaone-, protone+, protone- e K.*

Dagli istogrammi di massa invariante è possibile osservare il segnale della K*. Quest'ultimo infatti è un segnale relativamente raro, e il suo "picco" di risonanza è sommerso dal fondo di combinazioni accidentali. Tuttavia, sottraendo l'istogramma di massa invariante tra le particelle di carica concorde (caratterizzato da sole combinazioni accidentali), da quello tra le particelle di carica discorde (combinazioni accidentali e K*), è possibile separare il segnale della K* dal fondo. Un procedimento analogo, ma ancora più efficace, è la stessa sottrazione ma tenendo conto dell'informazione sul tipo di particella, considerando cioè solo combinazioni tra Pioni e Kaoni.

In Tabella 3 vengono presentati i dati ottenuti dall'analisi della massa invariante della particella K*, adattando un fit gaussiano ai dati, mentre in Figura 2 sono riportati i rispettivi grafici:

Distribuzione	Media	Sigma	Ampiezza	χ^2/D.O.F.
Massa Invariante vere K* (gauss)	$(8916\pm 2)\cdot 10^{-4}$ GeV/c^2	$(499.5\pm 1.1)\cdot 10^{-4}$ GeV/c^2	(6380 ± 20)	1.000
Massa Invariante differenza tra cariche discordi e concordi (gauss)	$(8880\pm 50)\cdot 10^{-4}$ GeV/c^2	$(490\pm 50)\cdot 10^{-4}$ GeV/c^2	(6600 ± 500)	1.124
Massa Invariante differenza tra cariche πK discordi e concordi (gauss)	$(8920\pm 30)\cdot 10^{-4}$ GeV/c^2	$(500\pm 30)\cdot 10^{-4}$ GeV/c^2	(6500 ± 300)	1.033

Tabella 3- Analisi massa invariante della K. Nella prima riga si riportano i dati del fit all'istogramma di controllo, cioè dei veri prodotti del decadimento, perfettamente compatibile con una gaussiana. Si nota quindi che i dati impostati in fase di generazione della K* (massa=0.8916 GeV/c^2 e width =0.050 GeV/c^2) sono compatibili con il primo istogramma. Anche negli istogrammi sottrazione si individua una buona compatibilità dei dati con il fit, sebbene gli errori siano più elevati. in particolare, come ci si aspettava, i dati ottenuti dalla differenza Pione-Kaone, in riga 3, sono più precisi rispetto a quelli di riga 2.*

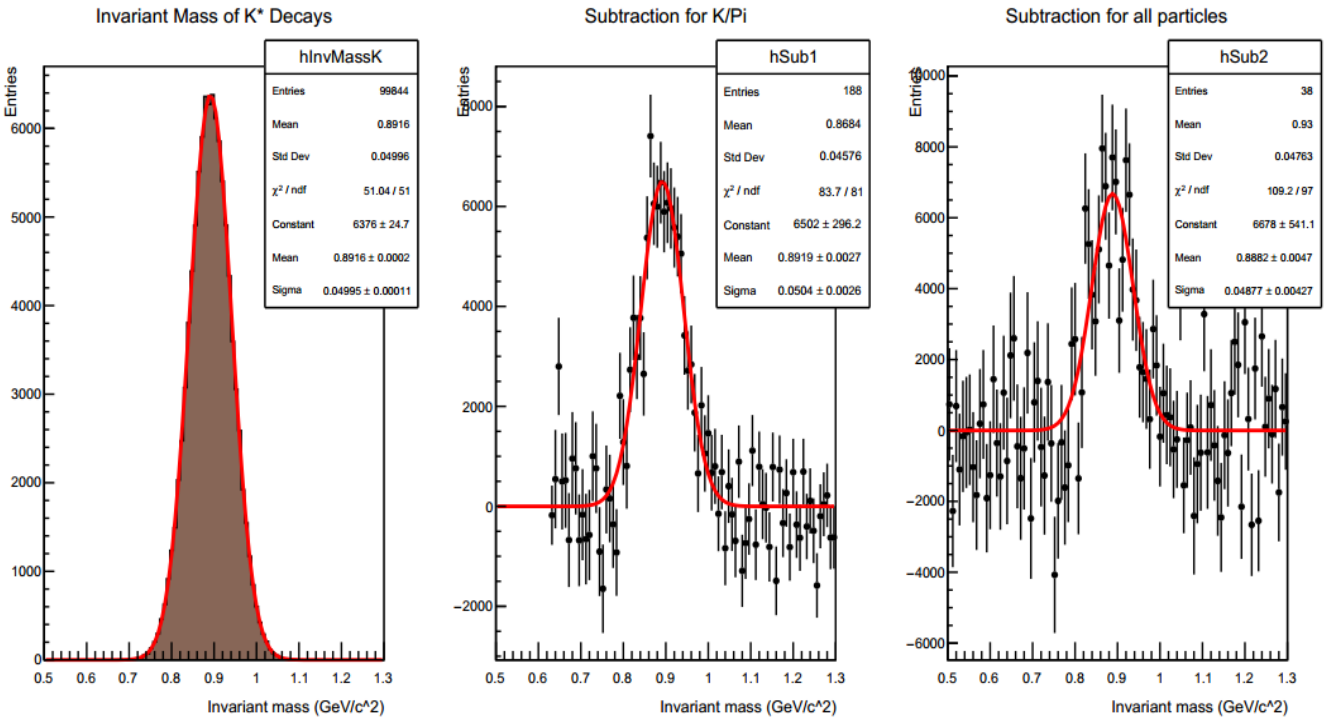

Figura 2 – Distribuzioni di massa invariante della K. Il primo grafico rappresenta la massa dei veri prodotti del decadimento, perfettamente compatibile con il fit gaussiano. Gli ultimi due sono ottenuti da sottrazioni di istogramma con il procedimento elencato sopra. Grazie al metodo TH1F:: Sumw2() si ottengono dati compatibili anche per il secondo e terzo grafico; tuttavia a causa di una limitazione del metodo di sottrazione, non sono calcolate correttamente le entrate nei due grafici. Nonostante ciò, con l'esecuzione del programma senza Sumw2(), si individuano entrate dell'ordine di 10^5, provando quindi che effettivamente sono consistenti con quelle del primo grafico.*

Appendice

Viene allegato in appendice un listato stampato del codice:

PARTICLETYPE.H

```
#ifndef PARTICLETYPE_H
#define PARTICLETYPE_H

class ParticleType
{
 public:
  ParticleType(const char* name,double mass ,int charge );
  const char* GetParticleName() const;
  double GetParticleMass() const;
  int GetParticleCharge() const;
  virtual void Print() const;
  virtual double GetResonanceWidth() const;
 private:
  const char* fName;
  const double fMass;
  const int fCharge;
};

#endif
```

PARTICLETYPE.C

```
#include "particleType.h"
#include <iostream>
using namespace std;

ParticleType :: ParticleType(const char* name,double mass,int charge) :
fName(name),fMass(mass),fCharge(charge) {}

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

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

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

void ParticleType :: Print() const
{
 cout<<"Particle name: "<<fName<<endl;
 cout<<"Particle mass: "<<fMass<<endl;
 cout<<"Particle charge: "<<fCharge<<"\n"<<endl;
}

double ParticleType :: GetResonanceWidth() const
{return 0.0;}
```

RESONANCETYPE.H

```
#ifndef RESONANCETYPE_H
#define RESONANCETYPE_H
#include "particleType.h"

class ResonanceType : public ParticleType
{
 public:
  ResonanceType (const char* name ,double mass,int charge, double width
);
  virtual double GetResonanceWidth() const;
  virtual void Print() const;
 private:
  const double fWidth;
};

#endif
```

RESONANCETYPE.C

```
#include "resonanceType.h"
#include <iostream>
using namespace std;

ResonanceType :: ResonanceType(const char* name,double mass,int charge,
double width) : ParticleType(name,mass,charge), fWidth(width) {}

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

void ResonanceType :: Print() const
{
 ParticleType:: Print();
 cout<<"Resonance width: "<<fWidth<<"\n"<<endl;
}
```

PARTICLE.H

```
#ifndef PARTICLE_H
#define PARTICLE_H
#include "particleType.h"

class Particle
{
 public:
  Particle();
  Particle (const char* name,double px,double py,double pz);
  int GetParticleIndex();
  static void AddParticleType(const char* name,double mass,int
charge,double width=0);
  void SetParticleIndex(int index);
  void SetParticleIndex(const char* name);
  static void PrintParticleArray();
  void Print() const;
```

```
  double GetParticlePx();
  double GetParticlePy();
  double GetParticlePz();
  double GetParticleMass() const;
  int GetParticleCharge() const;
  const char* GetParticleName () const;
  double GetParticleEnergy();
  double GetInvariantMass(Particle& p);
  void SetParticleP (double px,double py,double pz);
  int Decay2body(Particle &dau1,Particle &dau2) const;
  static const int fMaxNumParticleType=10;
 private:
  static ParticleType *fParticleType[fMaxNumParticleType];
  static int fNParticleType;
  int fIndex;
  double fPx,fPy,fPz;
  static int FindParticle (const char* name);
  void Boost(double bx, double by, double bz);
};

#endif
```

PARTICLE.C

```
#include "particleType.h"
#include "resonanceType.h"
#include "particle.h"
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

int Particle :: fNParticleType =0;

ParticleType *Particle :: fParticleType[Particle::fMaxNumParticleType];

Particle :: Particle ()
{fIndex=-1,fPx=0,fPy=0,fPz=0;}

Particle :: Particle (const char* name, double px, double py, double pz)
: fPx(px),fPy(py),fPz(pz)
{
 int index = FindParticle(name);
 if(index!=-1) fIndex=index;
 else cout<<name<<" not found.\n"<<endl,fIndex=-1;
}

int Particle :: FindParticle (const char* name)
{
 for (int i=0 ; i<fNParticleType ; i++)
   if (name == fParticleType[i]->GetParticleName()) return i;
 return -1;
}
```

```
int Particle :: GetParticleIndex()
{return fIndex;}

void Particle :: AddParticleType (const char* name,double mass,int
charge,double width)
{
  if(fNParticleType< fMaxNumParticleType)
  {
   if(FindParticle(name)==-1)
   {
    if (width == 0)
    fParticleType[fNParticleType]=new ParticleType (name,mass,charge);
    else fParticleType[fNParticleType]=new ResonanceType
(name,mass,charge,width);
    fNParticleType++;
    cout<<name<<" added to array.\n"<<endl;
   }
   else cout<<name<<" already exists.\n"<<endl;
  }
  else cout<<"too much particles added, max limit exceeded.\n"<<endl;
}

void Particle :: SetParticleIndex(int index)
{
 if(index <= fNParticleType && index>-1) fIndex=index;
 else cout<<"index not allowed.\n"<<endl;
}

void Particle :: SetParticleIndex(const char* name)
{
 if(FindParticle(name) <= fNParticleType && FindParticle(name)>-1)
  fIndex=FindParticle(name);
 else cout<<"index not allowed.\n"<<endl;
}

void Particle :: PrintParticleArray()
{
 for(int i=0;i<fNParticleType;i++)
 {
  cout<<"particle "<<i<<":"<<endl;
  fParticleType[i]->Print();
 }
}

void Particle :: Print() const
{
 cout<<"Particle index: "<<fIndex<<endl;
 cout<<"Particle name: "<<fParticleType[fIndex]->GetParticleName()<<endl;
 cout<<"Particle momentum vector:
("<<fPx<<","<<fPy<<","<<fPz<<")\n"<<endl;
}

double Particle :: GetParticlePx()
{return fPx;}
```

```
double Particle :: GetParticlePy()
{return fPy;}

double Particle :: GetParticlePz()
{return fPz;}

double Particle :: GetParticleMass () const
{
 return (fParticleType[fIndex]->GetParticleMass());
}

int Particle :: GetParticleCharge () const
{
 return (fParticleType[fIndex]->GetParticleCharge());
}

const char* Particle :: GetParticleName () const
{
 return(fParticleType[fIndex]->GetParticleName());
}

double Particle :: GetParticleEnergy ()
{
 const double mass = fParticleType[fIndex]->GetParticleMass();
 double energy = sqrt
 (mass*mass+(fPx*fPx)+(fPy*fPy)+(fPz*fPz));
 return energy;
}

double Particle :: GetInvariantMass (Particle &p)
{
 double e1= GetParticleEnergy();
 double e2= p.GetParticleEnergy();
 double px= fPx+p.GetParticlePx();
 double py= fPy+p.GetParticlePy();
 double pz= fPz+p.GetParticlePz();
 double invmass=sqrt((e1+e2)*(e1+e2)-(px*px+py*py+pz*pz));
 return invmass;
}

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

int Particle::Decay2body(Particle &dau1,Particle &dau2) const
{
  if(GetParticleMass() == 0.0)
  {
   printf("Decayment cannot be preformed if mass is zero\n");
   return 1;
  }
  double massMot = GetParticleMass();
  double massDau1 = dau1.GetParticleMass();
  double massDau2 = dau2.GetParticleMass();
  if(fIndex > -1) //add width effect
  {
   float x1, x2, w, y1, y2;  //gaussian random numbers
```

```
   double invnum = 1./RAND_MAX;
   do
   {
    x1 = 2.0 * rand()*invnum - 1.0;
    x2 = 2.0 * rand()*invnum - 1.0;
    w = x1 * x1 + x2 * x2;
   }
   while ( w >= 1.0 );
   w = sqrt( (-2.0 * log( w ) ) / w );
   y1 = x1 * w;
   y2 = x2 * w;
   massMot += fParticleType[fIndex]->GetResonanceWidth() * y1;
  }
  if(massMot < massDau1 + massDau2)
  {
   printf("Decayment cannot be preformed because mass is too low in this
channel\n");
   return 2;
  }
  double pout = sqrt((massMot*massMot-
(massDau1+massDau2)*(massDau1+massDau2))*(massMot*massMot-(massDau1-
massDau2)*(massDau1-massDau2)))/massMot*0.5;
  double norm = 2*M_PI/RAND_MAX;
  double phi = rand()*norm;
  double theta = rand()*norm*0.5 - M_PI/2.;

dau1.SetParticleP(pout*sin(theta)*cos(phi),pout*sin(theta)*sin(phi),pout*
cos(theta));
  dau2.SetParticleP(-pout*sin(theta)*cos(phi),-
pout*sin(theta)*sin(phi),-pout*cos(theta));
  double energy = sqrt(fPx*fPx+fPy*fPy+fPz*fPz+massMot*massMot);
  double bx = fPx/energy;
  double by = fPy/energy;
  double bz = fPz/energy;
  dau1.Boost(bx,by,bz);
  dau2.Boost(bx,by,bz);
  return 0;
}


void Particle::Boost(double bx, double by, double bz)
{
  double energy = GetParticleEnergy();
  //Boost this Lorentz vector
  double b2 = bx*bx + by*by + bz*bz;
  double gamma = 1.0 / sqrt(1.0 - b2);
  double bp = bx*fPx + by*fPy + bz*fPz;
  double gamma2 = b2 > 0 ? (gamma - 1.0)/b2 : 0.0;
  fPx += gamma2*bp*bx + gamma*bx*energy;
  fPy += gamma2*bp*by + gamma*by*energy;
  fPz += gamma2*bp*bz + gamma*bz*energy;
}
```

MAIN.C

```
#include "particleType.h"
#include "resonanceType.h"
#include "particle.h"
#include <iostream>
#include <cmath>
#include "TRandom.h"
#include "TH1F.h"
#include "TCanvas.h"
#include "TStyle.h"

using namespace std;


int main()
{
 gStyle->SetOptFit(1111);

 TCanvas *c1=new TCanvas("c1","c1",65,75,700,700);
 c1->Divide(2,2);

 TH1F *hParticle=new TH1F("hParticle","Generated Particles",7,0,7);
 hParticle->SetFillColor(kRed);
 hParticle->SetLineColor(kBlack);
 hParticle->SetMarkerStyle(kFullCircle);
 hParticle->SetMarkerSize(0.8);
 hParticle->GetYaxis()->SetTitleOffset(1.2);
 hParticle->GetXaxis()->SetTitleSize(0.04);
 hParticle->GetYaxis()->SetTitleSize(0.04);
 hParticle->GetXaxis()->SetTitle("particle type");
 hParticle->GetYaxis()->SetTitle("Entries");

 TH1F *hTheta=new TH1F("hTheta","Polar Angle Distribution",1000,0,M_PI);
 hTheta->SetFillColor(kBlue);
 hTheta->SetLineColor(kBlack);
 hTheta->SetMarkerStyle(kFullCircle);
 hTheta->SetMarkerSize(0.8);
 hTheta->GetYaxis()->SetTitleOffset(1.2);
 hTheta->GetXaxis()->SetTitleSize(0.04);
 hTheta->GetYaxis()->SetTitleSize(0.04);
 hTheta->GetXaxis()->SetTitle("Polar angle (radiants)");
 hTheta->GetYaxis()->SetTitle("Entries");

 TH1F *hPhi= new TH1F("hPhi","Azimuth Angle Distribution",1000,0,2*M_PI);
 hPhi->SetFillColor(kGreen);
 hPhi->SetLineColor(kBlack);
 hPhi->SetMarkerStyle(kFullCircle);
 hPhi->SetMarkerSize(0.8);
 hPhi->GetYaxis()->SetTitleOffset(1.2);
 hPhi->GetXaxis()->SetTitleSize(0.04);
 hPhi->GetYaxis()->SetTitleSize(0.04);
 hPhi->GetXaxis()->SetTitle("Azimuth angle (radiants)");
 hPhi->GetYaxis()->SetTitle("Entries");
```

```
TH1F *hP=new TH1F("hP","Momentum Distribution",1000,0,7);
 hP->SetFillColor(kYellow);
 hP->SetLineColor(kBlack);
 hP->SetMarkerStyle(kFullCircle);
 hP->SetMarkerSize(0.8);
 hP->GetYaxis()->SetTitleOffset(1.2);
 hP->GetXaxis()->SetTitleSize(0.04);
 hP->GetYaxis()->SetTitleSize(0.04);
 hP->GetXaxis()->SetTitle("Particle momentum (GeV)");
 hP->GetYaxis()->SetTitle("Entries");


 TCanvas *cA=new TCanvas("cA","cA",65,75,700,700);
 cA->Divide(2,2);

 TH1F *hPTr=new TH1F ("hPTr","Transverse Momentum
Distribution",1000,0,5);
 hPTr->SetFillColor(kYellow);
 hPTr->SetLineColor(kBlack);
 hPTr->SetMarkerStyle(kFullCircle);
 hPTr->SetMarkerSize(0.8);
 hPTr->GetYaxis()->SetTitleOffset(1.2);
 hPTr->GetXaxis()->SetTitleSize(0.04);
 hPTr->GetYaxis()->SetTitleSize(0.04);
 hPTr->GetXaxis()->SetTitle("Transverse momentum (GeV)");
 hPTr->GetYaxis()->SetTitle("Entries");

 TH1F *hEnergy=new TH1F("hEnergy","Particle Energy
Distribution",1000,0,6);
 hEnergy->SetFillColor(kBlue);
 hEnergy->SetLineColor(kBlack);
 hEnergy->SetMarkerStyle(kFullCircle);
 hEnergy->SetMarkerSize(0.8);
 hEnergy->GetYaxis()->SetTitleOffset(1.2);
 hEnergy->GetXaxis()->SetTitleSize(0.04);
 hEnergy->GetYaxis()->SetTitleSize(0.04);
 hEnergy->GetXaxis()->SetTitle("Particle Energy (GeV)");
 hEnergy->GetYaxis()->SetTitle("Entries");

 TH1F *hTotInvMass=new TH1F("hTotInvMass","Invariant Mass Distribution
for all Particles",100,0.5,1.3);
 hTotInvMass->SetFillColor(kGreen);
 hTotInvMass->SetLineColor(kBlack);
 hTotInvMass->SetMarkerStyle(kFullCircle);
 hTotInvMass->SetMarkerSize(0.8);
 hTotInvMass->GetYaxis()->SetTitleOffset(1.2);
 hTotInvMass->GetXaxis()->SetTitleSize(0.04);
 hTotInvMass->GetYaxis()->SetTitleSize(0.04);
 hTotInvMass->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hTotInvMass->GetYaxis()->SetTitle("Entries");


 TCanvas *cB=new TCanvas("cB","cB",65,75,700,700);
 cB->Divide(2,2);
```

```
 TH1F *hDisInvMass=new TH1F("hDisInvMass","Invariant Mass Distribution
for Opposite Charge Particles",100,0.5,1.3);
 hDisInvMass->SetFillColor(kRed);
 hDisInvMass->SetLineColor(kBlack);
 hDisInvMass->SetMarkerStyle(kFullCircle);
 hDisInvMass->SetMarkerSize(0.8);
 hDisInvMass->GetYaxis()->SetTitleOffset(1.2);
 hDisInvMass->GetXaxis()->SetTitleSize(0.04);
 hDisInvMass->GetYaxis()->SetTitleSize(0.04);
 hDisInvMass->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hDisInvMass->GetYaxis()->SetTitle("Entries");

 TH1F *hConInvMass=new TH1F("hConInvMass","Invariant Mass Distribution
for Same Charge Particles",100,0.5,1.3);
 hConInvMass->SetFillColor(kBlue);
 hConInvMass->SetLineColor(kBlack);
 hConInvMass->SetMarkerStyle(kFullCircle);
 hConInvMass->SetMarkerSize(0.8);
 hConInvMass->GetYaxis()->SetTitleOffset(1.2);
 hConInvMass->GetXaxis()->SetTitleSize(0.04);
 hConInvMass->GetYaxis()->SetTitleSize(0.04);
 hConInvMass->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hConInvMass->GetYaxis()->SetTitle("Entries");

 TH1F *hDisInvMassPiK=new TH1F("hDisInvMassPiK","Invariant Mass
Distribution for K+/Pi- or K-/Pi+",100,0.5,1.3);
 hDisInvMassPiK->SetFillColor(kRed);
 hDisInvMassPiK->SetLineColor(kBlack);
 hDisInvMassPiK->SetMarkerStyle(kFullCircle);
 hDisInvMassPiK->SetMarkerSize(0.8);
 hDisInvMassPiK->GetYaxis()->SetTitleOffset(1.2);
 hDisInvMassPiK->GetXaxis()->SetTitleSize(0.04);
 hDisInvMassPiK->GetYaxis()->SetTitleSize(0.04);
 hDisInvMassPiK->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hDisInvMassPiK->GetYaxis()->SetTitle("Entries");

 TH1F *hConInvMassPiK=new TH1F("hConInvMassPik","Invariant Mass
Distribution for K+/Pi+ or K-/Pi-",100,0.5,1.3);
 hConInvMassPiK->SetFillColor(kBlue);
 hConInvMassPiK->SetLineColor(kBlack);
 hConInvMassPiK->SetMarkerStyle(kFullCircle);
 hConInvMassPiK->SetMarkerSize(0.8);
 hConInvMassPiK->GetYaxis()->SetTitleOffset(1.2);
 hConInvMassPiK->GetXaxis()->SetTitleSize(0.04);
 hConInvMassPiK->GetYaxis()->SetTitleSize(0.04);
 hConInvMassPiK->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hConInvMassPiK->GetYaxis()->SetTitle("Entries");


 TCanvas *c2=new TCanvas("c2","c2",65,75,700,700);
 c2->Divide(3,1);

 TH1F *hInvMassK=new TH1F("hInvMassK","Invariant Mass of K*
Decays",100,0.5,1.3);
 hInvMassK->SetFillColor(28);
 hInvMassK->SetLineColor(kBlack);
 hInvMassK->SetMarkerStyle(kFullCircle);
```

```
 hInvMassK->SetMarkerSize(0.8);
 hInvMassK->GetYaxis()->SetTitleOffset(1.2);
 hInvMassK->GetXaxis()->SetTitleSize(0.04);
 hInvMassK->GetYaxis()->SetTitleSize(0.04);
 hInvMassK->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hInvMassK->GetYaxis()->SetTitle("Entries");

 TH1F *hSub1= new TH1F ("hSub1","Subtraction for K/Pi",100,0.5,1.3);
 hSub1->SetFillColor(kBlue);
 hSub1->SetLineColor(kBlack);
 hSub1->SetMarkerStyle(kFullCircle);
 hSub1->SetMarkerSize(0.8);
 hSub1->GetYaxis()->SetTitleOffset(1.2);
 hSub1->GetXaxis()->SetTitleSize(0.04);
 hSub1->GetYaxis()->SetTitleSize(0.04);
 hSub1->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hSub1->GetYaxis()->SetTitle("Entries");

 TH1F *hSub2= new TH1F ("hSub2","Subtraction for all
particles",100,0.5,1.3);
 hSub2->SetFillColor(kYellow);
 hSub2->SetLineColor(kBlack);
 hSub2->SetMarkerStyle(kFullCircle);
 hSub2->SetMarkerSize(0.8);
 hSub2->GetYaxis()->SetTitleOffset(1.2);
 hSub2->GetXaxis()->SetTitleSize(0.04);
 hSub2->GetYaxis()->SetTitleSize(0.04);
 hSub2->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
 hSub2->GetYaxis()->SetTitle("Entries");


 Particle::AddParticleType("Pi+",0.13957,1,0);
 Particle::AddParticleType("Pi-",0.13957,-1,0);
 Particle::AddParticleType("K+",0.49367,1,0);
 Particle::AddParticleType("K-",0.49367,-1,0);
 Particle::AddParticleType("P+",0.93827,1,0);
 Particle::AddParticleType("P-",0.93827,-1,0);
 Particle::AddParticleType("K*",0.89166,0,0.050);

 Particle particle[120];
 double p,px,py,pz;
 double phi,theta;
 double x=0,y=0;
 int alarm=-1;
 int decay=0;
 gRandom->SetSeed();
 double invmassK;

 for(Int_t i=0;i<100000;i++) //10^5 EVENTS CYCLE
 {
   decay=0;

   for(Int_t j=0;j<100;j++) //SINGLE EVENT CYCLE
   {

    p=gRandom->Exp(1);
    hP->Fill(p);
```

```
phi=gRandom->Uniform(0,2*M_PI);
hPhi->Fill(phi);

theta=gRandom->Uniform(0,M_PI);
hTheta->Fill(theta);

px=p*sin(theta)*cos(phi);
py=p*sin(theta)*sin(phi);
pz=p*cos(theta);

hPTr->Fill(sqrt(px*px+py*py));

particle[j].SetParticleP(px,py,pz);

x=gRandom->Rndm();
y=gRandom->Rndm();

if(x<0.8) //GENERATION WITH PROPORTIONS
{
 if(y<0.5)
 particle[j].SetParticleIndex("Pi+"),
 hParticle->Fill(0);
 else
 particle[j].SetParticleIndex("Pi-"),
 hParticle->Fill(1);
}
 else if(x<0.9)
{
 if(y<0.5)
 particle[j].SetParticleIndex("K+"),
 hParticle->Fill(2);
 else
 particle[j].SetParticleIndex("K-"),
 hParticle->Fill(3);
}
else if(x<0.99)
{
 if(y<0.5)
 particle[j].SetParticleIndex("P+"),
 hParticle->Fill(4);
 else
 particle[j].SetParticleIndex("P-"),
 hParticle->Fill(5);
}
else
{
 particle[j].SetParticleIndex("K*");
 hParticle->Fill(6);
 if(y<0.5)
 {
  particle[100+2*decay].SetParticleIndex("Pi+");
  particle[101+2*decay].SetParticleIndex("K-");
 }
 else
 {
  particle[100+2*decay].SetParticleIndex("Pi-");
```

```
      particle[101+2*decay].SetParticleIndex("K+");
     }
     particle[j].Decay2body(particle[100+2*decay],particle[101+2*decay]);

invmassK=particle[100+2*decay].GetInvariantMass(particle[101+2*decay]);
     hInvMassK->Fill(invmassK);
     decay++;

    } //END OF GENERATION WITH PROPORTIONS

    hEnergy->Fill(particle[j].GetParticleEnergy());

   }//END OF EVENT


    for(int k=0;k<100+2*decay;k++) //INVARIANT MASS COMBINATIONS
    {
     for(int l=1+k;l<100+2*decay;l++)
     {
      hTotInvMass->Fill(particle[k].GetInvariantMass(particle[l]));
      if(
particle[k].GetParticleCharge()*particle[l].GetParticleCharge()<0 )
      {
        hDisInvMass->Fill(particle[k].GetInvariantMass(particle[l]));
        if( particle[k].GetParticleMass()+particle[l].GetParticleMass()
== (0.49367 + 0.13957) )
        hDisInvMassPiK->Fill(particle[k].GetInvariantMass(particle[l]));
      }
      else if(
particle[l].GetParticleCharge()*particle[k].GetParticleCharge()>0 )
      {
        hConInvMass->Fill(particle[k].GetInvariantMass(particle[l]));
        if( particle[k].GetParticleMass()+particle[l].GetParticleMass()
== (0.49367 + 0.13957) )
        hConInvMassPiK-> Fill(particle[k].GetInvariantMass(particle[l]));
      }
     }
   } //END OF INVARIANT MASS COMBINATIONS

 } //END OF 10^5 EVENTS

 c1->cd(1);
 hParticle->Draw();
 for(Int_t i=0;i<7;i++)
 cout<<"Generated Particles content, bin "<<i+1<<": ("<<hParticle-
>GetBinContent(i+1)<<" +/- "<<hParticle->GetBinError(i+1)<<")"<<endl;

 c1->cd(2);
 hTheta->Fit("pol0"),cout<<endl;
 hTheta->Draw();

 c1->cd(3);
 hPhi->Fit("pol0"),cout<<endl;
 hPhi->Draw();

 c1->cd(4);
 hP->Fit("expo"),cout<<endl;
```

```
hP->Draw();

cA->cd(1);
hPTr->Draw();

cA->cd(2);
hEnergy->Draw();

cA->cd(3);
hTotInvMass->Draw();

cB->cd(1);
hDisInvMass->Draw();

cB->cd(2);
hConInvMass->Draw();

cB->cd(3);
hDisInvMassPiK->Draw();

cB->cd(4);
hConInvMassPiK->Draw();

c2->cd(1);
hInvMassK->Fit("gaus");
hInvMassK->Draw();

c2->cd(2);
hSub1->Sumw2();
hSub1->Add(hDisInvMassPiK,hConInvMassPiK,1,-1);
hSub1->Fit("gaus","","",0.7,1.1),cout<<endl;
hSub1->Draw();

c2->cd(3);
hSub2->Sumw2()
hSub2->Add(hDisInvMass,hConInvMass,1,-1);
hSub2->Fit("gaus","","",0.7,1.1),cout<<endl;
hSub2->Draw();

} //END OF MAIN FUNCTION
```