SIMULATING PARTICLE COLLISIONS WITH C++ AND ROOT

Laboratory report

Jaime Bruno and Marta Aznar

INDEX

INTRODUCTION

We have developed a project in C++ with the following functionalities:

- Simulate many physical events of elementary particles collisions, belonging to a limited number of types ($\pi^+, \pi^-, K^+, K^-, P^+, P^-, K^*$) using Monte Carlo generation methods.
- Represent and analyze the data acquired from the simulation, obtaining the distributions of the composition of the particle beam and of some kinematic characteristic and dynamic quantities, as well as the distributions of invariant mass between various combinations of particles.

In particle physics, resonances are identified through the invariant mass of the decay products. The final aim of the analysis, in addition to verifying the correctness of the distributions generated, was therefore the extraction of the signal of the K^* particle, unstable, decaying into π^+K^- or π^-K^+.

1. CODE STRUCTURE

The program is based on a previous implementation of three classes: *ParticleType*, *ResonanceType* and *Particle*. For each one, the code is divided into a header file (*.h*) and an implementation file (*.cxx*). These classes are used in the Monte Carlo generation managed by the main module (*main.cxx*). Data representation and analysis are carried out by an independent macro, *analysis.cxx*.

1.1. ParticleType

It describes the three basic properties of a stable particle: name, mass and charge, represented as const type attributes, as they cannot vary because they identify the particle type.

1.2. ResonanceType

It describes the four basic properties of an unstable particle (or resonance): name, mass, charge and width (this last one linked to the average lifetime of the particle, this relationship is given by the following expression $\Gamma = \frac{\hbar}{\tau}$, where Γ represents the width of the particle, τ the average lifetime and $\hbar$ is the Dirac constant).

These attributes are declared const for similar reasons as *ParticleType*. Since *ResonanceType* is a specialization of *ParticleType* with an additional attribute, the code reuse mechanism based on virtual inheritance: *ResonanceType* publicly inherits from *ParticleType* (“is-a” relationship).

1.3. Particle

Its instances describe individual particles which, in addition to basic properties, are equipped with kinematic properties, represented by the momentum components Px, Py and Pz.
Inheritance this time would lead to a useless duplication of the basic attributes for a very large number of objects. To save memory, we chose aggregation as a code reuse mechanism: we include in *Particle* as a static member (common to all instances) an array of pointers to *ParticleType* that constitutes a “table” that associates a numerical index (preserved as a private attribute for each *Particle* instance) to each particle type.

2. GENERATION

The simulation involves the generation of 10^5 events, in each of which approximately 120 particles are produced. The first 100 are generated according to defined proportions, as follows:

- 40% π^+ (positive pions).
- 40% π^- (negative pions).
- 5% K^+ (positive kaons).
- 5% K^- (negative kaons).
- 4.5% P^+ (positive protons).
- 4.5% P^- (negative protons).
- 1% K^* (resonance).

The generation according to defined proportions is carried out by extracting a random number from a uniform distribution between 0 and 1 with the *TRandom::Uniform* and *TRandom::Exp* methods of ROOT, we also use *gRandom->SetSeed()* to initialize a new generation seed.

Through a chain of if/else, depending on the generated random number and the percentage previously indicated, the type of the particle will be determined. The remaining particles derived from the decay of K^*, which equiprobably produces the π^+K^- or π^-K^+ pairs.

For the kinematic properties of the first 100 particles, the momentum modulus are generated according to a decreasing exponential distribution with mean 1 GeV (c=1), via *TRandom::Exp(double mean)*. The azimuthal angle φ and the polar angle θ are extracted from uniform distributions in the ranges [0,2π] and [0,π], via *TRandom::Uniform(double xmin, double xmax)*.

The components Px, Py and Pz are then obtained using the formulas for the transition from spherical to cartesian coordinates and set using the *Particle::SetP(Px,Py,Pz)* method. Note that it is fundamental to use independent extractions for φ and θ to obtain independent variables. The kinematic properties of the decay products are instead assigned by the *Particle::Decay2Body()* method, which takes care of decaying the K^*.

At each event, the generated data is used to fill the histograms of the different distributions. The histograms of particle types, polar and azimuthal angles and momentum are filled considering only the first 100 particles. However, those referring to the invariant mass, are considering all the particles (even the daughters), excluding the K^*.

3. ANALYSIS

Firstly, the compatibility of the distributions obtained with the data input to the generation was verified. Considering the histogram of the generation of the particle types (*Fig.1*), and compared with the *Tab.1*, within the statistical errors, there is an excellent correspondence between observed and expected occurrences: the particles are generated according to the required proportions.

For the angular distributions, they were verified to be consistent with uniform distributions across a fit, as well as the exponential behavior of the distribution of the momentum module. The fitted distributions are shown in *Fig.1* and the fit statistics in *Tab.2*. The results obtained are consistent as $\frac{X^2}{DOF}$ is approximately 1 for all three fits. Also, for the uniform distributions we have $P_0 = (9999 \pm 3)$, compatible with the expected value of $\frac{N_{particles}}{N_{bins}} = \frac{100\times100000}{100}$.

The K^* signal is relatively rare: therefore, by carrying out only invariant mass combinations of type π^+K^- or π^-K^+, the resonance peak would be submerged by accidental combinations. Therefore, the signal was extracted by subtracting from the invariant mass distribution among all particles with discordant charge (given by K^*+ accidental combinations) the invariant mass distribution among the particles with concordant charge (only accidental combinations), to eliminate the background. The difference histogram provides the K^* signal.

The procedure is repeated for the πK pairs only for an even more effective result. The signal distribution, a Breit-Wigner, is assumed to be Gaussian for simplicity: a Gaussian fit is carried out on the difference histograms and that of the true K^*, used as a control (*Fig.2*). The fit statistics are reported in *Tab.3:* mean and sigma are compatible with those of the control histogram and provide a correct estimate of the mass ($m_{K^*} = 0.89166\ GeV$) and width ($\Gamma_{K^*} = 0.050$ GeV) of the resonance respectively.

Species	Observed occurrences	Expected occurrences
π^+	$(3998 \pm 2) \times 10^3$	4000×10^3
π^-	$(4001 \pm 2) \times 10^3$	4000×10^3
K^+	$(500.2 \pm 0.7) \times 10^3$	500×10^3
K^-	$(501.3 \pm 0.7) \times 10^3$	500×10^3
P^+	$(451.2 \pm 0.7) \times 10^3$	450×10^3
P^-	$(448.7 \pm 0.7) \times 10^3$	450×10^3
K^*	$(99.9 \pm 0.3) \times 10^3$	100×10^3

Tab.1: Comparison of observed and expected occurrences for each particle species.

Distribution	Fit parameters	X^2	NDF	$\frac{X^2}{NDF}$
Fit to angle distribution θ (pol0)	(2500 ± 8) Zero-degree pol.	422	399	1.06
Fit to angle distribution φ (pol0)	(2500 ± 8) Zero-degree pol.	369	399	0.93
Fit to impulse modulus distribution (expo)	(0.9974 ± 0.0003) Exp mean	437	398	1.1

Tab.2: Fit parameters for the uniform distributions of the θ and φ angles and for the exponential distribution of the pulse modulus. The exponential mean was obtained with the change of sign of the slope parameter shown in Fig.1.

Distribution	Mean	Sigma	Amplitude	$\frac{X^2}{NDF}$
Invariant mass distribution K^* (gauss)	(0.8916 ± 0.0002)	(0.0503 ± 0.0001)	(9905 ± 40)	0.91
Invariant mass obtained from the difference of the combinations of discordant and concordant charges (gauss)	(0.890 ± 0.006)	(0.054 ± 0.006)	(8680 ± 807)	0.98
Invariant mass obtained from the difference of the πK combinations of discordant and concordant charge (gauss)	(0.885 ± 0.003)	(0.048 ± 0.003)	(10003 ± 475)	0.93

Tab.3: Gaussian fit statistics done on the difference and control histograms

Figures

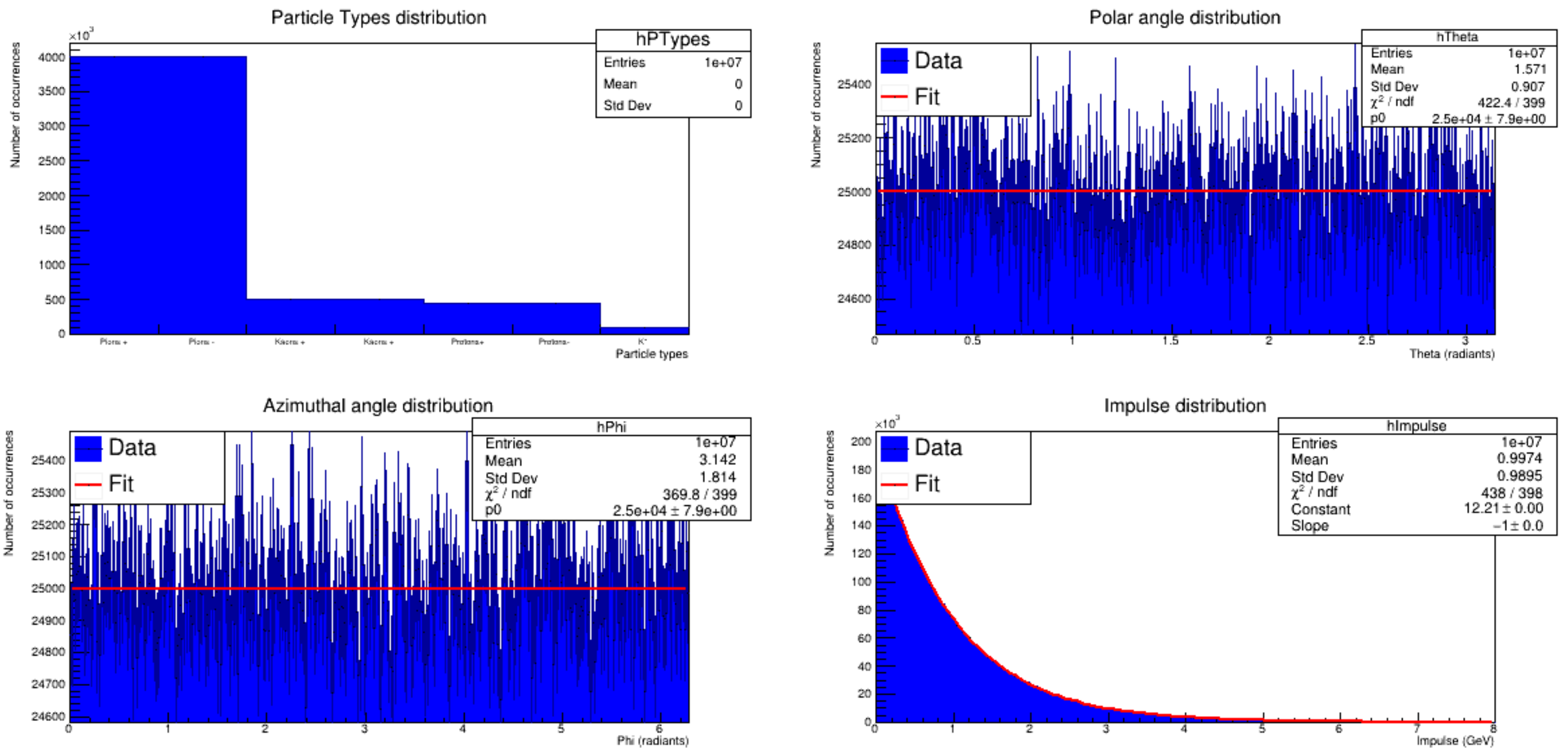

Fig.1: Histograms representing the occurrences of the particle type, the azimuthal and polar angles and the momentum modulus (in blue) with relative fits for the last three (in red). The statistics are reported in the box at the top left.

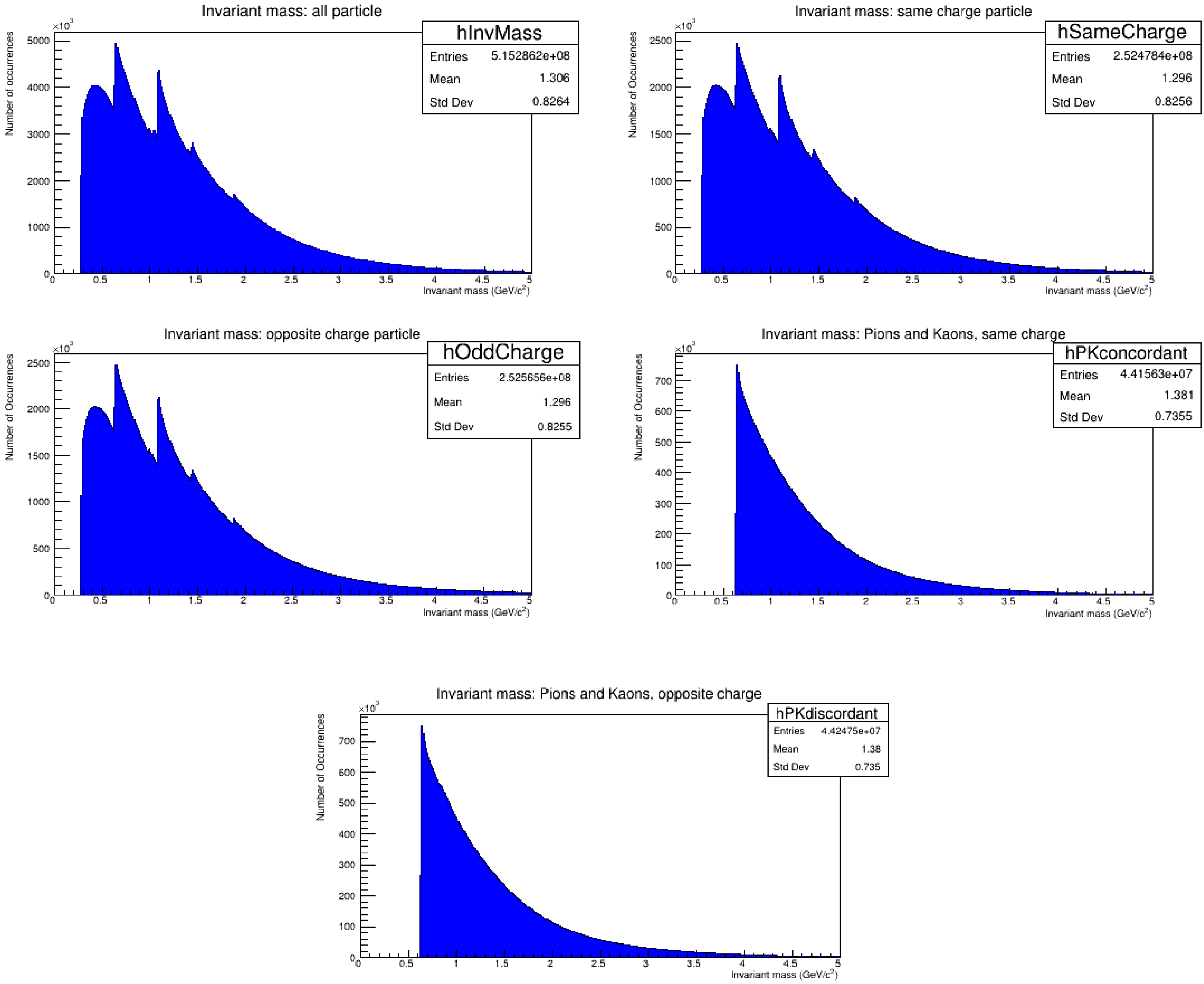

Fig.2: Invariant mass histograms: discordant minus concordant charges, discordant minus concordant πK and the combination between decay products. The occurrences are in blue and the Gaussian fit in red. The invariant masses are expressed in GeV since c=1.

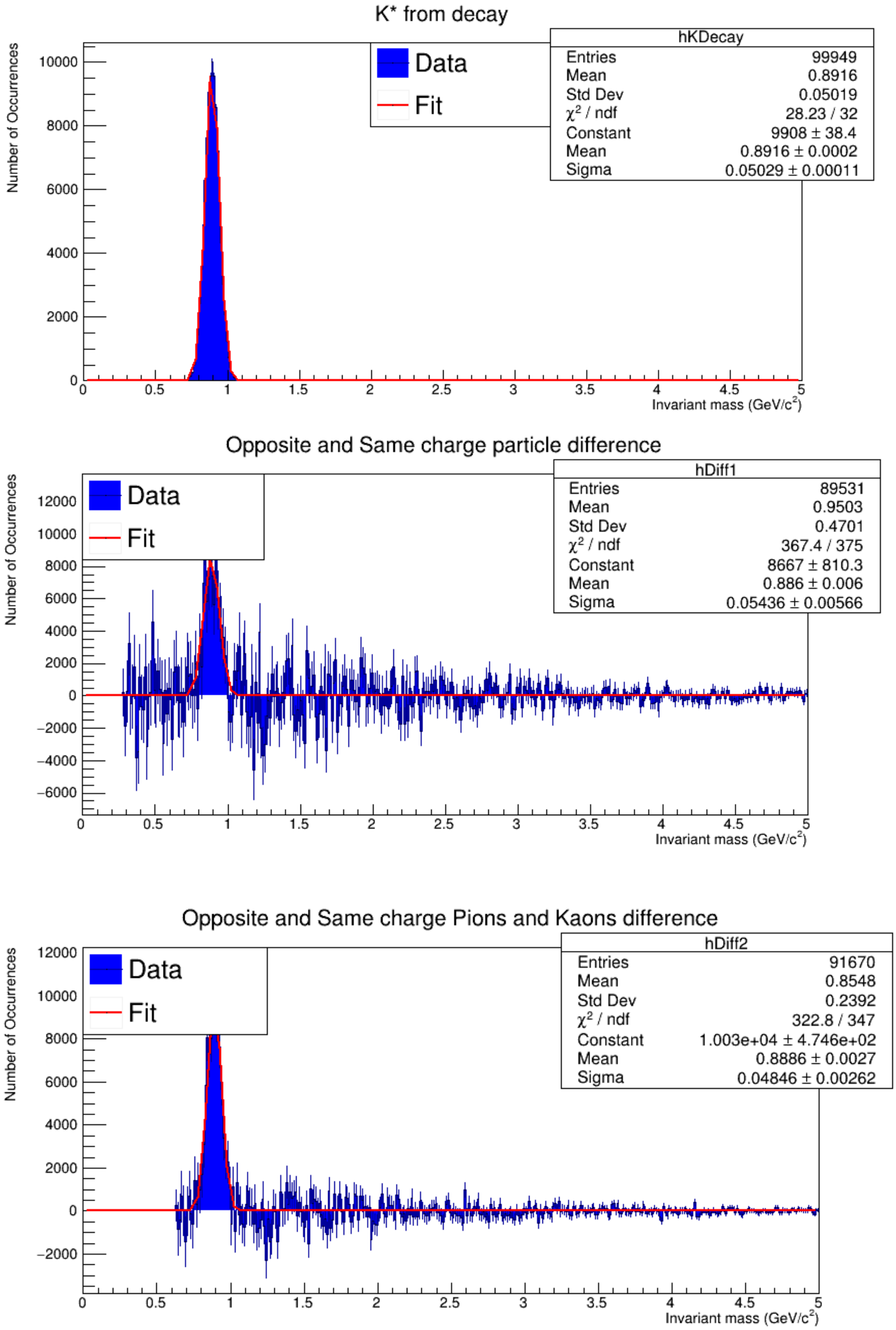

Fig.3: Invariant mass histograms between different combination of particles.

4. CODE LIST

ParticleType.h

```
#ifndef PARTICLETYPE_H
#define PARTICLETYPE_H

#include "iostream"
#include "stdlib.h"

class ParticleType{

        private:

                //atributos
                std::string fName;
                const double fMass;
                const int fCharge;


        public:

                ParticleType( std::string fName_,double fMass_,int fCharge_); //Constructor

                //Métodos
                virtual void Print() const;


                double GetfMass() const;
                int GetfCharge() const;
                std::string GetfName() const;
                virtual double GetWidth() const;

};
#endif
```

ParticleType.cxx

```
#include "ParticleType.h"
using namespace std;

//Inicializamos

ParticleType::ParticleType( std::string fName_,double fMass_,int fCharge_) :
fMass(fMass_), fName(fName_), fCharge(fCharge_){} //constructor

//Métodos

void ParticleType::Print() const{

        cout<<"Particle type: " <<fName<<endl;

        cout<<"Mass:" <<fMass<<endl;

```

```
        cout<<"Charge:" <<fCharge<<endl;
}

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

 std::string ParticleType::GetfName() const{
        return fName;}

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

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

ResonanceType.h

```
#ifndef RESONANCETYPE_H
#define RESONANCETYPE_H

#include "ParticleType.h"

class ResonanceType : public ParticleType {

public:
ResonanceType(std::string  fName_, double fMass_, int fCharge_, double fWidth_);

double GetWidth() const override;

void Print() const override;

private:
const double fWidth;

};
#endif //RESONANCETYPE_H
```

ResonanceType.cxx

```
#include "ResonanceType.h"

ResonanceType::ResonanceType( std::string  fName_, double fMass_,int fCharge_, double
fWidth_):
ParticleType(fName_, fMass_, fCharge_), fWidth(fWidth_) {}

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

void ResonanceType::Print() const {
  ParticleType::Print();
  std::cout << "Particle width " << fWidth << '\n';
}
```

Particle.h

```cpp
#ifndef PARTICLE_H
#define PARTICLE_H

#include "ParticleType.h"
#include "ResonanceType.h"

#include <string>


class Particle {
public:
  Particle();
  Particle(std::string  fName, double Px = 0, double Py = 0, double Pz = 0);

  int GetIndex() const;

  static void AddParticleType(std::string  fName, double fMass, int fCharge, double fWdith
= 0);
  static void AddResonanceType(std::string  fName, double fMass, int fCharge, double
fWidth);

  void SetParticle(int &index);
  void SetParticle(std::string  fName); //overload

  static void PrintArray();
  void PrintParticle();

  double GetXImpulse() const;
  double GetYImpulse() const;
  double GetZImpulse() const;
  double GetMass() const;
  double GetCharge() const;
  double GetEnergy() const;

  double InvMass(Particle & p);

  void SetP(double px,double py,double pz);

  int Decay2body(Particle &dau1,Particle &dau2) const;

private:
  void Boost(double bx, double by, double bz);
  static int FindParticle(std::string  fName);
  static const int fMaxNumParticleType = 10;
  static ParticleType* fParticleType[fMaxNumParticleType];

  static int fNParticleType;
  int fIParticle = 0;
  double Px_ = 0;
  double Py_ = 0;
  double Pz_ = 0;
};

#endif // PARTICLE_H
```

Particle.cxx

```
#include <cmath>
#include <cstdlib> //for RAND_MAX
#include <iostream>

#include "Particle.h"

int constexpr c = 299792458; // m/s

Particle::Particle() = default;

Particle::Particle(std::string  fName, double Px, double Py, double Pz)
    : Px_(Px), Py_(Py), Pz_(Pz) {
  fIParticle = FindParticle(fName);
}

int Particle::fNParticleType = 0;
ParticleType *Particle::fParticleType[fMaxNumParticleType];

int Particle::FindParticle(std::string  fName) {
  int result = -1;
  for (int i = 0; i < fNParticleType; ++i) {
    auto name = fParticleType[i]->GetfName();
    if (name == fName) {
      result = i;
      // particle found
    } else {
    }
  }
  return result;
}

int Particle::GetIndex() const { return fIParticle; }

void Particle::AddParticleType(std::string  fName, double fMass, int fCharge,
                               double fWidth) {

  if (FindParticle(fName) > fNParticleType) {
    std::cout << "Exceeded array capacity. Too many particle types.\n";
  }
  // adding particle identified by fName
  if (fWidth == 0) { // not a resonance
    fParticleType[fNParticleType] = new ParticleType(fName, fMass, fCharge);
    fNParticleType++;
  } else {
    fParticleType[fNParticleType] =
        new ResonanceType(fName, fMass, fCharge, fWidth);
    fNParticleType++;
  }

  std::cout << "Particle " << fName << " successfully added to the array.\n";
}

void Particle::SetParticle(int &index) {
  if (index < 0) {
```

```
    std::cout << "Invalid index value. Try with a positive number.\n";
  } else if (index < fNParticleType) {
    fIParticle = index; // particle index turns into the selected one, if chosen
                        // number is allowed
  } else {
    std::cout << "Invalid index value.\n";
  }
}

//Setting index particle by its name
void Particle::SetParticle(std::string  fName) {
  int index = FindParticle(fName);
  if (index < fNParticleType) {
    fIParticle = index;
  } else {
    std::cout << "Requested value is not in the array.\n";
  }
}

void Particle::PrintParticle() {
  int index = fIParticle;
  std::cout << "Printing particle " << fParticleType[fIParticle]->GetfName()
            << " information...\n";
  std::cout << "Position in the array = " << index << '\n';
  std::cout << "Impulse = "
            << "(" << Px_ << "," << Py_ << "," << Pz_ << ")\n\n";
}

void Particle::PrintArray() {
  for (int i = 0; i < fNParticleType; i++) {
    ParticleType *pt = fParticleType[i];
    std::string  name = fParticleType[i]->GetfName();
    std::cout << "Particle index = " << FindParticle(name) << '\n';
    pt->Print();
  }
}

double Particle::GetXImpulse() const { return Px_; }
double Particle::GetYImpulse() const { return Py_; }
double Particle::GetZImpulse() const { return Pz_; }
double Particle::GetMass() const {
  auto particle = *fParticleType[fIParticle];
  return particle.ParticleType::GetfMass();
}

double Particle::GetCharge() const {
  auto particle = *fParticleType[fIParticle];
  return particle.ParticleType::GetfCharge();
}

double Particle::GetEnergy() const {
  double pSquareModulus =
      std::pow(Px_, 2) + std::pow(Py_, 2) + std::pow(Pz_, 2);
  double pSquareMass = std::pow(GetMass(), 2);
  return std::sqrt(pSquareMass + pSquareModulus);
}
```

```

// invariant mass calculator
double Particle::InvMass(Particle &p) {
  double ImpulseSumSquared = std::pow(Px_ + p.GetXImpulse(), 2) +
                             std::pow(Py_ + p.GetYImpulse(), 2) +
                             std::pow(Pz_ + p.GetZImpulse(), 2);
  return std::sqrt(std::pow(GetEnergy() + p.GetEnergy(), 2) -
                   (ImpulseSumSquared));
}

void Particle::SetP(double px, double py, double pz) {
  Px_ = px;
  Py_ = py;
  Pz_ = pz;
}

int Particle::Decay2body(Particle &dau1, Particle &dau2) const {
  if (GetMass() == 0.0) {
    printf("Decayment cannot be preformed if mass is zero\n");
    return 1;
  }

  double massMot = GetMass();
  double massDau1 = dau1.GetMass();
  double massDau2 = dau2.GetMass();

  if (fIParticle > -1) { // add width effect

    // gaussian random numbers

    float x1, x2, w, y1, y2;

    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[fIParticle]->GetWidth() * 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.SetP(pout * sin(theta) * cos(phi), pout * sin(theta) * sin(phi),
            pout * cos(theta));
  dau2.SetP(-pout * sin(theta) * cos(phi), -pout * sin(theta) * sin(phi),
            -pout * cos(theta));

  double energy =
      sqrt(Px_ * Px_ + Py_ * Py_ + Pz_ * Pz_ + massMot * massMot);

  double bx = Px_ / energy;
  double by = Py_ / energy;
  double bz = Pz_ / energy;

  dau1.Boost(bx, by, bz);
  dau2.Boost(bx, by, bz);

  return 0;
}

void Particle::Boost(double bx, double by, double bz) {

  double energy = GetEnergy();

  // Boost this Lorentz vector
  double b2 = bx * bx + by * by + bz * bz;
  double gamma = 1.0 / sqrt(1.0 - b2);
  double bp = bx * Px_ + by * Py_ + bz * Pz_;
  double gamma2 = b2 > 0 ? (gamma - 1.0) / b2 : 0.0;

  Px_ += gamma2 * bp * bx + gamma * bx * energy;
  Py_ += gamma2 * bp * by + gamma * by * energy;
  Pz_ += gamma2 * bp * bz + gamma * bz * energy;
}
```

main.cxx

```
#include <cmath>
#include <cstdlib>
#include <iostream>

#include "TCanvas.h"
#include "TFile.h"
#include "TH1.h"
#include "TMath.h"
#include "TRandom.h"

#include "Particle.h"
#include "ParticleType.h"
#include "ResonanceType.h"

```

```cpp
int Generate2(int genLoops = 1e5) {
  int arrayDim = 130;
  int N = 100;

  Particle particle[arrayDim];
  Particle::AddParticleType("Pion+", 0.13957, +1);
  Particle::AddParticleType("Pion-", 0.13957, -1);
  Particle::AddParticleType("Kaon+", 0.49367, +1);
  Particle::AddParticleType("Kaon-", 0.49367, -1);
  Particle::AddParticleType("Proton+", 0.93827, +1);
  Particle::AddParticleType("Proton-", 0.93827, -1);
  Particle::AddParticleType("K*", 0.89166, 0, 0.050);

  TFile *file = new TFile("analysis.root", "RECREATE");

  TH1F *hPTypes = new TH1F("hPTypes", "Particle Types distribution", 7, 0., 7);
  TH1F *hTheta =
      new TH1F("hTheta", "Polar angle distribution", 400, 0, TMath::Pi());
  TH1F *hPhi =
      new TH1F("hPhi", "Azimuthal angle distribution", 400, 0, 2 * TMath::Pi());
  TH1F *hImpulse = new TH1F("hImpulse", "Impulse distribution", 400, 0, 8);
  TH1F *hTImpulse = new TH1F(
      "hTImpulse", "Transverse (X & Y) Impulse distribution", 400, 0, 8);
  TH1F *hEnergy = new TH1F("hEnergy", "Particle Energy", 400, 0, 4.5);


  TH1F *hInvMass =
      new TH1F("hInvMass", "Invariant mass: all particle", 400, 0, 5);
  TH1F *hSameCharge = new TH1F(
      "hSameCharge", "Invariant mass: same charge particle", 400, 0, 5);
  TH1F *hOddCharge = new TH1F(
      "hOddCharge", "Invariant mass: opposite charge particle", 400, 0, 5);
  TH1F *hPKconcordant =
      new TH1F("hPKconcordant", "Invariant mass: Pions and Kaons, same charge",
               400, 0, 5);
  TH1F *hPKdiscordant =
      new TH1F("hPKdiscordant",
               "Invariant mass: Pions and Kaons, opposite charge", 400, 0, 5);
  TH1F *hKDecay = new TH1F("hKDecay", "K* from decay", 400, 0, 5);

  gRandom->SetSeed();


  for (int i = 0; i < genLoops; i++) {
    double phi;   // azimuthal coordinate
    double theta; // polar coordinate
    double P;     // Impulse

    int extraPos = 0; // starting point for K* decay

    for (int j = 0; j < N; j++) {
      // initialization of angle coordinate and Impulse variables.
      phi = gRandom->Uniform(2 * TMath::Pi());
      theta = gRandom->Uniform(TMath::Pi());
      P = gRandom->Exp(1); // medium Impulse = 1Gev
      double Px = P * TMath::Sin(theta) * TMath::Cos(phi);
```

```
   double Py = P * TMath::Sin(theta) * TMath::Sin(phi);
   double Pz = P * TMath::Cos(theta);

   double transverseImpulse =
     std::sqrt(Px * Px + Py * Py); // Impulse on X and Y axis
   particle[j].SetP(Px, Py, Pz);

   // random number, uniformly distributed
   double index = gRandom->Uniform(1);
   // particle types percentages defined here
   if (index < 0.4) {
    particle[j].SetParticle("Pion+");
    // particle[j].GetCharge();

   } else if (index < 0.8) {
    particle[j].SetParticle("Pion-");

   } else if (index < 0.85) {
    particle[j].SetParticle("Kaon+");

   } else if (index < 0.9) {
    particle[j].SetParticle("Kaon-");

   } else if (index < 0.945) {
    particle[j].SetParticle("Proton+");

   } else if (index < 0.99) {
    particle[j].SetParticle("Proton-");
   } else if (index < 0.995)
      { //K* into Pion+ Kaon-
        particle[j].SetParticle("K*");
        particle[N + extraPos].SetParticle("Pion+");
        particle[N + extraPos + 1].SetParticle("Kaon-");
        particle[j].Decay2body(particle[N + extraPos], particle[N + extraPos +
1]);
        extraPos++;
        extraPos++;
      }
      else
      { //K* into Pion- Kaon+
        particle[j].SetParticle("K*");
        particle[N + extraPos].SetParticle("Pion-");
        particle[N + extraPos + 1].SetParticle("Kaon+");
        particle[j].Decay2body(particle[N + extraPos], particle[N + extraPos +
1]);
        extraPos++;
        extraPos++;
      }
   // particle types histogram filled according to percentages distribution
   hPTypes->Fill(particle[j].GetIndex());
   hTheta->Fill(theta);
   hPhi->Fill(phi);
   hImpulse->Fill(P);
   hTImpulse->Fill(transverseImpulse);
   hEnergy->Fill(particle[j].GetEnergy());
  }
```

```

    // filling invariant mass histogram
   int newArrayDim = N + extraPos;
    for (int h = 0; h < newArrayDim - 1; h++) {
      for (int k = h + 1; k < newArrayDim; k++) {
        int hCharge = particle[h].GetCharge();
        int kCharge = particle[k].GetCharge();
        int hIndex = particle[h].GetIndex();
        int kIndex = particle[k].GetIndex();

        double invMass = particle[h].InvMass(particle[k]);
        hInvMass->Fill(invMass); // filling invariant mass (with no charge
                                 // constraints) histogram

        if ((hCharge > 0 && kCharge > 0) || (hCharge < 0 && kCharge < 0)) {
          hSameCharge->Fill(invMass); // if confronted particle have the same
                                      // charge hSameCharge is filled
        }

        if (((hCharge > 0) && (kCharge < 0)) ||
            ((hCharge < 0) && (kCharge > 0))) {
          hOddCharge->Fill(invMass); // particle with opposite charge
        }

        if (((hIndex == 0) && (kIndex == 2)) ||
            ((hIndex == 1) && (kIndex == 3)) ||
            ((hIndex == 2) && (kIndex == 0)) ||
            ((hIndex == 3) && (kIndex == 1))) {
          hPKconcordant->Fill(invMass); // Pions and Kaons with equal charge
        }

        if (((hIndex == 0) && (kIndex == 3)) ||
            ((hIndex == 1) && (kIndex == 2)) ||
            ((hIndex == 3) && (kIndex == 0)) ||
            ((hIndex == 2) && (kIndex == 1))) {
          hPKdiscordant->Fill(invMass); // Pions and Kaons with opposite charge
        }
      }
    }

    // if any K* particle decayed => filling of relative invariant mass
    // histogram
    if (extraPos != 0) {
      for (int f = 0; f < extraPos; f += 2) {
        hKDecay->Fill(particle[N + f].InvMass(particle[N + f + 1]));
      }
    }
  }

  // definition of difference histograms
  // hDiff1 = hOddCharge - hSameCharge
  TH1F *hDiff1 = new TH1F(
      "hDiff1", "Opposite and Same charge particle difference", 400, 0, 5);
  hDiff1->Sumw2();
  hDiff1->Add(hOddCharge, hSameCharge, 1, -1);
  hDiff1->SetEntries(hDiff1->Integral());
```

```
  // hDiff2 = Pions-Kaons opposite charge - Pions-Kaons same charge
  TH1F *hDiff2 =
      new TH1F("hDiff2", "Opposite and Same charge Pions and Kaons difference",
               400, 0, 5);
  hDiff2->Sumw2();
  hDiff2->Add(hPKdiscordant, hPKconcordant, 1, -1);
  hDiff2->SetEntries(hDiff2->Integral());

  //Saving histograms to file

  TCanvas *c3 = new TCanvas("c3", "Types, angles, Impulse");
  c3->Divide(2, 2);
  c3->cd(1);
  hPTypes->Write();
  c3->cd(2);
  hTheta->Write();
  c3->cd(3);
  hPhi->Write();
  c3->cd(4);
  hImpulse->Write();

  TCanvas *c4 = new TCanvas("c4", "Invariant Mass Decay");
  c4->Divide(3, 1);
  c4->cd(1);
  hKDecay->Write();
  c4->cd(2);
  hDiff1->Write();
  c4->cd(3);
  hDiff2->Write();


  hTImpulse->Write();
  hEnergy->Write();
  hInvMass->Write();
  hSameCharge->Write();
  hOddCharge->Write();
  hPKconcordant->Write();
  hPKdiscordant->Write();

  file->Close();

  return 0;
}
```

Analysis.cxx

```
#include "TCanvas.h"
#include "TLegend.h"
#include "TF1.h"
#include "TFile.h"
#include "TH1F.h"
#include "TStyle.h"
#include "TROOT.h"
#include "TString.h"
```

```
#include <iostream>
#include <iomanip>
#include <array>
#include <cmath>

void Analize(int genLoops = 1e5) {


        std::cout<<"\n";
        TFile *file = new TFile("analysis.root", "UPDATE");

        // resuming histograms from ROOT file
        TH1F *hPTypes = (TH1F *)file->Get("hPTypes");
        TH1F *hTheta = (TH1F *)file->Get("hTheta");
        TH1F *hPhi = (TH1F *)file->Get("hPhi");
        TH1F *hImpulse = (TH1F *)file->Get("hImpulse");
        TH1F *hTImpulse = (TH1F *)file->Get("hTImpulse");
        TH1F *hEnergy = (TH1F *)file->Get("hEnergy");
        TH1F *hInvMass = (TH1F *)file->Get("hInvMass");
        TH1F *hSameCharge = (TH1F *)file->Get("hSameCharge");
        TH1F *hOddCharge = (TH1F *)file->Get("hOddCharge");
        TH1F *hPKconcordant = (TH1F *)file->Get("hPKconcordant");
        TH1F *hPKdiscordant = (TH1F *)file->Get("hPKdiscordant");
        TH1F *hKDecay = (TH1F *)file->Get("hKDecay");
        TH1F *hDiff1 = (TH1F *)file->Get("hDiff1");
        TH1F *hDiff2 = (TH1F *)file->Get("hDiff2");

        hKDecay->Sumw2();

        /////////////////////////////////////////////////////
        // Analizing particle types histograms
        std::cout << "Particle Types distribution stats:\n";
        for (int i = 1; i < 8; i++)
        {
        std::cout << "Bin " << i << " Entries fraction = " << hPTypes->GetBinContent(i) /
(genLoops * 100) << " ± "
                  << hPTypes->GetBinError(i) / (genLoops * 100) << "\n";
        }

        std::cout<<"\n";


        /////////////////////////////////////////////////////
        // analizing angles distributions
        std::cout << "Polar angle uniform distribution fit: \n";
        hTheta->Fit("pol0", "Q"); // uniform distribution fit
        hTheta->GetFunction("pol0")->SetLineColor(kRed);
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);
        hTheta->GetFunction("pol0")->Draw("SAME");
        std::cout << "Chi Square = " << hTheta->GetFunction("pol0")->GetChisquare() <<
"\n";
        std::cout << "NDF = " << hTheta->GetFunction("pol0")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = "<< hTheta->GetFunction("pol0")->GetChisquare()
/ hTheta->GetFunction("pol0")->GetNDF()<< "\n\n";

```

```

        ////////////////////////////////////////////////////////

        std::cout << "Azimuthal angle uniform distribution fit: \n";
        hPhi->Fit("pol0", "Q"); // uniform distribution fit
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);
        hPhi->GetFunction("pol0")->SetLineColor(kRed);
        std::cout << "Chi Square = " << hPhi->GetFunction("pol0")->GetChisquare()<< "\n";
        std::cout << "NDF = " << hPhi->GetFunction("pol0")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = "<< hPhi->GetFunction("pol0")->GetChisquare() /
hPhi->GetFunction("pol0")->GetNDF() << "\n\n";

        //////////////////////////////////////////////////////////
        // analizing Impulse distribution

        std::cout << "Impulse exponential fit: \n";
        hImpulse->Fit("expo", "Q"); // exponential dist. fit
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);
        hImpulse->GetFunction("expo")->SetLineColor(kRed);
        std::cout << "Chi Square = " << hImpulse->GetFunction("expo")->GetChisquare() <<
"\n";
        std::cout << "NDF = " << hImpulse->GetFunction("expo")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = "<< hImpulse->GetFunction("expo")-
>GetChisquare() / hImpulse->GetFunction("expo")->GetNDF() << "\n";
        std::cout << "Mean = " << hImpulse->GetMean() << " ± " << hImpulse->GetMeanError()
<< "(GeV)\n";


        std::cout << "Decay of K* gaussian fit: \n";
        hKDecay->Fit("gaus", "", "", 0, 5);
        hKDecay->GetFunction("gaus")->SetLineColor(kRed);
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);
        std::cout << "Chi Square = " << hKDecay->GetFunction("gaus")->GetChisquare()<<
"\n";
        std::cout << "NDF = " << hKDecay->GetFunction("gaus")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = " << hKDecay->GetFunction("gaus")-
>GetChisquare() / hKDecay->GetFunction("gaus")->GetNDF()<< "\n\n";


        std::cout << "Opposite and Same charge particles difference gaussian fit: \n";
        hDiff1->Fit("gaus", "", "", 0, 5);
        hDiff1->Fit("fit1");
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);

        std::cout << "Chi Square = " << hDiff1->GetFunction("gaus")->GetChisquare()
               << "\n";
        std::cout << "NDF = " << hDiff1->GetFunction("gaus")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = "
               << hDiff1->GetFunction("gaus")->GetChisquare() / hDiff1-
>GetFunction("gaus")->GetNDF()<< "\n\n";


```

```
        std::cout << "Opposite and Same charge Pions and Kaons difference gaussian fit:
\n";
        hDiff2->Fit("gaus", "", "", 0, 5);
        hDiff2->GetFunction("gaus")->SetLineColor(kRed);
        gStyle->SetOptStat(1111);
        gStyle->SetOptFit(111);
        std::cout << "Chi Square = " << hDiff2->GetFunction("gaus")->GetChisquare()
               << "\n";
        std::cout << "NDF = " << hDiff2->GetFunction("gaus")->GetNDF() << "\n";
        std::cout << "Reduced Chi Square = "
               << hDiff2->GetFunction("gaus")->GetChisquare() / hDiff2-
>GetFunction("gaus")->GetNDF()<< "\n\n";


        // histograms cosmetics

        TCanvas *c1 = new TCanvas("c1", "Detector statistics");
        c1->Divide(2, 3);

        c1->cd(1);
        hPTypes->SetMinimum(0);
        hPTypes->SetFillColor(kBlue);
        hPTypes->GetXaxis()->SetTitle("Particle types");
        hPTypes->GetYaxis()->SetTitle("Number of occurrences");
        hPTypes->GetXaxis()->SetBinLabel(1, "Pions +");
        hPTypes->GetXaxis()->SetBinLabel(2, "Pions -");
        hPTypes->GetXaxis()->SetBinLabel(3, "Kaons +");
        hPTypes->GetXaxis()->SetBinLabel(4, "Kaons +");
        hPTypes->GetXaxis()->SetBinLabel(5, "Protons+");
        hPTypes->GetXaxis()->SetBinLabel(6, "Protons-");
        hPTypes->GetXaxis()->SetBinLabel(7, "K*");
        hPTypes->Draw("H");
        hPTypes->Draw("E,SAME");

        c1->cd(2);
        hTheta->SetFillColor(kBlue);
        hTheta->GetXaxis()->SetTitle("Theta (radiants)");
        hTheta->GetYaxis()->SetTitle("Number of occurrences");
        TLegend* Leg1 = new TLegend(.1,.7,.3,.9,"");
        Leg1->SetFillColor(0);
        Leg1->AddEntry(hTheta,"Data");
        Leg1->AddEntry(hTheta->GetFunction("pol0"),"Fit");
        hTheta->Draw("H");
        hTheta->Draw("E,SAME");
        Leg1->Draw("SAME");

        c1->cd(3);
        hPhi->SetFillColor(kBlue);
        hPhi->GetXaxis()->SetTitle("Phi (radiants)");
        hPhi->GetYaxis()->SetTitle("Number of occurrences");
        TLegend* Leg2 = new TLegend(.1,.7,.3,.9,"");
        Leg2->SetFillColor(0);
        Leg2->AddEntry(hPhi,"Data");
        Leg2->AddEntry(hPhi->GetFunction("pol0"),"Fit");
        hPhi->Draw("H");
        hPhi->Draw("E,SAME");
```

```
        Leg2->Draw("SAME");

        c1->cd(4);
        hImpulse->SetFillColor(kBlue);
        hImpulse->GetXaxis()->SetTitle("Impulse (GeV)");
        hImpulse->GetYaxis()->SetTitle("Number of occurrences");
        TLegend* Leg3 = new TLegend(.1,.7,.3,.9,"");
        Leg3->SetFillColor(0);
        Leg3->AddEntry(hImpulse,"Data");
        Leg3->AddEntry(hImpulse->GetFunction("expo"),"Fit");
        hImpulse->Draw("H");
        hImpulse->Draw("E,SAME");
        Leg3->Draw("SAME");

        c1->cd(5);
        hTImpulse->SetFillColor(kBlue);
        hTImpulse->GetXaxis()->SetTitle("Transverse Impulse (GeV)");
        hTImpulse->GetYaxis()->SetTitle("Number of occurrences");
        hTImpulse->Draw("H");
        hTImpulse->Draw("E,SAME");

        c1->cd(6);
        hEnergy->SetFillColor(kBlue);
        hEnergy->GetXaxis()->SetTitle("Energy (GeV)");
        hEnergy->GetYaxis()->SetTitle("Number of occurrences");
        hEnergy->Draw("H");
        hEnergy->Draw("E,SAME");

        TCanvas *massCanvas = new TCanvas("massCanvas", "Invariant mass histograms");
        massCanvas->Divide(2, 3);


        massCanvas->cd(1);
        hInvMass->SetFillColor(kBlue);
        hInvMass->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hInvMass->GetYaxis()->SetTitle("Number of occurrences");
        hInvMass->Draw("H");

        massCanvas->cd(2);
        hSameCharge->SetFillColor(kBlue);
        hSameCharge->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hSameCharge->GetYaxis()->SetTitle("Number of Occurrences");
        hSameCharge->Draw("H");
        hSameCharge->Draw("E,SAME");

        massCanvas->cd(3);
        hOddCharge->SetFillColor(kBlue);
        hOddCharge->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hOddCharge->GetYaxis()->SetTitle("Number of Occurrences");
        hOddCharge->Draw("H");
        hOddCharge->Draw("E,SAME");

        massCanvas->cd(4);
        hPKconcordant->SetFillColor(kBlue);
        hPKconcordant->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hPKconcordant->GetYaxis()->SetTitle("Number of Occurrences");
```

```
        hPKconcordant->Draw("H");
        hPKconcordant->Draw("E,SAME");

        massCanvas->cd(5);
        hPKdiscordant->SetFillColor(kBlue);
        hPKdiscordant->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hPKdiscordant->GetYaxis()->SetTitle("Number of Occurrences");
        hPKdiscordant->Draw("H");
        hPKdiscordant->Draw("E,SAME");

        TCanvas *c2 = new TCanvas("c2", "K* decay statistics");
        c2->Divide(2, 2);

        c2->cd(1);
        hKDecay->SetFillColor(kBlue);
        hKDecay->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hKDecay->GetYaxis()->SetTitle("Number of Occurrences");
        hKDecay->Draw("H");
        hKDecay->Draw("E,SAME");
        TLegend *Leg4=new TLegend(.1,.7,.3,.9,"");
        Leg4->SetFillColor(0);
        Leg4->AddEntry(hKDecay,"Data");
        Leg4->AddEntry(hKDecay->GetFunction("gaus"),"Fit");
        Leg4->Draw("SAME");


        c2->cd(2);

        hDiff1->SetFillColor(kBlue);
        hDiff1->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hDiff1->GetYaxis()->SetTitle("Number of Occurrences");
        hDiff1->Draw("H");
        hDiff1->Draw("E,SAME");
        TLegend *Leg5=new TLegend(.1,.7,.3,.9,"");
        Leg5->SetFillColor(0);
        Leg5->AddEntry(hDiff1,"Data");
        Leg5->AddEntry(hDiff1->GetFunction("gaus"),"Fit");
        Leg5->Draw("SAME");


        c2->cd(3);
        hDiff2->SetFillColor(kBlue);
        hDiff2->GetXaxis()->SetTitle("Invariant mass (GeV/c^{2})");
        hDiff2->GetYaxis()->SetTitle("Number of Occurrences");
        hDiff2->Draw("H");
        hDiff2->Draw("E,SAME");
        TLegend *Leg6=new TLegend(.1,.7,.3,.9,"");
        Leg6->SetFillColor(0);
        Leg6->AddEntry(hDiff2,"Data");
        Leg6->AddEntry(hDiff2->GetFunction("gaus"),"Fit");
        Leg6->Draw("SAME");


        c1->SaveAs("types-Impulse-angles.pdf");
        c2->SaveAs("Kstar-stats.pdf");
        massCanvas->SaveAs("invariant-masses.pdf");
```

```
        c1->SaveAs("types-Impulse-angles.root");
        c2->SaveAs("Kstar-stats.root");
        massCanvas->SaveAs("invariant-masses.root");

}
```