Relazione di Laboratorio - ROOT

Marta Barbieri, Riccardo Franchi, Giorgio Germano

Dicembre 2023

Introduzione

È stato implementato un codice il cui scopo è, in primo luogo, quello di simulare l'acquisizione di dati di un rivelatore in seguito ad una collisione di particelle elementari e, in secondo luogo, quello di effettuare un'analisi statistica di tali dati tramite istogrammi e individuare le particelle decadute. Per fare ciò si è scritto un programma in C++, che utilizza il metodo Monte Carlo, utilizzando del framework di analisi dati `ROOT`. Sono state considerate varie simulazioni da un numero fisso di particelle di 7 tipi diversi ($\pi^+, \pi^-, K^+, K^-, p^+, p^-$ e K^*); fra queste, le particelle di tipo K^* sono instabili e sono dunque quelle soggette a decadimento. Ciò fa sì che esse non possano essere rilevate direttamente, ma debbano appunto essere individuate tramite un'analisi della massa invariante del campione, noti i prodotti di decadimento delle stesse.

1 Struttura del codice

Il codice è composto da due sezioni. La prima, suddivisa in diversi file, si occupa di generare le particelle, utilizzando il metodo Monte Carlo, e di riempire degli istogrammi con i dati relativi alle particelle generate. Nella seconda parte, invece, vengono letti gli istogrammi dal file `root` generato nella prima sezione, eseguiti i fit, e salvati i risultati su file.
Al contrario della seconda parte, che è costituita da un unico file macro di ROOT, la prima parte è composta dalle classi `ParticleType`, `ResonanceType` e `Particle`, ciascuna dichiarata e implementata nei rispettivi file header e source, oltre al file principale `main.cpp`, ed è strutturata in modo da poter essere eseguita sia come macro ROOT sia da shell, utilizzando ROOT come libreria.
La classe `ParticleType` ha come membri il nome, la massa e la carica caratteristici di ogni tipo di particella. Vi è poi la classe `ResonanceType` che, essendo derivata da `ParticleType`, eredita da essa tali membri, e ha in aggiunta quello relativo alla larghezza della particella. Questo approccio è particolarmente conveniente perché permette di sfruttare il polimorfismo dinamico, salvando le risonanze nello stesso container delle particelle.
La classe `Particle` invece, contiene un array statico che contiene le informazioni relative ai tipi di particelle che vengono generate, e alcuni metodi statici annessi. In questo modo, si ha come membro non statico della classe, oltre alle tre componenti cartesiane della quantità di moto, soltanto l'indice dell'array statico che corrisponde al tipo di particella considerato. Sono inoltre presenti vari metodi, tra cui quelli per calcolare l'energia della particella e la massa invariante, e quello per gestire il decadimento di una risonanza.
Infine, nel file `main.cpp`, avviene la generazione delle particelle, il decadimento delle risonanze e il salvataggio dei dati generati negli istogrammi, salvati in un file `root`.

2 Generazione

È stato generato un totale di 100 000 eventi, ognuno dei quali con inizialmente 100 particelle di tipi diversi (si veda la Tabella 1 per proporzioni e caratteristiche dei diversi tipi). Utilizzando `TRandom::Uniform()`, per ogni particella viene generato un numero pseudo-casuale nell'intervallo $[0, 1]$, tramite una PDF uniforme, e sulla base di esso viene determinato il tipo della particella, tramite una generazione con proporzioni definite.
Ogni particella, poi, è caratterizzata da un angolo azimutale ϕ, generato uniformemente in $[0, 2\pi]$, da uno polare θ generato uniformemente in $[0, \pi]$ e da un impulso p generato a partire da una PDF esponenziale con media $\bar{p} = 1$ GeV. Per determinarli si è proceduto come nel caso precedente, utilizzando

Particella	Probabilità (%)	Massa (GeV/c^2)	Carica (e)
π^+	40	0.13957	$+1$
π^-	40	0.13957	-1
K^+	5	0.49367	$+1$
K^-	5	0.49367	-1
p^+	4.5	0.93827	$+1$
p^-	4.5	0.93827	-1
K^* (Risonanza)	1	0.89166	0

Tabella 1: Probabilità, massa e carica delle particelle a seconda del tipo

`TRandom::Uniform()` e `TRandom::Exp(double mean)`. Le coordinate sferiche dell'impulso (p, ϕ, θ) sono poi state convertite in coordinate cartesiane (p_x, p_y, p_z).
La K^* è una risonanza caratterizzata da una larghezza $\Gamma = 0.050$ GeV/c^2; è una particella instabile, che decade in una coppia π^+/K^- o in una π^-/K^+, con medesima probabilità.

3 Analisi

In primo luogo, sono state confrontate le particelle generate, fornite dalla stampa a schermo del programma, con quelle attese in base alle probabilità della Tabella 1. I dati, come si può vedere nella Tabella 2, sono in ottimo accordo.

Particella	Occorrenze Osservate	Occorrenze Attese
π^+	$(4001.9 \pm 2.0) \cdot 10^3$	$4000 \cdot 10^3$
π^-	$(3998.3 \pm 2.0) \cdot 10^3$	$4000 \cdot 10^3$
K^+	$(4995.9 \pm 7.1) \cdot 10^2$	$5000 \cdot 10^2$
K^-	$(4993.5 \pm 7.1) \cdot 10^2$	$5000 \cdot 10^2$
p^+	$(4502.8 \pm 6.7) \cdot 10^2$	$4500 \cdot 10^2$
p^-	$(4501.9 \pm 6.7) \cdot 10^2$	$4500 \cdot 10^2$
K^*	$(1004.3 \pm 3.2) \cdot 10^2$	$1000 \cdot 10^2$

Tabella 2: Occorrenze osservate e attese delle particelle

Si è, poi, controllata la correttezza delle distribuzioni di θ, ϕ e p. Come evidente dalla Tabella 3, le particelle generate seguono effettivamente le distribuzioni desiderate.

Distribuzione	Parametri del Fit	χ^2	DOF	χ^2/DOF
Fit a distribuzione angolo θ (`pol0`)	19999.1 $\pm$ 6.3	453.953	499	0.909725
Fit a distribuzione angolo ϕ (`pol0`)	19999.1 $\pm$ 6.3	454.444	499	0.91071
Fit a distribuzione modulo impulso (`expo`)	(-1.00013 $\pm$ 0.00035) GeV/c	458.64	498	0.920963

Tabella 3: Informazioni sui fit

Sono stati quindi analizzati i grafici relativi alla massa invariante delle particelle. Sapendo che la K^* decade in due particelle di segno opposto, e che la massa invariante si conserva nel processo di decadimento, è possibile risalire alla sua massa e larghezza. Infatti, sottraendo il grafico della massa invariante di tutte le coppie di particelle con segno concorde (in cui si hanno solo combinazioni causali) da quello che considera particelle di segno discorde (combinazioni casuali e prodotti della K^*), è possibile ottenere il segnale cercato. È stato effettivamente ottenuto un segnale non attribuibile al rumore di fondo (Figura 2, secondo grafico).
Ripetendo il medesimo procedimento con le coppie $\pi - K$ di segno concorde e discorde si ottiene un

risultato ancora più vicino a quello atteso (Figura 2, terzo grafico). La distribuzione ottenuta è approssimabile ad una gaussiana con media pari alla massa della K^* e deviazione standard uguale alla sua larghezza (si veda la Tabella 4).

Distribuzione e fit	**Media** (GeV/c^2)	$\boldsymbol{\sigma}$ (GeV/c^2)	**Ampiezza** (GeV/c^2)	$\boldsymbol{\chi^2}$**/DOF**
Massa Invariante vere K^*	0.89182 ± 0.00016	0.04987 ± 0.00011	200.50 ± 0.63	0.990962
Massa Invariante ottenuta da differenza delle combinazioni di carica discorde e concorde (fit gauss)	0.8939 ± 0.0043	0.0431 ± 0.0037	914 ± 76	1.01202
Massa Invariante ottenuta da differenza delle combinazioni $\pi - K$ di carica discorde e concorde (fit gauss)	0.8933 ± 0.0025	0.0460 ± 0.0023	959 ± 44	1.04174

Tabella 4: Informazioni sulla K^*

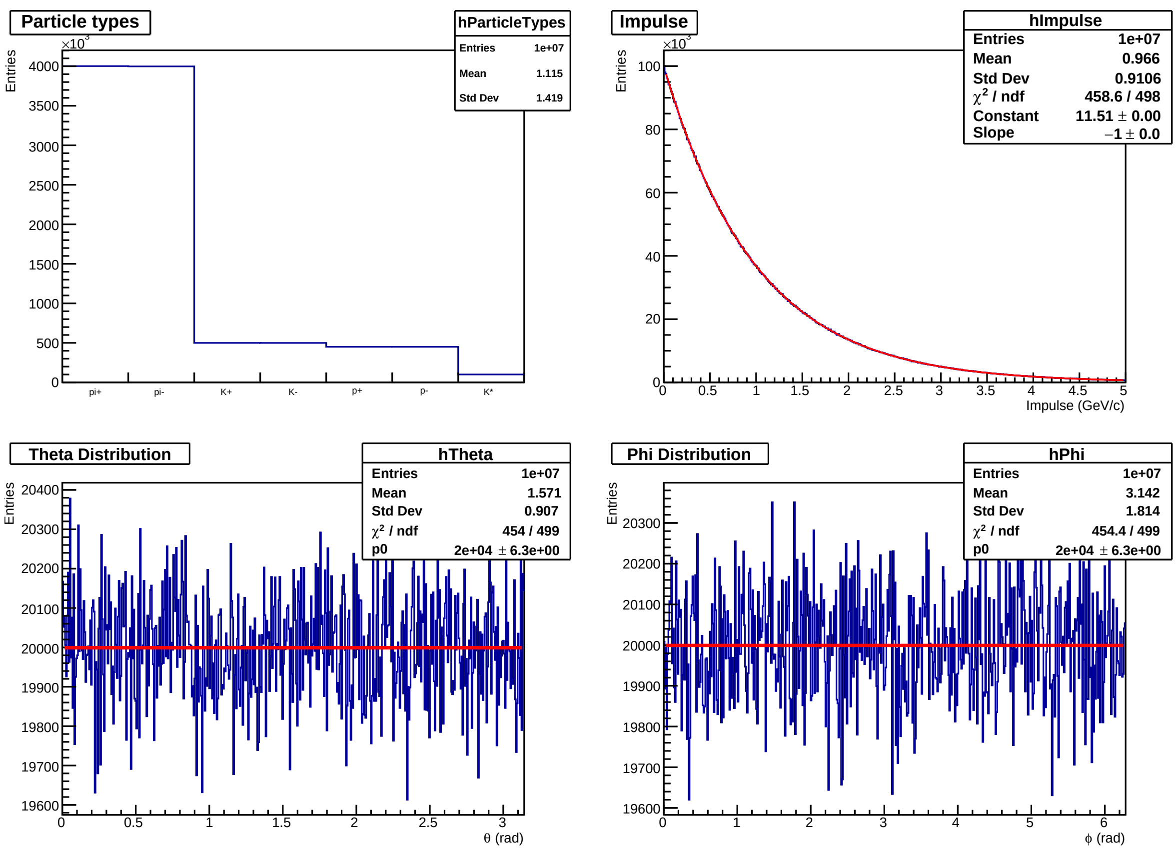

Figura 1: Istogrammi relativi alla distribuzione dei tipi di particelle, del modulo del loro impulso e degli angoli azimutale e polare, con relativi fit

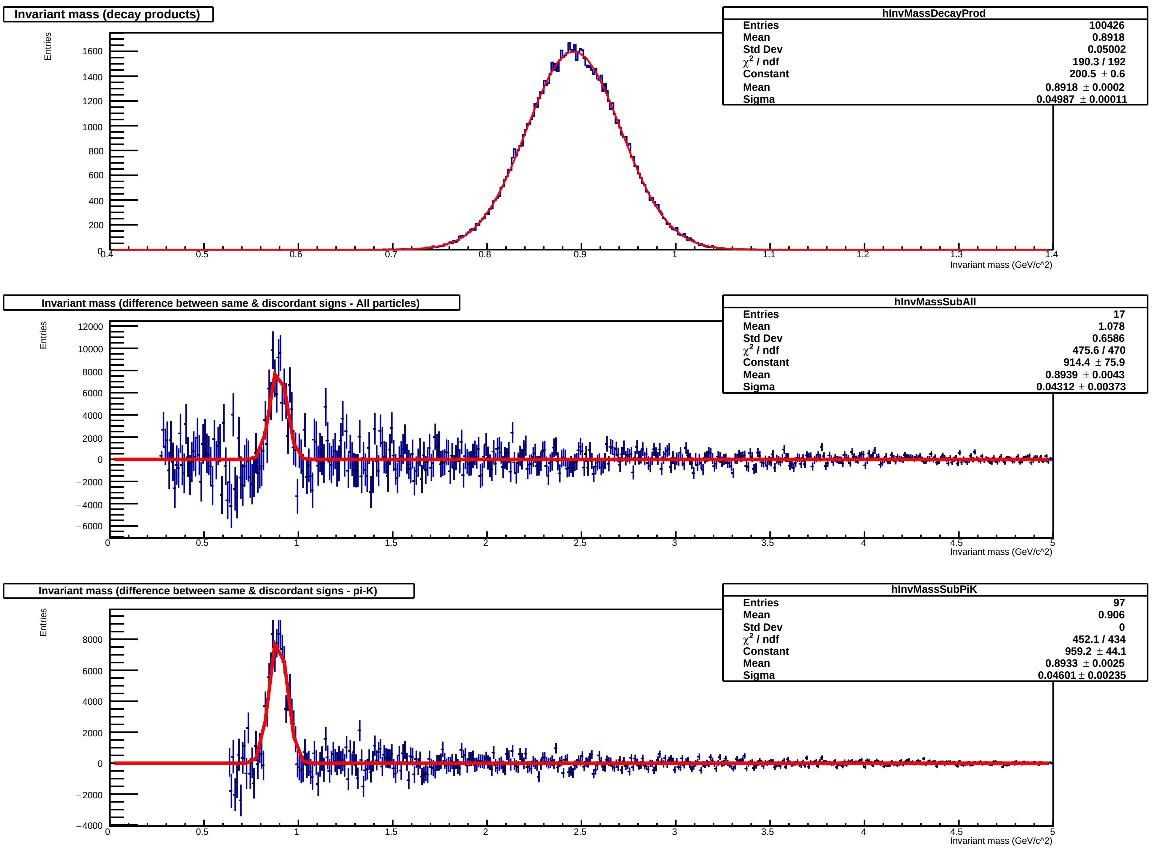

Figura 2: Dall'alto verso il basso: istogramma della massa invariante dei prodotti della K^*; istogramma della differenza tra particelle di carica discorde e concorde; istogramma della differenza tra coppie $\pi - K$ discordi e concordi

A Listing del codice

Il codice può anche essere trovato sulla *repository* GitHub `https://github.com/gioger/ROOTLab`

A.1 `particle_type.hpp`

```cpp
#ifndef PARTICLE_TYPE_HPP
#define PARTICLE_TYPE_HPP

#include <string>

class ParticleType
{
public:
  ParticleType(std::string name, double mass, int charge);
  virtual ~ParticleType() = default;

  double GetMass() const { return fMass; }
  int GetCharge() const { return fCharge; }
  const std::string& GetName() const { return fName; }
  virtual void Print() const;
  virtual double GetWidth() const { return 0; };

private:
  const std::string fName{};
  const double fMass{};
  const int fCharge{};
};

#endif // PARTICLE_TYPE_HPP
```

Listing 1: File `particle_type.hpp`

A.2 `particle_type.cpp`

```cpp
#include "particle_type.hpp"

#include <iostream>

ParticleType::ParticleType(std::string name, double mass, int charge)
  : fName{std::move(name)}, fMass{mass}, fCharge{charge}
{
  if (fMass < 0.)
  {
    std::cerr << "ParticleType::ParticleType: "
          << "invalid mass value: " << fMass << '\n';
    std::exit(EXIT_FAILURE);
  }
}

void ParticleType::Print() const
{
  std::cout << "Particle type: " << fName << '\n'
        << "Mass: " << fMass << " GeV/c^2\n"
        << "Charge: " << fCharge << " e\n";
}
```

Listing 2: File `particle_type.cpp`

A.3 `resonance_type.hpp`

```cpp
#ifndef RESONANCE_TYPE_HPP
#define RESONANCE_TYPE_HPP

#include "particle_type.hpp"

class ResonanceType : public ParticleType
{
```

```
public:
  ResonanceType(std::string name, double mass, int charge, double width = 0);
  ~ResonanceType() override = default;

  double GetWidth() const override { return fWidth; }
  void Print() const override;

private:
  const double fWidth{};
};

#endif // RESONANCE_TYPE_HPP
```

Listing 3: File resonance_type.hpp

A.4 resonance_type.cpp

```
#include "resonance_type.hpp"

#include <iostream>

ResonanceType::ResonanceType(std::string name, double mass, int charge, double width)
  : ParticleType{std::move(name), mass, charge}, fWidth{width}
{
  if (fWidth < 0.)
  {
    std::cerr << "ResonanceType::ResonanceType: "
           << "invalid width value: " << fWidth << '\n';
    std::exit(EXIT_FAILURE);
  }
}

void ResonanceType::Print() const
{
  ParticleType::Print();
  std::cout << "Width: " << fWidth << " GeV/c^2\n";
}
```

Listing 4: File resonance_type.cpp

A.5 particle.hpp

```
#ifndef PARTICLE_HPP
#define PARTICLE_HPP

#include "particle_type.hpp"

#include <array>
#include <iostream>
#include <memory>

class Particle
{
public:
  static void AddParticleType(std::string name, double mass, int charge, double width =
    0.);
  static void PrintParticleTypes();

  Particle(const std::string& name, double px = 0., double py = 0., double pz = 0.);
  Particle() = default;
  void PrintParticleData() const;

  size_t GetIndex() const { return fIndex; }
  void SetIndex(size_t index);
  void SetIndex(const std::string& name);

  double GetPx() const { return fPx; }
  double GetPy() const { return fPy; }
  double GetPz() const { return fPz; }
  double GetMass() const { return fParticleTypes[fIndex]->GetMass(); }
```

```cpp

  int GetCharge() const { return fParticleTypes[fIndex]->GetCharge(); }
  const std::string& GetName() const { return fParticleTypes[fIndex]->GetName(); }

  void SetP(double px, double py, double pz)
  {
    fPx = px;
    fPy = py;
    fPz = pz;
  }

  double Energy() const;
  double InvMass(const Particle& particle) const;
  void Decay2Body(Particle& dau1, Particle& dau2) const;

private:
  static constexpr size_t fMaxNumParticleType{10};
  static inline std::array<std::unique_ptr<ParticleType>, fMaxNumParticleType>
    fParticleTypes{};

  static inline size_t fNParticleType{};

  size_t FindParticle(const std::string& particleName);

  void Boost(double bx, double by, double bz);

  size_t fIndex{};

  double fPx{};
  double fPy{};
  double fPz{};
};

#endif // PARTICLE_HPP
```

Listing 5: File particle.hpp

A.6 particle.cpp

```cpp
#include "particle.hpp"
#include "resonance_type.hpp"

#include <cmath>
#include <random>

void Particle::AddParticleType(std::string name, double mass, int charge, double width)
{
  if (fNParticleType >= fMaxNumParticleType)
  {
    std::cerr << "Maximum number of particle types reached.\n";
    std::exit(EXIT_FAILURE);
  }

  fParticleTypes[fNParticleType] = (width != 0)
                         ? std::make_unique<ResonanceType>(std::move(name), mass, charge,
    width)
                         : std::make_unique<ParticleType>(std::move(name), mass, charge);

  ++fNParticleType;
}

void Particle::PrintParticleTypes()
{
  for (size_t i{}; i < fNParticleType; ++i)
  {
    fParticleTypes[i]->Print();
  }
}

Particle::Particle(const std::string& name, double px, double py, double pz) : fPx{px},
    fPy{py}, fPz{pz}
{
```

```
  fIndex = FindParticle(name);
}

size_t Particle::FindParticle(const std::string& particleName)
{
  for (size_t i{0}; i < fNParticleType; ++i)
  {
    if (fParticleTypes[i]->GetName() == particleName)
    {
      return i;
    }
  }

  std::cerr << "Particle type not found.\n";
  std::exit(EXIT_FAILURE);
}

void Particle::SetIndex(size_t index)
{
  if (index >= fNParticleType)
  {
    std::cerr << "Invalid index.\n";
    std::exit(EXIT_FAILURE);
  }

  fIndex = index;
}

void Particle::SetIndex(const std::string& name)
{
  fIndex = FindParticle(name);
}

void Particle::PrintParticleData() const
{
  const std::string& name{fParticleTypes[fIndex]->GetName()};
  std::cout << "Particle index: " << fIndex << '\n';
  std::cout << "Particle name: " << name << '\n';
  std::cout << "Particle Px: " << fPx << '\n';
  std::cout << "Particle Py: " << fPy << '\n';
  std::cout << "Particle Pz: " << fPz << '\n';
}

double Particle::Energy() const
{
  const double mass{GetMass()};
  return std::sqrt(mass * mass + fPx * fPx + fPy * fPy + fPz * fPz);
}

double Particle::InvMass(const Particle& particle) const
{
  const double sumEnergy{Energy() + particle.Energy()};
  const double sumPx{fPx + particle.fPx};
  const double sumPy{fPy + particle.fPy};
  const double sumPz{fPz + particle.fPz};

  return std::sqrt(sumEnergy * sumEnergy - sumPx * sumPx - sumPy * sumPy - sumPz * sumPz
    );
}
void Particle::Decay2Body(Particle& dau1, Particle& dau2) const
{
  if (GetMass() == 0.0)
  {
    std::cerr << "Decayment cannot be preformed if mass is zero\n";
    std::exit(EXIT_FAILURE);
  }

  double massMot{GetMass()};
  const double massDau1{dau1.GetMass()};
  const double massDau2{dau2.GetMass()};

    // Initialize random engine
  std::default_random_engine engine{std::random_device{}()};
```

```
    // Initialize a normal distribution with mean 0 and std.dev 1
  std::normal_distribution<double> normDistr{0., 1.};

  const double y1{normDistr(engine)};

  massMot += fParticleTypes[fIndex]->GetWidth() * y1;

  if (massMot < massDau1 + massDau2)
  {
    std::cerr << "Decayment cannot be preformed because mass is too low in this channel\n";
    std::exit(EXIT_FAILURE + 1);
  }

  const double pOut{std::sqrt((massMot * massMot - (massDau1 + massDau2) * (massDau1 + massDau2)) *
                        (massMot * massMot - (massDau1 - massDau2) * (massDau1 - massDau2))) /
                    massMot * 0.5};

  std::uniform_real_distribution<double> phiDistr{0., M_PI * 2.};
  std::uniform_real_distribution<double> thetaDistr{-M_PI_2, M_PI_2};
    // Save some reused values as variables,
    // since calculating sine & cosine is computationally expensive
  const double phi{phiDistr(engine)};
  const double theta{thetaDistr(engine)};
  const double sinTheta{std::sin(theta)};
  const double cosTheta{std::cos(theta)};
  const double sinPhi{std::sin(phi)};
  const double cosPhi{std::cos(phi)};

  dau1.SetP(pOut * sinTheta * cosPhi, pOut * sinTheta * sinPhi, pOut * cosTheta);
  dau2.SetP(-pOut * sinTheta * cosPhi, -pOut * sinTheta * sinPhi, -pOut * cosTheta);

  const double energy{std::sqrt(fPx * fPx + fPy * fPy + fPz * fPz + massMot * massMot)};

  const double bx{fPx / energy};
  const double by{fPy / energy};
  const double bz{fPz / energy};

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

void Particle::Boost(double bx, double by, double bz)
{
  const double energy{Energy()};
  // Boost this Lorentz vector
  const double b2{bx * bx + by * by + bz * bz};
  const double gamma{1.0 / sqrt(1.0 - b2)};
  const double bp{bx * fPx + by * fPy + bz * fPz};
  const 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;
}
```

Listing 6: File particle.cpp

A.7 main.cpp

```
#include <TFile.h>
#include <TH1.h>
#include <TMath.h>
#include <TRandom.h>

#include <algorithm>
#include <vector>

#include "particle.hpp"

void Setup()
{
```

```cpp
  gRandom->SetSeed();
  // Initialize all particle types
  Particle::AddParticleType("pi+", 0.13957, 1);
  Particle::AddParticleType("pi-", 0.13957, -1);
  Particle::AddParticleType("K+", 0.49367, 1);
  Particle::AddParticleType("K-", 0.49367, -1);
  Particle::AddParticleType("p+", 0.93827, 1);
  Particle::AddParticleType("p-", 0.93827, -1);
  Particle::AddParticleType("K*", 0.89166, 0, 0.05);
}

int main()
{
  Setup();

  // Initialize histograms with the proper axes labels
  auto* hParticleTypes{new TH1I{"hParticleTypes", "Particle types", 7, 0, 7}};
  hParticleTypes->GetXaxis()->SetBinLabel(1, "pi+");
  hParticleTypes->GetXaxis()->SetBinLabel(2, "pi-");
  hParticleTypes->GetXaxis()->SetBinLabel(3, "K+");
  hParticleTypes->GetXaxis()->SetBinLabel(4, "K-");
  hParticleTypes->GetXaxis()->SetBinLabel(5, "p+");
  hParticleTypes->GetXaxis()->SetBinLabel(6, "p-");
  hParticleTypes->GetXaxis()->SetBinLabel(7, "K*");
  hParticleTypes->GetYaxis()->SetTitle("Entries");

  auto* hPhi{new TH1D{"hPhi", "Phi Distribution", 500, 0., TMath::TwoPi()}};
  hPhi->GetXaxis()->SetTitle("#phi (rad)");
  hPhi->GetYaxis()->SetTitle("Entries");

  auto* hTheta{new TH1D{"hTheta", "Theta Distribution", 500, 0., TMath::Pi()}};
  hTheta->GetXaxis()->SetTitle("#theta (rad)");
  hTheta->GetYaxis()->SetTitle("Entries");

  auto* hImpulse{new TH1D{"hImpulse", "Impulse", 500, 0., 5.}};
  hImpulse->GetXaxis()->SetTitle("Impulse (GeV/c)");
  hImpulse->GetYaxis()->SetTitle("Entries");

  auto* hTransverseImpulse{new TH1D{"hTransverseImpulse", "Transverse impulse", 100, 0., 4.}};
  hTransverseImpulse->GetXaxis()->SetTitle("Transverse impulse (GeV/c)");
  hTransverseImpulse->GetYaxis()->SetTitle("Entries");

  auto* hEnergy{new TH1D{"hEnergy", "Energy", 500, 0., 5.}};
  hEnergy->GetXaxis()->SetTitle("Energy (GeV)");
  hEnergy->GetYaxis()->SetTitle("Entries");

  auto* hInvMass{new TH1D{"hInvMass", "Invariant mass (all particles)", 500, 0., 6.}};
  hInvMass->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
  hInvMass->GetYaxis()->SetTitle("Entries");

  auto* hInvMassSameSign{new TH1D{"hInvMassSameSign", "Invariant mass (same sign particles)", 500, 0., 5.}};
  hInvMassSameSign->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
  hInvMassSameSign->GetYaxis()->SetTitle("Entries");

  auto* hInvMassDiscSign{new TH1D{"hInvMassDiscSign", "Invariant mass (discordant sign particles)", 500, 0., 5.}};
  hInvMassDiscSign->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
  hInvMassDiscSign->GetYaxis()->SetTitle("Entries");

  auto* hInvMassPiKSame{new TH1D{"hInvMassPiKSame", "Invariant mass (pi-K same sign)", 500, 0., 5.}};
  hInvMassPiKSame->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
  hInvMassPiKSame->GetYaxis()->SetTitle("Entries");

  auto* hInvMassPiKDisc{new TH1D{"hInvMassPiKDisc", "Invariant mass (pi-K discordant sign)", 500, 0., 5.}};
  hInvMassPiKDisc->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
  hInvMassPiKDisc->GetYaxis()->SetTitle("Entries");

  auto* hInvMassDecayProd{new TH1D{"hInvMassDecayProd", "Invariant mass (decay products)", 500, 0.4, 1.4}};
```

```
hInvMassDecayProd->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
hInvMassDecayProd->GetYaxis()->SetTitle("Entries");

hInvMassSameSign->Sumw2();
hInvMassDiscSign->Sumw2();
hInvMassPiKSame->Sumw2();
hInvMassPiKDisc->Sumw2();

auto* hInvMassSubAll{new TH1D{
  "hInvMassSubAll", "Invariant mass (difference between same & discordant signs - All
  particles)", 500, 0., 5.}};
hInvMassSubAll->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
hInvMassSubAll->GetYaxis()->SetTitle("Entries");

auto* hInvMassSubPiK{
  new TH1D{"hInvMassSubPiK", "Invariant mass (difference between same & discordant
  signs - pi-K)", 500, 0., 5.}};
hInvMassSubPiK->GetXaxis()->SetTitle("Invariant mass (GeV/c^2)");
hInvMassSubPiK->GetYaxis()->SetTitle("Entries");

constexpr size_t numEvents{100000};
constexpr size_t numParts{100};

std::vector<Particle> eventParticles;
eventParticles.reserve(numParts);

for (size_t i{0}; i < numEvents; i++)
{
  // Generate particles for each event
  eventParticles.clear();
  std::generate_n( //
    std::back_inserter(eventParticles), numParts,
    [&]()
    {
      const double phi{gRandom->Uniform(0., TMath::TwoPi())};
      const double theta{gRandom->Uniform(0., TMath::Pi())};
      const double p{gRandom->Exp(1.)};

      hPhi->Fill(phi);
      hTheta->Fill(theta);
      hImpulse->Fill(p);

      const double sinTheta{TMath::Sin(theta)};

      const double px{p * sinTheta * TMath::Cos(phi)};
      const double py{p * sinTheta * TMath::Sin(phi)};
      const double pz{p * TMath::Cos(theta)};

      hTransverseImpulse->Fill(std::sqrt(px * px + py * py));

      const double x{gRandom->Uniform()};

      std::string particleName;
      if (x < 0.4)
      {
        particleName = "pi+";
      }
      else if (x < 0.8)
      {
        particleName = "pi-";
      }
      else if (x < 0.85)
      {
        particleName = "K+";
      }
      else if (x < 0.9)
      {
        particleName = "K-";
      }
      else if (x < 0.945)
      {
        particleName = "p+";
      }
```

```
        else if (x < 0.99)
        {
          particleName = "p-";
        }
        else
        {
          particleName = "K*";
        }

        Particle particle{particleName, px, py, pz};
        hParticleTypes->Fill(particle.GetIndex());
        hEnergy->Fill(particle.Energy());
        return particle;
      });

    // Handle K* decays
    for (const auto& p : eventParticles)
    {
      if (p.GetName() == "K*")
      {
        // K* decays into pi-K
        const double x{gRandom->Uniform()};

        Particle p1{};
        Particle p2{};

        if (x < 0.5)
        {
          p1.SetIndex("pi+");
          p2.SetIndex("K-");
        }
        else
        {
          p1.SetIndex("pi-");
          p2.SetIndex("K+");
        }

        p.Decay2Body(p1, p2);

        eventParticles.push_back(p1);
        eventParticles.push_back(p2);
        // K* must not be removed from the eventParticles vector
      }
    }

    for (size_t i{0}; i < eventParticles.size(); i++)
    {
      if (eventParticles[i].GetName() == "K*")
      {
        continue;
      }
      for (size_t j{i + 1}; j < eventParticles.size(); j++)
      {
        if (eventParticles[j].GetName() == "K*")
        {
          continue;
        }

        // Calculate the invariant mass of every couple of particles of the event
        // Note that particles of type K* are not considered, as they are decayed
        const double invMass{eventParticles[i].InvMass(eventParticles[j])};

        hInvMass->Fill(invMass);

        // If the product of the charges is > 0, the sign is the same; if it's < 0, the sign is discordant
        if (eventParticles[i].GetCharge() * eventParticles[j].GetCharge() > 0)
        {
          hInvMassSameSign->Fill(invMass);
        }
        else if (eventParticles[i].GetCharge() * eventParticles[j].GetCharge() < 0)
        {
          hInvMassDiscSign->Fill(invMass);
```

```cpp
        }
        else
        {
          std::cerr << "A particle with no charge was entered" << std::endl;
        }

        // check if the particles are pi-K, with opposite signs
        if ((eventParticles[i].GetName() == "pi+" && eventParticles[j].GetName() == "K-"
    ) ||
          (eventParticles[i].GetName() == "K-" && eventParticles[j].GetName() == "pi+")
    ||
          (eventParticles[i].GetName() == "pi-" && eventParticles[j].GetName() == "K+")
    ||
          (eventParticles[i].GetName() == "K+" && eventParticles[j].GetName() == "pi-"))
        {
          hInvMassPiKDisc->Fill(invMass);
        }

        // check if the particles are pi-K, with the same sign
        if ((eventParticles[i].GetName() == "pi+" && eventParticles[j].GetName() == "K+"
    ) ||
          (eventParticles[i].GetName() == "K+" && eventParticles[j].GetName() == "pi+")
    ||
          (eventParticles[i].GetName() == "pi-" && eventParticles[j].GetName() == "K-")
    ||
          (eventParticles[i].GetName() == "K-" && eventParticles[j].GetName() == "pi-"))
        {
          hInvMassPiKSame->Fill(invMass);
        }
      }
    }

    // Calculate the invariant mass of the 2 decay products of the K* particles
    for (size_t i{numParts}; i < eventParticles.size(); i += 2)
    {
      const double invMass{eventParticles[i].InvMass(eventParticles[i + 1])};
      hInvMassDecayProd->Fill(invMass);
    }
  }

  hInvMassSubAll->Add(hInvMassDiscSign, hInvMassSameSign, 1., -1.);
  hInvMassSubPiK->Add(hInvMassPiKDisc, hInvMassPiKSame, 1., -1.);

  // Open a file to save the histograms
  auto* outFile{TFile::Open("histos.root", "RECREATE")};

  hParticleTypes->Write();
  hPhi->Write();
  hTheta->Write();
  hImpulse->Write();
  hTransverseImpulse->Write();
  hEnergy->Write();
  hInvMass->Write();
  hInvMassSameSign->Write();
  hInvMassDiscSign->Write();
  hInvMassPiKSame->Write();
  hInvMassPiKDisc->Write();
  hInvMassDecayProd->Write();
  hInvMassSubAll->Write();
  hInvMassSubPiK->Write();

  outFile->Close();
}
```

Listing 7: File `main.cpp`

A.8 histos.C

```cpp
#include <TCanvas.h>
#include <TF1.h>
#include <TFile.h>
#include <TH1.h>
```

```cpp
#include <TROOT.h>
#include <TStyle.h>

#include <array>
#include <iostream>
#include <string>

void setStyle()
{
  gROOT->SetStyle("Plain");
  gStyle->SetPalette(57);
  gStyle->SetOptTitle(1);
  gStyle->SetOptFit(1);

  // Don't display canvases on screen
  gROOT->SetBatch(kTRUE);
}

void setFitStyle(TF1* f)
{
  f->SetLineColor(kRed);
  f->SetLineStyle(1);
  f->SetLineWidth(2);
}

void histos()
{
  auto* inFile{new TFile{"build/histos.root"}};

  // Load histograms from file
  auto* hParticleTypes{dynamic_cast<TH1I*>(inFile->Get("hParticleTypes"))};
  auto* hPhi{dynamic_cast<TH1D*>(inFile->Get("hPhi"))};
  auto* hTheta{dynamic_cast<TH1D*>(inFile->Get("hTheta"))};
  auto* hImpulse{dynamic_cast<TH1D*>(inFile->Get("hImpulse"))};
  auto* hTransverseImpulse{dynamic_cast<TH1D*>(inFile->Get("hTransverseImpulse"))};
  auto* hEnergy{dynamic_cast<TH1D*>(inFile->Get("hEnergy"))};
  auto* hInvMass{dynamic_cast<TH1D*>(inFile->Get("hInvMass"))};
  auto* hInvMassSameSign{dynamic_cast<TH1D*>(inFile->Get("hInvMassSameSign"))};
  auto* hInvMassDiscSign{dynamic_cast<TH1D*>(inFile->Get("hInvMassDiscSign"))};
  auto* hInvMassPiKSame{dynamic_cast<TH1D*>(inFile->Get("hInvMassPiKSame"))};
  auto* hInvMassPiKDisc{dynamic_cast<TH1D*>(inFile->Get("hInvMassPiKDisc"))};
  auto* hInvMassDecayProd{dynamic_cast<TH1D*>(inFile->Get("hInvMassDecayProd"))};
  auto* hInvMassSubAll{dynamic_cast<TH1D*>(inFile->Get("hInvMassSubAll"))};
  auto* hInvMassSubPiK{dynamic_cast<TH1D*>(inFile->Get("hInvMassSubPiK"))};

  const std::array<std::string, 7> particleTypes{"pi+", "pi-", "K+", "K-", "p+", "p-", "K*"};

  std::cout << "Particles generated by type:\n";
  for (size_t i{0}; i < particleTypes.size(); ++i)
  {
    std::cout << particleTypes[i] << ": " << hParticleTypes->GetBinContent(i + 1) << " +/- "
           << hParticleTypes->GetBinError(i + 1) << '\n';
  }

  // Set histograms style
  setStyle();

  auto* fUnifTheta{new TF1{"fUnifTheta", "pol0", 0., TMath::Pi()}};
  setFitStyle(fUnifTheta);
  fUnifTheta->SetParameter(0, 20000);
  hTheta->Fit(fUnifTheta, "Q");
  std::cout << "Theta fit: " << fUnifTheta->GetParameter(0) << " +/- " << fUnifTheta->GetParError(0) << '\n';
  std::cout << "Theta chi2/NDF: " << fUnifTheta->GetChisquare() / fUnifTheta->GetNDF() << '\n';
  std::cout << "Theta chi2 prob: " << fUnifTheta->GetProb() << '\n';
  std::cout << "Theta NDF: " << fUnifTheta->GetNDF() << '\n';
  std::cout << "Theta chi2: " << fUnifTheta->GetChisquare() << '\n';

  auto* fUnifPhi{new TF1{"fUnifPhi", "pol0", 0., TMath::TwoPi()}};
  setFitStyle(fUnifPhi);
```

```
fUnifPhi->SetParameter(0, 20000);
hPhi->Fit(fUnifPhi, "Q");
std::cout << "Phi fit: " << fUnifPhi->GetParameter(0) << " +/- " << fUnifPhi->GetParError(0) << '\n';
std::cout << "Phi chi2/NDF: " << fUnifPhi->GetChisquare() / fUnifPhi->GetNDF() << '\n';
std::cout << "Phi chi2 prob: " << fUnifPhi->GetProb() << '\n';
std::cout << "Phi NDF: " << fUnifPhi->GetNDF() << '\n';
std::cout << "Phi chi2: " << fUnifPhi->GetChisquare() << '\n';

auto* fExpImpulse{new TF1{"fExpImpulse", "expo", 0., 5.}};
setFitStyle(fExpImpulse);
fExpImpulse->SetLineWidth(1);
fExpImpulse->SetParameter(0, 1.);
fExpImpulse->SetParameter(1, 1.);
hImpulse->Fit(fExpImpulse, "Q");
std::cout << "Impulse fit amplitude: " << fExpImpulse->GetParameter(0) << " +/- " << fExpImpulse->GetParError(0)
      << '\n';
std::cout << "Impulse fit decay: " << fExpImpulse->GetParameter(1) << " +/- " << fExpImpulse->GetParError(1)
      << '\n';
std::cout << "Impulse chi2/NDF: " << fExpImpulse->GetChisquare() / fExpImpulse->GetNDF() << '\n';
std::cout << "Impulse chi2 prob: " << fExpImpulse->GetProb() << '\n';
std::cout << "Impulse NDF: " << fExpImpulse->GetNDF() << '\n';
std::cout << "Impulse chi2: " << fExpImpulse->GetChisquare() << '\n';

auto* fGausAll{new TF1{"fGausAll", "gausn", 0., 5.}};
setFitStyle(fGausAll);
fGausAll->SetParameter(0, 800.);
fGausAll->SetParameter(1, 0.9);
fGausAll->SetParameter(2, 0.05);
hInvMassSubAll->Fit(fGausAll, "Q");
for (int i{0}; i < 3; ++i)
{
  std::cout << "GausAll fit parameter " << i << ": " << fGausAll->GetParameter(i) << " +/- "
        << fGausAll->GetParError(i) << '\n';
}
std::cout << "GausAll chi2/NDF: " << fGausAll->GetChisquare() / fGausAll->GetNDF() << '\n';
std::cout << "GausAll chi2 prob: " << fGausAll->GetProb() << '\n';
std::cout << "GausAll NDF: " << fGausAll->GetNDF() << '\n';
std::cout << "GausAll chi2: " << fGausAll->GetChisquare() << '\n';

auto* fGausPiK{new TF1{"fGausPiK", "gausn", 0., 5.}};
setFitStyle(fGausPiK);
fGausPiK->SetParameter(0, 800.);
fGausPiK->SetParameter(1, 0.9);
fGausPiK->SetParameter(2, 0.05);
hInvMassSubPiK->Fit(fGausPiK, "Q");
for (int i{0}; i < 3; ++i)
{
  std::cout << "GausPiK fit parameter " << i << ": " << fGausPiK->GetParameter(i) << " +/- "
        << fGausPiK->GetParError(i) << '\n';
}
std::cout << "GausPiK chi2/NDF: " << fGausPiK->GetChisquare() / fGausPiK->GetNDF() << '\n';
std::cout << "GausPiK chi2 prob: " << fGausPiK->GetProb() << '\n';
std::cout << "GausPiK NDF: " << fGausPiK->GetNDF() << '\n';
std::cout << "GausPiK chi2: " << fGausPiK->GetChisquare() << '\n';

auto* fGausDecayProd{new TF1{"fGausDecayProd", "gausn", 0., 5.}};
setFitStyle(fGausDecayProd);
fGausDecayProd->SetLineWidth(1);
fGausDecayProd->SetParameter(0, 800.);
fGausDecayProd->SetParameter(1, 0.9);
fGausDecayProd->SetParameter(2, 0.05);
hInvMassDecayProd->Fit(fGausDecayProd, "Q");
for (int i{0}; i < 3; ++i)
{
```

```
    std::cout << "GausDecayProd fit parameter " << i << ": " << fGausDecayProd->
    GetParameter(i) << " +/- "
          << fGausDecayProd->GetParError(i) << '\n';
  }
  std::cout << "GausDecayProd chi2/NDF: " << fGausDecayProd->GetChisquare() /
    fGausDecayProd->GetNDF() << '\n';
  std::cout << "GausDecayProd chi2 prob: " << fGausDecayProd->GetProb() << '\n';
  std::cout << "GausDecayProd NDF: " << fGausDecayProd->GetNDF() << '\n';
  std::cout << "GausDecayProd chi2: " << fGausDecayProd->GetChisquare() << '\n';

  // Draw each histo in a canvas
  std::array<TCanvas*, 14> canvases;

  canvases[0] = new TCanvas{"cParticleTypes", "Particle types", 800, 600};
  hParticleTypes->Draw();
  canvases[1] = new TCanvas{"cPhi", "Phi", 800, 600};
  hPhi->Draw();
  canvases[2] = new TCanvas{"cTheta", "Theta", 800, 600};
  hTheta->Draw();
  canvases[3] = new TCanvas{"cImpulse", "Impulse", 800, 600};
  hImpulse->Draw();
  canvases[4] = new TCanvas{"cTransverseImpulse", "Transverse impulse", 800, 600};
  hTransverseImpulse->Draw();
  canvases[5] = new TCanvas{"cEnergy", "Energy", 800, 600};
  hEnergy->Draw();
  canvases[6] = new TCanvas{"cInvMass", "Invariant mass", 800, 600};
  hInvMass->Draw();
  canvases[7] = new TCanvas{"cInvMassSameSign", "Invariant mass same sign", 800, 600};
  hInvMassSameSign->Draw();
  canvases[8] = new TCanvas{"cInvMassDiscSign", "Invariant mass disc sign", 800, 600};
  hInvMassDiscSign->Draw();
  canvases[9] = new TCanvas{"cInvMassPiKSame", "Invariant mass pi K same", 800, 600};
  hInvMassPiKSame->Draw();
  canvases[10] = new TCanvas{"cInvMassPiKDisc", "Invariant mass pi K disc", 800, 600};
  hInvMassPiKDisc->Draw();
  canvases[11] = new TCanvas{"cInvMassDecayProd", "Invariant mass decay products", 800,
    600};
  hInvMassDecayProd->Draw();
  canvases[12] = new TCanvas{"cInvMassSubAll", "Invariant mass sub all", 800, 600};
  hInvMassSubAll->Draw();
  canvases[13] = new TCanvas{"cInvMassSubPiK", "Invariant mass sub pi K", 800, 600};
  hInvMassSubPiK->Draw();

  // Save each canvas as pdf, C and root file
  system("mkdir -p build/pdf");
  system("mkdir -p build/C");
  system("mkdir -p build/root");
  for (auto* canvas : canvases)
  {
    canvas->Print((std::string{"build/pdf/"} + canvas->GetName() + ".pdf").c_str());
    canvas->Print((std::string{"build/C/"} + canvas->GetName() + ".C").c_str());
    canvas->Print((std::string{"build/root/"} + canvas->GetName() + ".root").c_str());
  }

  auto* particles{new TCanvas{"particles", "Particle parameters", 800, 600}};
  particles->Divide(2, 2);
  particles->cd(1);
  hParticleTypes->Draw();
  particles->cd(2);
  hImpulse->Draw();
  fExpImpulse->Draw("SAME");
  particles->cd(3);
  hTheta->GetYaxis()->SetTitleOffset(1.6);
  hTheta->Draw();
  fUnifTheta->Draw("SAME");
  particles->cd(4);
  hPhi->GetYaxis()->SetTitleOffset(1.6);
  hPhi->Draw();
  fUnifPhi->Draw("SAME");

  auto* invMass{new TCanvas{"invMass", "Invariant masses", 800, 600}};
  invMass->Divide(1, 3);
  invMass->cd(1);
```

```
  hInvMassDecayProd->Draw();
  fGausDecayProd->Draw("SAME");
  invMass->cd(2);
  hInvMassSubAll->Draw();
  fGausAll->Draw("SAME");
  invMass->cd(3);
  hInvMassSubPiK->Draw();
  fGausPiK->Draw("SAME");

  particles->Print((std::string{"build/"} + particles->GetName() + ".pdf").c_str());
  invMass->Print((std::string{"build/"} + invMass->GetName() + ".pdf").c_str());
}
```

Listing 8: Macro ROOT `histos.C`