Relazione di Laboratorio - C++/ROOT

Damiano Scevola

16/11/2021

1 Introduzione

Il programma descritto nella presente relazione ha lo scopo di simulare ed analizzare eventi fisici risultanti da collisioni di particelle elementari. Nello specifico, ci proponiamo di generare un numero statisticamente significativo di eventi in ognuno dei quali più particelle di diverso tipo e con impulsi diversi collidono tra loro. Alcune di queste particelle sono risonanze, e a causa della loro natura instabile decadono in altre particelle elementari. Analizzando le distribuzioni di massa invariante, si rileva il segnale delle risonanze, da cui si possono ricavare la massa e la larghezza che risultano compatibili con quelle imposte in fase di generazione.

2 Struttura del Codice

Il codice è strutturato in tre classi (`ParticleType`, `ResonanceType` e `Particle`), due macro (`GenerateParticles` e `AnalyzeData`) e un file di definizione delle costanti utilizzate per favorire la leggibilità del codice (`Parameters.h`). In appendice è riportato il listato del codice.

Come mostrato nella figura 1, che riporta il diagramma delle classi, `ResonanceType` eredita da `ParticleType` poichè legate concettualmente da una relazione "*is a*", infatti ogni risonanza *è una* particella. La classe `Particle` è invece legata per composizione a `ParticleType` attraverso l'attributo statico `fParticleType`, che è un array di puntatori ad oggetti di tipo `ParticleType`. Poichè `ResonanceType` estende `ParticleType`, gli elementi all'interno di `fParticleType` possono essere anche di tipo `ResonanceType`, la relazione di composizione si estende anche a quest'ultima classe.

Le classi `ParticleType` e `ResonanceType` rappresentano un determinato tipo di particella, con nome (`fName`), massa (`fMass`), carica (`fCharge`) ed eventualmente larghezza (`fWidth`). I relativi attributi sono dichiarati `const` poichè una volta inizializzati non possono essere modificati durante l'esecuzione, essendo caratteristiche intrinseche del tipo di particella considerato. I metodi costruttori si occupano quindi di inizializzare tali attributi mediante liste di inizializzazione. Come di norma, gli attributi della classe padre sono dichiarati con visibilità `protected` per permettere alla classe figlia di accedervi, e quelli di quest'ultima `private`. Per ottenere o il valore di tali attributi sono presenti i metodi getter. Sono presenti inoltre i metodi `Print()` per stampare su terminale i vari attributi. Da notare infine che i metodi `GetWidth` e `Print` sono `virtual` per garantire il polimorfismo degli oggetti di tipo `ResonanceType` quando si accede agli elementi dell'array `fParticleType`, dichiarato di tipo `ParticleType*`.

La classe `Particle` rappresenta, invece, una particella vera e propria, infatti sono presenti gli attributi relativi alle tre componenti dell'impulso `fPx`, `fPy` e `fPz` con i relativi metodi getter e setter. Per definire i tipi delle particelle è presente il metodo `AddParticleType`, che aggiunge un elemento (di tipo `ResonanceType` se la larghezza non è nulla, `ParticleType` altrimenti)

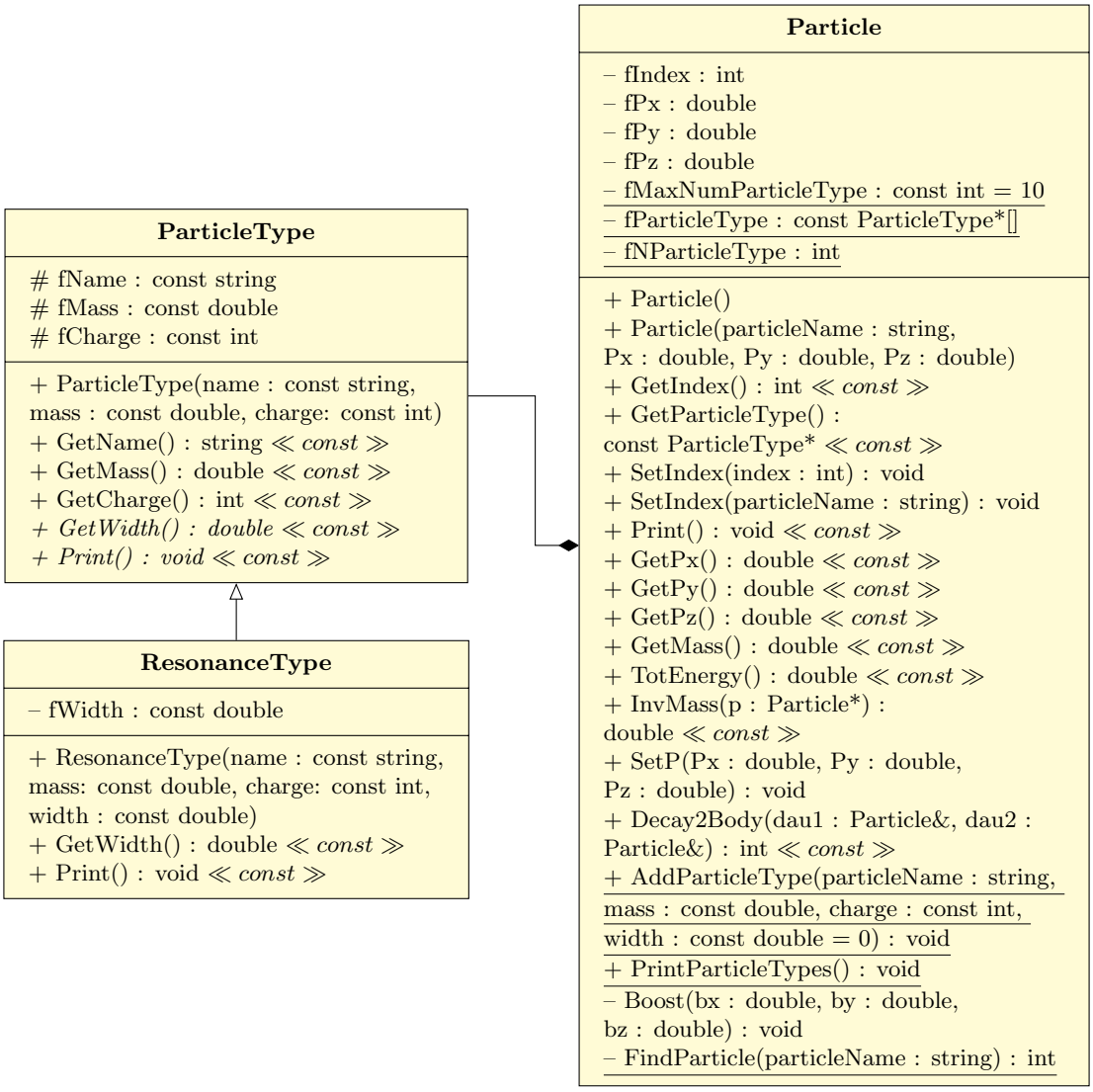

Figura 1: Diagramma UML delle classi: `ResonanceType` eredita da `ParticleType` e `Particle` ha un attributo statico che per composizione è un array contenente istanze delle due classi precedenti. Secondo la notazione UML, i membri contrassegnati con + sono *public*, quelli con - *private* e quelli con # *protected*. Il tipo è indicato dopo i due punti. I membri sottolineati sono *static* e quelli *corsivo* sono *virtual*.

all'array `fParticleType`. Il numero di elementi contenuti al suo interno è pari al valore di `fNParticleType`, che può arrivare al massimo a `fMaxNumParticleType`, e il contenuto dell'array può essere stampato su terminale con il metodo statico `PrintParticleTypes`. Ogni particella istanziata avrà un determinato tipo a seconda del valore dell'attributo `fIndex`, che rappresenta l'indice del tipo della particella nell'array `fParticleType` e che viene settato con il metodo `SetIndex`, di cui sono presenti due overload. Per ottenere l'indice di un tipo di particella a partire dalla stringa del suo nome è presente il metodo `FindParticle`.

Oltre al metodo `Print`, che serve per stampare su terminale il tipo e l'impulso della particella, sono presenti i metodi `GetMass` e `TotEnergy`, che restituiscono rispettivamente la massa (data dal tipo) e l'energia relativistica della particella. Il metodo `InvMass`, invece, restituisce la massa invariante del sistema costituito dalla particella attuale e da quella passata come parametro. Il metodo `Decay2Body`, infine, presi come parametri per riferimento due oggetti di tipo `Particle` non inizializzati (`fIndex == -1`), simula il decadimento della particella e inserisce i dati delle figlie in questi due oggetti, richiamando il metodo `Boost` durante la procedura.

3 Generazione

4 Analisi

Appendice

Parameters.h

```
#include <string>

using namespace std;

#ifndef PARAMETERS_H
#define PARAMETERS_H

const int N_PARTICLE_TYPES = 7;
const int N_ITERATIONS = 1E5;
const int N_PARTICLES_PER_ITERATION = 100;
const int MAX_PRODUCTS = 200;
const double AVG_P = 1.0;
const int N_BINS = 200;
const double MAX_MOMENTUM = 5.0;
const double MAX_ENERGY = 8.0;
const double MIN_INVARIANT_MASS = 0.0;
const double MAX_INVARIANT_MASS = 2.0;
const double MIN_INVARIANT_MASS_PEAK = 0.5;
const double MAX_INVARIANT_MASS_PEAK = 1.5;
const double ERROR_FACTOR = 3.0;

const int PION_PLUS_BIN = 1;
const int PION_MINUS_BIN = 2;
const int KAON_PLUS_BIN = 3;
const int KAON_MINUS_BIN = 4;
const int PROTON_PLUS_BIN = 5;
const int PROTON_MINUS_BIN = 6;
const int KAON_STAR_BIN = 7;

const double PION_PLUS_PROB = 0.4;
const double PION_MINUS_PROB = 0.4;
const double KAON_PLUS_PROB = 0.05;
```

```
const double KAON_MINUS_PROB = 0.05;
const double PROTON_PLUS_PROB = 0.045;
const double PROTON_MINUS_PROB = 0.045;
const double KAON_STAR_PROB = 0.01;

const double PROBABILITIES[] = {
    PION_PLUS_PROB,
    PION_MINUS_PROB,
    KAON_PLUS_PROB,
    KAON_MINUS_PROB,
    PROTON_PLUS_PROB,
    PROTON_MINUS_PROB,
    KAON_STAR_PROB
};

const string PION_PLUS_LABEL = "π+";
const string PION_MINUS_LABEL = "π-";
const string KAON_PLUS_LABEL = "K+";
const string KAON_MINUS_LABEL = "K-";
const string PROTON_PLUS_LABEL = "p+";
const string PROTON_MINUS_LABEL = "p-";
const string KAON_STAR_LABEL = "K*";

const string LABELS[] = {
    PION_PLUS_LABEL,
    PION_MINUS_LABEL,
    KAON_PLUS_LABEL,
    KAON_MINUS_LABEL,
    PROTON_PLUS_LABEL,
    PROTON_MINUS_LABEL,
    KAON_STAR_LABEL
};

const double PION_PLUS_CUMULATIVE = PION_PLUS_PROB;
const double PION_MINUS_CUMULATIVE = PION_PLUS_CUMULATIVE + PION_MINUS_PROB;
const double KAON_PLUS_CUMULATIVE = PION_MINUS_CUMULATIVE + KAON_PLUS_PROB;
const double KAON_MINUS_CUMULATIVE = KAON_PLUS_CUMULATIVE + KAON_MINUS_PROB;
const double PROTON_PLUS_CUMULATIVE = KAON_MINUS_CUMULATIVE + PROTON_PLUS_PROB;
const double PROTON_MINUS_CUMULATIVE = PROTON_PLUS_CUMULATIVE +
↪  PROTON_MINUS_PROB;
const double KAON_STAR_CUMULATIVE = PROTON_MINUS_CUMULATIVE + KAON_STAR_PROB;

#endif
```

ParticleType.h

```
#ifndef PARTICLE_TYPE_H
#define PARTICLE_TYPE_H
```

```
using namespace std;

class ParticleType {
    public:
        ParticleType(const string name, const double mass, const int charge) :
            fName(name), fMass(mass), fCharge(charge) {}
        string GetName() const;
        double GetMass() const;
        int GetCharge() const;
        virtual double GetWidth() const;
        virtual void Print() const;

    protected:
        const string fName;
        const double fMass;
        const int fCharge;
};

#endif
```

ParticleType.cpp

```
#include "ParticleType.h"

#include <iostream>

using namespace std;

string ParticleType::GetName() const {
    return fName;
}

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

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

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

void ParticleType::Print() const {
    std::cout << "Particle Type: " << fName << std::endl <<
                 "\tMass = " << fMass << " MeV/c^2" << std::endl <<
                 "\tCharge = " << fCharge << " e" << std::endl;
}
```

ResonanceType.h

```
#include "ParticleType.h"

#ifndef RESONANCE_TYPE_H
#define RESONANCE_TYPE_H

using namespace std;

class ResonanceType : public ParticleType {
    public:
        ResonanceType(const string name, const double mass, const int charge,
        ↪  const double width) :
            ParticleType(name, mass, charge), fWidth(width) {}
        double GetWidth() const;
        void Print() const;

    private:
        const double fWidth;
};

#endif
```

ResonanceType.cpp

```
#include "ResonanceType.h"

#include "ParticleType.h"
#include <iostream>

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

void ResonanceType::Print() const {
    ParticleType::Print();
    std::cout << "\tWidth = " << fWidth << " s^-1" << std::endl;
}
```

Particle.h

```
#include "ParticleType.h"

#ifndef PARTICLE_H
#define PARTICLE_H

using namespace std;

class Particle {
    public:
```

```
        Particle();
        Particle(string particleName, double Px, double Py, double Pz);
        int GetIndex() const;
        const ParticleType *GetParticleType() const;
        void SetIndex(int index);
        void SetIndex(string particleName);
        void Print() const;
        double GetPx() const;
        double GetPy() const;
        double GetPz() const;
        double GetMass() const;
        double TotEnergy() const;
        double InvMass(Particle *p) const;
        void SetP(double Px, double Py, double Pz);
        int Decay2Body(Particle &dau1, Particle &dau2) const;

        static void AddParticleType(string particleName, const double mass,
        ↪  const int charge, const double width = 0);
        static void PrintParticleTypes();

    private:
        int fIndex;
        double fPx, fPy, fPz;

        static const int fMaxNumParticleType = 10;
        static const ParticleType *fParticleType[];
        static int fNParticleType;

        static int FindParticle(string particleName);

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

#endif
```

Particle.cpp

```
#include "Particle.h"

#include "ParticleType.h"
#include "ResonanceType.h"
#include <string>
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

int Particle::fNParticleType = 0;
```

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

int Particle::FindParticle(string particleName) {
    for (int i = 0; i < fNParticleType; i++)
        if (fParticleType[i]->GetName() == particleName)
            return i;
    return -1;
}

Particle::Particle() {
    fIndex = -1;
}

Particle::Particle(string particleName, double Px = 0, double Py = 0, double Pz
↪  = 0) {
    fIndex = FindParticle(particleName);
    if (fIndex < 0)
        std::cout << "Particle " << particleName << " not found" << std::endl;
    fPx = Px;
    fPy = Py;
    fPz = Pz;
}

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

const ParticleType *Particle::GetParticleType() const {
    return fParticleType[fIndex];
}

void Particle::SetIndex(int index) {
    if (index >= fNParticleType) {
        std::cout << "Cannot set index " << index << ": out of bounds error" <<
        ↪  std::endl;
        return;
    }
    fIndex = index;
}

void Particle::SetIndex(string particleName) {
    int index = FindParticle(particleName);
    if (index < 0) {
        std::cout << "Cannot set index for searched particle " << particleName
        ↪  << ": particle not found" << std::endl;
        return;
    }
    fIndex = index;
}
```

```
void Particle::AddParticleType(const string particleName, const double mass,
↪  const int charge, const double width) {
    if (fNParticleType >= fMaxNumParticleType) {
        std::cout << "Maximum number of particle types reached: cannot add new
        ↪  particle type" << std::endl;
        return;
    }

    const int index = FindParticle(particleName);
    if (index >= 0) {
        std::cout << "Particle type " << particleName << " already exists:
        ↪  cannot add duplicate" << std::endl;
        return;
    }

    fParticleType[fNParticleType] = width == 0 ? new ParticleType(particleName,
    ↪  mass, charge) : new ResonanceType(particleName, mass, charge, width);
    fNParticleType++;
}

void Particle::PrintParticleTypes() {
    for (int i = 0; i < fNParticleType; i++)
        fParticleType[i]->Print();
}

void Particle::Print() const {
    std::cout << fParticleType[fIndex]->GetName() << " [index = " << fIndex <<
    ↪  "]" << std::endl <<
                "\tP = (" << fPx << ", " << fPy << ", " << fPz << ")" <<
                ↪  std::endl;
}

double Particle::GetPx() const {
    return fPx;
}

double Particle::GetPy() const {
    return fPy;
}

double Particle::GetPz() const {
    return fPz;
}

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

```
double Particle::TotEnergy() const {
    return sqrt(pow(GetMass(), 2) + pow(fPx, 2) + pow(fPy, 2) + pow(fPz, 2));
}

double Particle::InvMass(Particle *p) const {
    return sqrt(pow(TotEnergy() + p->TotEnergy(), 2) - (pow(fPx + p->GetPx(),
    ↪  2) + pow(fPy + p->GetPy(), 2) + pow(fPz + p->GetPz(), 2)));
}

void Particle::SetP(double Px, double Py, double Pz) {
    fPx = Px;
    fPy = Py;
    fPz = 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();

    // add width effect
    if (fIndex > -1) {
        // 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[fIndex]->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(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 = TotEnergy();

    // 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;
}
```

GenerateParticles.cpp

AnalyzeData.cpp