Particle Detector Simulation – PHYS30762 C++ Project

Harry Taylor
(Dated: May 10, 2024)

The aim of this project was to create a simulation of a particle detector using classes for the different particles and sub-detectors. A `detector` class took an `event` class as an input and printed its findings to the terminal. The `detector` class acted as a container for the different sub detectors, whilst the `event` class acted as a container for the particles in each event.

1. INTRODUCTION

Particle detectors are used to test theories about our current understanding of particle physics. They do this by gathering information about a particle so that it can be identified. Modern detectors gather this information via several layers of sub-detectors, of which there are two main types, tracking devices and calorimeters. Tracking devices detect charged particles by showing their path (or track) through the detector. Calorimeters detect particles by recording how much energy the particle loses as they pass through [1]. If these types of sub-detectors are used in tandem, particles can be identified based off of how they interact with each sub-detector.

The goal of this project was to use a series of sub-detectors, like those mentioned above, to simulate particle detection and identification. The user should be able to input a series of particles, called an 'event', into a detector, which contained several sub-detectors. These particles should then be simulated entering each sub-detector. Once this was done for all particles in the event, a print-out containing information relating to the input particles, sub-detectors, predicted particles and overall detector should be returned.

2. CODE DESIGN AND IMPLEMENTATION

In order to achieve this goal, several classes were made, both for the particles and sub-detectors. Using polymorphism, all particles were given general data members. These general data members were then inherited by derived classes, with each derived class implementing unique data members relating to that specific particle. The same was done for the sub-detectors. The sub-detectors were also given randomised efficiencies, to simulate a real detector more accurately.

To simplify the user experience, they would only interact with two classes, `event` and `detector`. These acted as containers for particles and sub-detectors respectively. This was done to abstract the implementation of each particle and sub-detector class and make the program more user friendly.

2.1. Particle classes

For this project, seven of the seventeen fundamental particles from the standard model [2] were included as well as two composite particles. These were as follows: *electron, muon, tau, electron neutrino, muon neutrino, tau neutrino, photon, neutron and proton* (and their respective anti-particles where possible). Each particle, with one notable exception, had its own class, as can be seen below in Figure 1. This allowed for a simpler experience during programming and also for the user.

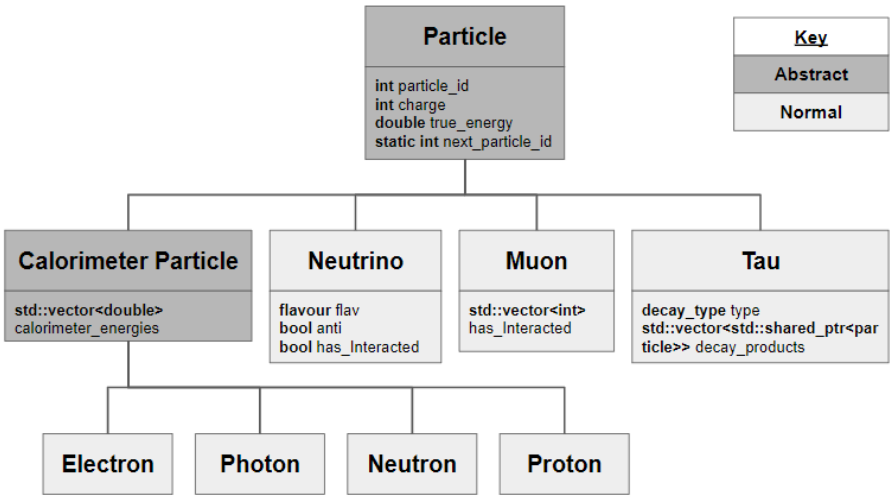

FIG. 1. Flow diagram of the hierarchy of the 'particle' type classes. The lower classes inherit from the higher ones, with the grey lines signifying base and derived class relationships. The bold lettering at the top of each box signifies the class name, whilst the text below each name shows that class' data members (not including those inherited).

2.1.1. particle

This was the abstract base class from which all other particle classes derived, and it contained three data members. Each particle required an ID, a charge, and a true energy. `particle_id` was used to keep track of which particle was in the simulation. This ID depended on the `next_particle_id`, which would be incremented each time a new particle was created. The `charge` determined whether or not the particle was an anti-particle, and the `true_energy` was equal to the energy (in GeV) that the particle entered the detector with.

The move and copy constructors and assignment operators were implemented such that the number of particles created would always be equal to `next_particle_id`. This was done by incrementing `next_particle_id` each time the copy constructor or operator was called, because the original particle still existed in this case and so the `particle_id` could not be copied. However, when the move constructor or operator was called, the `particle_id` was copied over to the new class, as in this case, the original particle no longer existed and so there were no repeat IDs. To make sure this happened for each derived class, the deep copy functionalities mentioned above were overrode each time a new class was programmed.

2.1.2. calorimeter_particle

There were four particles which interacted with the calorimeter and they were the `electron, photon, muon` and `proton`. These particles required a container of possible energies to deposit within the calorimeter. As these particles all had similar interactions, the abstract `calorimeter_particle` class was created, from which they all derived.

Depending on the given particle's ability to interact with the different layers of the calorimeter, the `true_energy` would be randomly split between the components of the `calorimeter_energies` vector. This vector had four components, the first two signifying interaction with the EM calorimeter layers, and the final two signifying interaction with the HAD calorimeter layers.

2.1.3. neutrino

The `neutrino` class was the exception mentioned earlier, as this one class encompassed all three types of neutrinos. This was done because all types of neutrinos acted the same and required the same data members. These types were still distinguishable however, as the data member `flav` is of the type `flavour`, which was an `enum class` of all possible neutrino flavours. Neutrinos were also unique in the fact that they had no charge, yet did have anti-particles, which required a new data member, `anti`, to show. Finally, the data member `has_interacted` was a flag signifying if the neutrino had interacted with any layers of the detector. This was purely cosmetic though, because the neutrino could not interact with any of the sub-detectors used.

2.1.4. muon

As the muon was the only particle capable of interacting with the muon-chamber, `has_interacted` was a vector containing two flags to show which layers of the muon-chamber it had interacted with. Initially a vector of `bool`s was used, but due to each `bool` only taking up one bit in memory when in an `std::vector`, this caused issues later when designing the `muon_chamber` class. For this reason, a vector of `int`s was used instead.

2.1.5. tau

Tau particles had two decay types, leptonic and hadronic, and each type determined the decay products. The data member `type` denoted this using an `enum class decay_type`, which can be either of the two decay modes. Depending on the value of `type`, the `decay_products` vector was filled with the particles resulting from the respective decay, using the `tau::set_decay_type()` member function, part of which is shown below.

```
// Decay type setter (sets decay products too)
void tau::set_decay_type(decay_type input)
{
  bool is_anti = (charge == -1) ? false : true;
  if(input == decay_type::leptonic)
  {
    type = input;
    // Tau decays to Electron, Electron Anti-Neutrino, Tau Neutrino
    // Charges flipped for Anti-Tau

    // Generate random energies for decay products
    std::vector<double> energies = generate_energies(true_energy, 3);

    // Create decay products and assign random energies
    decay_products.push_back(std::make_shared<electron>(is_anti, energies[0]));
    decay_products.push_back(std::make_shared<neutrino>(flavour::electron, !is_anti, energies[1]));
    decay_products.push_back(std::make_shared<neutrino>(flavour::tau, is_anti, energies[2]));
  }
```

FIG. 2. Code snippet showing the start of the `tau::set_decay_type()` member function. This section of the function deals with leptonic decays.

2.1.6. `event`

This class existed as a container for all of the particles in a single 'event'. To make the interface as user friendly as possible, the `event` constructor only required two inputs. One of the inputs was a `total_energy`, which was the total energy of all particles in the event. This energy would be randomly distributed amongst the particles in the event. The other input was a vector containing a `particle_type` for each particle desired in the event. This was an `enum class` that contained all of the names for the possible input particles. The `event` constructor would then use an `std::map` of lambda functions to instantiate the input particles, whilst assigning each a random energy. This map is shown below in Figure 3. Polymorhpism was used to contain all of the instantiated particles within an `std::vector` of `std::shared_ptr`. This meant that the detector could take an `event` as an input, which simplified the user experience greatly.

```
// Assign all values to the particle map, where each index of the map will be
// a particle type, and it will take an energy as an input, as to allow random
// energies to be assigned using lambda functions
particle_map[particle_type::electron] = [](double input) {return std::make_shared<electron>(false, input);};
particle_map[particle_type::antielectron] = [](double input) {return std::make_shared<electron>(true, input);};
particle_map[particle_type::electron_neutrino] = [](double input) {return std::make_shared<neutrino>(flavour::electron, false, input);};
particle_map[particle_type::electron_antineutrino] = [](double input) {return std::make_shared<neutrino>(flavour::electron, true, input);};
particle_map[particle_type::muon] = [](double input) {return std::make_shared<muon>(false, input);};
particle_map[particle_type::antimuon] = [](double input) {return std::make_shared<muon>(true, input);};
particle_map[particle_type::muon_neutrino] = [](double input) {return std::make_shared<neutrino>(flavour::muon, false, input);};
particle_map[particle_type::muon_antineutrino] = [](double input) {return std::make_shared<neutrino>(flavour::muon, true, input);};
particle_map[particle_type::leptonic_tau] = [](double input) {return std::make_shared<tau>(false, input, decay_type::leptonic);};
particle_map[particle_type::hadronic_tau] = [](double input) {return std::make_shared<tau>(false, input, decay_type::hadronic);};
particle_map[particle_type::leptonic_antitau] = [](double input) {return std::make_shared<tau>(true, input, decay_type::leptonic);};
particle_map[particle_type::hadronic_antitau] = [](double input) {return std::make_shared<tau>(true, input, decay_type::hadronic);};
particle_map[particle_type::tau_neutrino] = [](double input) {return std::make_shared<neutrino>(flavour::tau, false, input);};
particle_map[particle_type::tau_antineutrino] = [](double input) {return std::make_shared<neutrino>(flavour::tau, true, input);};
particle_map[particle_type::proton] = [](double input) {return std::make_shared<proton>(false, input);};
particle_map[particle_type::antiproton] = [](double input) {return std::make_shared<proton>(true, input);};
particle_map[particle_type::neutron] = [](double input) {return std::make_shared<neutron>(input);};
particle_map[particle_type::photon] = [](double input) {return std::make_shared<photon>(input);};
```

FIG. 3. Code snippet showing map of lambda functions used to instantiate each particle. Each function requires an energy as an input as well as a `particle_type` and it returns a pointer to an instantiated class of the desired particle.

2.2. Sub-Detector Classes

Three sub-detectors were chosen to be used in this project. They were as follows: a *tracker, calorimeter* and a *muon chamber*. These three were chosen because each particle had a unique interaction with the set of sub-detectors. This meant that it was possible for prediction to be accurate. As for the particles, each sub-detector had it's own class.

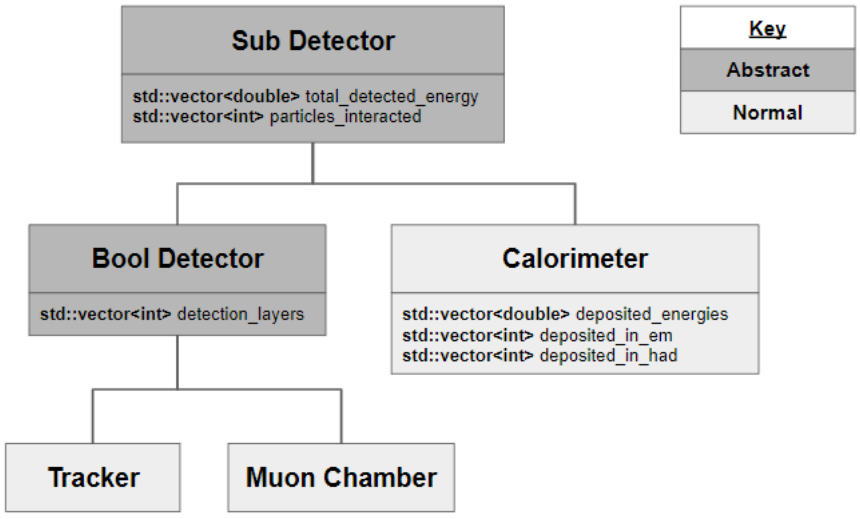

FIG. 4. Flow diagram of the hierarchy of the 'sub-detector' type classes. The lower classes inherit from the higher ones, with the grey lines signifying base and derived class relationships. The bold lettering at the top of each box signifies the class name, whilst the text below each name shows that class' data members (not including those inherited).

2.2.1. `sub_detector`

This was the abstract base class from which all sub-detectors derived. It contained two vectors, as can be seen in Figure 4, which acted as the memory of the sub-detector. `particles_interacted` was used to store the `particle_id` of all particles which could interact with the sub-detector. The other data member, `total_detected_energy`, contained the energies deposited by each particle. It was implemented such that the

particle_id of a given particle would be the index within the vector where its deposited energy was stored. Each sub-detector had a use_sub_detector() member function which took an std::shared_ptr<particle> as an input. This was used to simulate each particle entering each sub-detector.

2.2.2. bool_detector

The tracker and muon_chamber were both tracking devices, and so they worked in a similar way. Both sub-detectors had layers, each of which had a chance to detect the input particle. Both classes derived from the bool_detector abstract class. The data member detection_layers was a vector, whose size was chosen when the constructor was called, that simulated the layers within the sub-detector. The tracker had three layers whilst the muon_chamber had two. These derived classes also overrode the use_sub_detector() member function for each sub-detectors specific purpose. The tracker only detected charged particles whereas the muon_chamber only detected muons. When a particle was input into one of these sub-detectors, the detection layers would be randomly assigned to true or false (1 or 0 in actuality), and if enough layers were true, the particle would be classed as detected. If a particle was detected, an efficiency would be randomly chosen for each particle and its true energy would be multiplied by this efficiency. This 'detected energy' was then appended to the total_detected_energy vector.

2.2.3. calorimeter

Unlike the tracker and muon_chamber, the layers of the calorimeter had energy directly deposited into them. For this, the vector deposited_energies was used to contain the energy deposited in each of the four layers. As in the calorimeter_particle class, the first two components related to the EM layers, and the final two related to the HAD layers. Only particles derived from that class could deposit energy into the calorimeter layers. In a similar manner to the bool_detectors, the efficiency of each detection layer would be randomised for each particle. This meant that the energy deposited by the particle in each calorimeter layer would be a fraction of the true energy. Since some particles only deposited energy in certain portions of the calorimeter, the vectors deposited_in_em and deposited_in_had would record where the energy was deposited. These vectors were used later on when predicting the input particle.

2.2.4. detector

As the event class existed as a container for a set of particles, the detector class existed as a container for the sub-detectors. To construct, it took a vector of enum class sub_detector_type as an input for the configuration of the detector. The detector was also constructed in the same manner as the event, by using a map of lambda functions. Once the detector had been constructed, the detector::use_detector() function could be called with an event as an input. This function took each particle in the event and simulated it entering each of the sub-detectors. There was a special case for tau particles, as they interacted via their daughter particles, so the decay_products vector was accessed, and each of those particles were fed into the detector. After the event had simulated passing through the detector, the detector::predict_particles() function was called, which predicted the nature of each particle in the event based only on how it interacted with the sub-detectors. The detector then printed out all of the relevant data for each event, and then for all events.

```
// Add predicted particles to vector
if(tracker_detected)
{
  if(em_cal_detected && !had_cal_detected)
  {
    // A large invisible energy suggests this electron is accompanied
    // by many neutrinos, and is likely due to a leptonic tau
    if(invisible_energy >= total_true_energy/2)
      predicted_particles.push_back(particle_type::leptonic_tau);
    else
      predicted_particles.push_back(particle_type::electron);
  }
  else if(muon_cham_detected)
    predicted_particles.push_back(particle_type::muon);
  else if(em_cal_detected && had_cal_detected)
  {
    // A large invisible energy suggests this hadron is accompanied
    // by a neutrino, and is likely due to a hadronic tau
    if(invisible_energy >= total_true_energy/2)
      predicted_particles.push_back(particle_type::hadronic_tau);
    else
      predicted_particles.push_back(particle_type::proton);
  }
  else
    predicted_particles.push_back(particle_type::electron);
}
```

FIG. 5. Code snippet of a section of the detector::predict_particles() function. This section dealt with the prediction of the type of particle. To do this, earlier in the function it was determined which sub-detectors the particle had interacted with, and then based off of this, a prediction was made. The predictions were made per event, so each prediction was pushed back onto a predicted_particles vector which was later printed to the terminal

2.3. *Usage*

In order to simulate a new event passing through the detector, the user was required to do two things. Firstly, they must have created an event. This was done by deciding the `total_energy` of the event and then creating a vector of `particle_types` with all of the particles desired in the event. These components were then used to create an event, as shown below in Figure 6. Once the event was created, the second and final step was to add the event to the events vector, and then it would be passed through the detector.

```
// Event 5 (1 proton, 1 electron, 1 muon, 1000 GeV)
std::vector<particle_type> event_5_particles;
double event_5_energy = 1000; // GeV
event_5_particles.push_back(particle_type::proton);
event_5_particles.push_back(particle_type::electron);
event_5_particles.push_back(particle_type::muon);
event event5(event_5_energy, event_5_particles);

// Add all events to a vector
std::vector<event> events{event1, event2, event3, event4, event5};

// Run each event into the detector
for(auto it{events.begin()}; it != events.end(); it++)
  lhc_detector.use_detector(*it);

// Print data about all events
lhc_detector.print_total_data();
```

FIG. 6. Code snippet of the main.cpp file. This shows the creation of an event and it's subsequent simulation in the detector.

3. RESULTS

This program required no extra user input at the terminal, so once it was compiled and run, all of the data was printed. This began with a statement outlining the sub-detector configuration of the detector. Next, the data from each event was printed out, one event at a time.

```
Event 5:

Proton
ID: 15, Charge: 1, True Energy: 325 GeV,
Possible Energy Deposited in Calorimeter: [55.9, 68, 75.6, 125] GeV
  Particle Detected by Tracker: True
  Energy Deposited in the Calorimeter: [25.7, 62.8, 59.2, 48.3] GeV
  Particle Detected by Muon Chamber: False

Electron
ID: 16, Charge: -1, True Energy: 257 GeV,
Possible Energy Deposited in Calorimeter: [147, 110, 0, 0] GeV
  Particle Detected by Tracker: False
  Energy Deposited in the Calorimeter: [7.54, 47.6, 0, 0] GeV
  Particle Detected by Muon Chamber: False

Muon
ID: 17, Charge: -1, True Energy: 418 GeV,
Interacted with Inner Layer of Muon Chamber: False
Interacted with Outer Layer of Muon Chamber: False
  Particle Detected by Tracker: True
  Energy Deposited in the Calorimeter: [0, 0, 0, 0] GeV
  Particle Detected by Muon Chamber: False

Here is a list of particles which the detector predicts has passed through (in order):
Proton
Photon
Electron
```

```
Detector Details:
Invisible Energy: 230 GeV
Energy Efficiency: 77%, (770 GeV/1e+03 GeV)
Prediction Efficiency = 33.3% (1/3)


Summary After All Events Have Passed Through the Detector:

Tracker:
Number of Particles Interacted With: 5
Total Energy Detected: 1.61e+03 GeV

Calorimeter:
Number of Particles Interacted With: 10
Total Energy Detected: 1.28e+03 GeV

Muon Chamber:
Number of Particles Interacted With: 0
Total Energy Detected: 0 GeV
```

FIG. 7. Section of the printed results from simulated detection of the event described in Figure 6 (left image and top of right image). It also contains the total detector data for all events.

Figure 7 was an example layout for a simulated event. The data of each particle in the event was printed, starting with the general particle data like `particle_id`, `charge` and `true_energy`. This was followed by any data members unique to the specific particle, for example, the possible energies deposited in the calorimeter for an electron. Then, the sub-detector data was printed, documenting how each particle interacted with each sub-detector. As stated above, this information was then passed into the `detector::predict_particles()`

function, which printed a set of particles which the detector predicted has passed through. The last thing printed for each event was the detector data. This included the invisible energy, which was the energy discrepancy between the true and detected energies. It also included two detector efficiencies, one for the percentage of the true energy detected, and another for the percentage of particles correctly predicted.

After all events, the total event data was printed, as seen on the right of Figure 7. This included the total number of particles each sub-detector interacted with, and the total energy that each sub-detector had detected.

4. DISCUSSION AND CONCLUSIONS

To improve this project, a more sophisticated prediction function could be implemented, especially regarding tau particles. Also, most of the classes that had a inheritance chain length of two were redundant. They were used here to show polymorphism and inheritance, but they could be simplified by the use of an `enum class` like that used for the `neutrino`. An extension to the project could be including more particles such as quarks and W/Z bosons. Another extension could be simulating the collisions which produce the detected particles.

To conclude, the development of this particle detector simulation has provided valuable insights into the complexities of particle physics experimentation. Through the utilization of classes for particles and sub-detectors, coupled with the implementation of polymorphism and inheritance, I have created a versatile and user-friendly tool for simulating the particle detection and identification process.

According to Overleaf, this document contains 2350 words.

[1] CERN, How a detectors works, accessed on May 10, 2024.
[2] R. Oerter, *The Theory of Almost Everything: The Standard Model, the Unsung Triumph of Modern Physics* (Penguin Group, 2006).