Detector Simulation Project

Isaac Clemenger

School of Physics and Astronomy

University of Manchester

PHYS 30762 Object-Oriented Programming in C++ Final Project Report

May 2024

Abstract

This project simulates a simplified particle detector in C++. The particles considered are: electrons, muons, taus, neutrinos, protons, neutrons, and photons. They are detected one by one, simulating a "particle gun" like scenario. The code adheres to principles of object orientated programming, using inheritance chains for both the particles and detectors.

1 Introduction

The standard model of particle physics describes the interactions of the fundamental particles. It is the most successful theory of particle physics to date. Theorised throughout the 20^{th} century, it has been extensively tested, predominantly through collider experiments like the LHC at CERN [1]. At the LHC, high energy protons are collided, causing interactions between the particles that make up protons, called quarks. The motivation for simulating experiments like these, is that we can model the predictions made by the standard model in simulations, and compare them to real data. This allows us to search for new physics. In order to simplify this simulation, we will only consider the following particles: electron, muon, tau, neutrino, proton, neutron, and photon.

In this simulation, we model a general purpose detector, composed of a tracker, calorimeter, and muon chamber. These are designed to directly detect all particles except for neutrinos. The sub detectors are arranged in an onion like structure, with the tracker being the innermost part, the muon chamber being the outermost. The tracker is designed to detect charged particles. We will treat it as having three layers: the inner tracker, outer tracker, and the pixel strip. The calorimeter is designed to stop particles that enter it. Is has two main sections: the electromagnetic, EM, and hadronic, HAD. We will treat each of these sections as having two subsections, leaving the calorimeter with four subsections in total. As muons are so much heavier than electrons, they are not stopped by the calorimeter [2]. Instead they are detected in the muon chamber. We will consider this to have two layers, simply the inner and outer. It should be noted that the tau

is far heavier than both electrons and muons, and will decay into lighter particles before even reaching the tracker. 35% of the time the tau will decay leptonically, producing two neutrinos and either an electron or muon with equal probability [3]. The remaining 65% of decays are hadronic, which we will simplify as producing a neutrino and a set of calorimeter deposits.
In a real particle collision at a collider, many interactions will take place at once. We simplify this by modelling a "particle gun", firing individual particles through the detector one at a time.

2 Code Design and Implementation

2.1 Random Number Simulation

Random number generation, RNG, is used extensively throughout this project, to simulate the random nature of particle decays, as well as detection efficiencies. A namespace was created to store RNG functions, the header file for which is shown in Lisiting 1. The source and header files are split to adhere to abstraction [4].

```
1 namespace random
2 {
3   double gen_rand_0_and_1();
4   std::vector<double> gen_n_numbers(int n);
5   double gen_decimal_in_range(int min, int max);
6 }
```

Listing 1: Code for the header file of the random namespace

All of these functions use the C standard library function `rand()`, which returns an int. Thus, casting to a double is required for it to be used in calculations.
The function `gen_rand_0_and_1()` simply generates a number by `rand()`, and normalises it by `RANDMAX`. The `gen_n_numbers(int n)` function generates n random numbers which add up to one. We ensure this by normalising the generated numbers by their sum. The final function, `gen_decimal_in_range`, generates a random number in a decimal range. This was done using Equation 1,

$$\texttt{min + (rand() \% (max - min + 1))} \tag{1}$$

where % is the modulus operator. This works by the fact that if we divide a random number by our specified range + 1, the remainder, given be the % operator, will always be less than or equal to the range.

2.2 Designing Particle Classes

As stated in Section 1, we only consider seven particles in this project. We assign three base attributes for a general particle: id, an integer simply to track what particle is what; charge, also an integer; and energy, a double. The base particle class is abstract, thus adhering to polymorphism [4]. The approach taken for the particle inheritance chain is to split them up by

how they are detected. This is because particles will store the information about where they are detected, e.g. a particle detected in the calorimeter will have an associated object with its calorimeter deposits, and so these member variables need to distributed appropriately. Table 1 details where each particle is detected, forming the basis of the inheritance chain, which is displayed in Figure 1. Each class in the inheritance chain adheres to the rule of five.

Table 1: Table showing where each particle can be detected

Particle	Tracker	EM Calorimeter	HAD Calorimeter	Muon Chamber
Electron	Yes	Yes	No	No
Muon	Yes	No	No	Yes
Neutrino	No	No	No	No
Proton	Yes	Yes	Yes	No
Neutron	No	Yes	Yes	No
Photon	No	Yes	Yes	No

The methods of storing a particle's expected interactions with the detectors differed from detector to detector. For the tracker and muon chamber, a vector of booleans simply stating what layers the particles interacted in suffices, we call this `tracks_detected`. However for the calorimeter, information on a particle's deposits in each sub section of the calorimeter is required. This is stored in the `calorimeter_deposits` pointer, as shown in Figure 1. The calorimeter deposits class contains four doubles, representing a particles deposits in EM1, EM2, HAD1, HAD2, which could be accessed via setters and getters.
As shown in Figure 1, `calorimeter_deposits` is a member variable of `calorimeter_interacting`, which publicly inherits `particle`. The setter for this class is shown in Listing 2.

```
1 // Setters
2 void Calorimeter_interacting::set_calorimeter_deposits(const double
      particle_energy)
3 {
4   if (calorimeter_deposits != nullptr)
5   {
6     std::vector<double> rand_energies = random::gen_n_numbers(4);
7     calorimeter_deposits->set_em1(rand_energies[0]*particle_energy);
8     calorimeter_deposits->set_em2(rand_energies[1]*particle_energy);
9     calorimeter_deposits->set_had1(rand_energies[2]*particle_energy);
10    calorimeter_deposits->set_had2(rand_energies[3]*particle_energy);
11  }
12 }
```

Listing 2: Code for the `set_calorimeter_deposits` function, which is part of the `Calorimeter_interacting` class. Code randomly distributes a particle's energy into the four sections of the calorimeter.

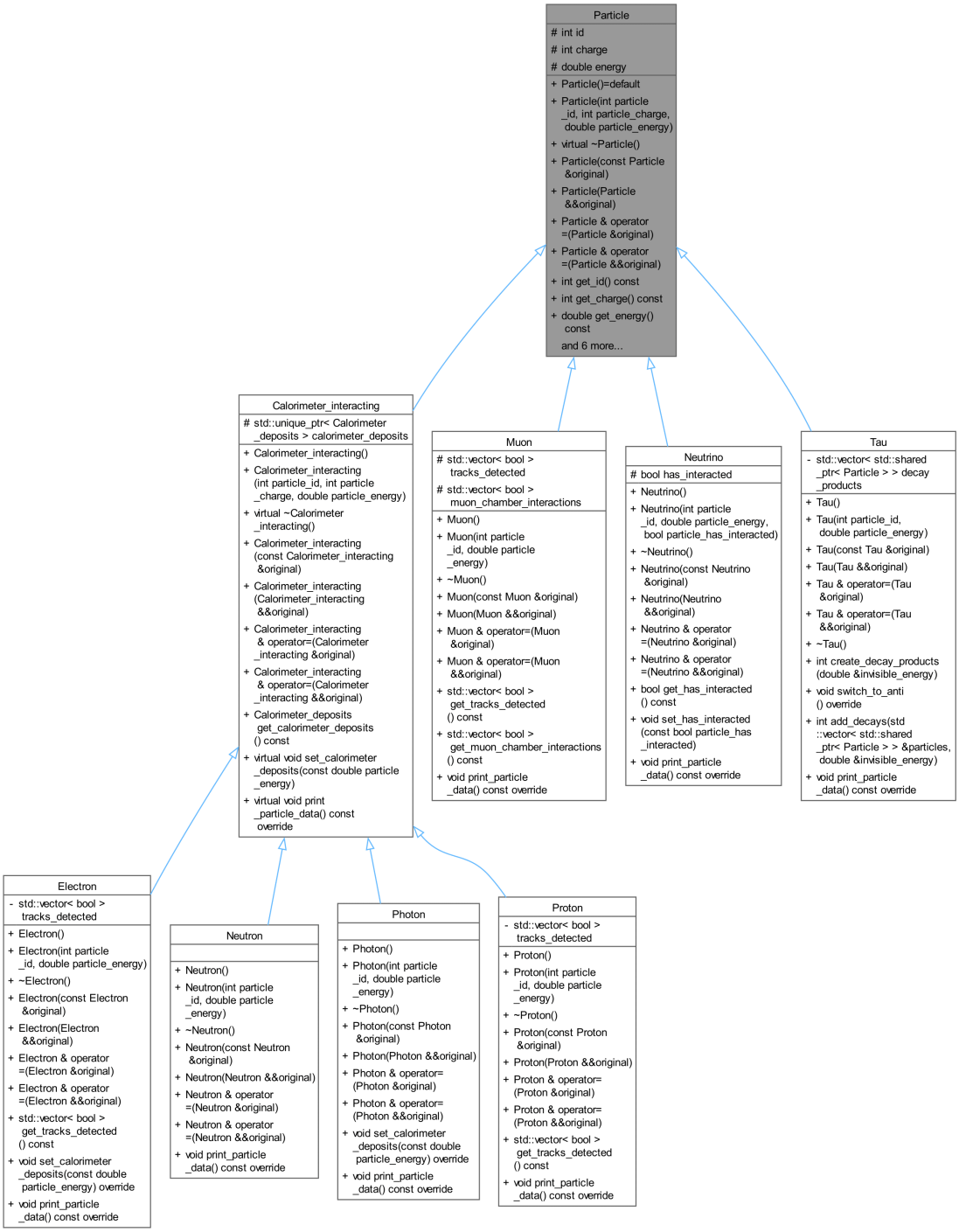

Figure 1: Image displaying the full inheritance chain for the particles in a UML format. Greyed out class indicates that it is abstract. This image is generated using Doxygen, which reads in the entire code and generates graphs based on it [5].

Line 6 uses the random function mentioned in Section 2.1, so that the energy is deposited randomly in each section of the sub detector. The inheriting classes electron and photon override `set_calorimeter_deposits` to ensure their hadronic deposits are zero.
As shown in Table 1, the muon and neutrino do not inherit from the particle class. Neither does the tau, as it only interacts via its decay products. The muon class contains two vectors of booleans: one for its tracker interactions; the other for the muon chamber interactions. These are set to true in the constructors. The neutrino class has no variables with its interactions in the detectors, as there are none. It only has a boolean for whether it has interacted or not.
The tau class has one member variable, containing pointers to its decay products. These are set in the `create_decay_products` function, a snippet of which is shown in Listing 3.

```
1 if (decay_type_rand > 0.65) // Tau decays leptonically
2   {
3     // Three decay products in leptonic decay so need three random
          numbers that add to one
4     std::vector<double> random_energies{random::gen_n_numbers(3)};
5     decay_products.push_back(std::make_shared<Neutrino>(Neutrino(id,
          energy * random_energies[0],  false)));
6     invisible_energy += energy * random_energies[0];
7     // By lepton universatliy, muons and electrons should have equal
          probability of being generated here
8     double leptonic_decay_rand{random::gen_rand_0_and_1()};
9     if (leptonic_decay_rand > 0.5)
10    {
11      decay_products.push_back(std::make_shared<Neutrino>(Neutrino(id
            + 1, energy * random_energies[1],  false)));
12      decay_products.push_back(std::make_shared<Electron>(Electron(id
            + 2, energy * random_energies[2])));
13      invisible_energy += energy * random_energies[1];
14    }
```

Listing 3: A snippet of the code for the `create_decay_products` function, which is part of the `tau` class. Function simulates the decay of a tau, and adds the decay products to the `decay_products` vector.

We simulate the random behaviour of tau decays, as outlined in Section 1, using RNG. This snippet outlines the leptonic decay channel. In lines 1 and 9, we chose the decay type randomly, and in line 4, we create a vector of random numbers to random divide the energy amongst the decay products. In lines: 4, 11, and 12, the decay products are added to the vector. The energy of the neutrinos is recorded in lines 6 and 13, such that the true invisible energy for an event, which is only due to neutrinos, can be displayed at the end of the event. For the hadronic decay, a particle of class `Calorimeter_interacting` is made, carrying the energy of the tau, as well as a tau neutrino.

2.3 Designing Detector Classes

The detector inheritance chain is set up in a simply fashion to the particle one. It is shown in Figure 2. The base detector class is abstract, adhering to polymorphism. It contains no member variables, as the methods of storing information in each sub detector are different. For instance, the tracker and muon chamber store their detected information in a map of integers, for the id, and doubles, for the energy. The calorimeter stores energy information for all sub layers, so a single double does not suffice, so a different type of map is used. However, a function `was_id_detected` is present in the base class, which determines if a particle is present in one of these maps. This is done simply using the `find()` function of `std::map`. This function is then overridden by the sub detectors.

As the tracker and muon chamber will either record all of the particle's energy, or none of it, only differing by the number of layers they have, it makes sense to group their functionalities. This is done in the class `Boolean_detector`. This stores a map of the detected ids and energy. A map is used here as it allows the detected energy to be paired to something we can identify the particle with. Hence, the energy for a specific particle can easily be extracted. The function `bool_detection_with_efficiencies`, encapsulates most of the detection of the tracker and muon chamber. A snippet is shown in Listing 4.

```
1  int layers_detected{0};
2  // Use rng to simulate the detector efficiency
3  double rand{random::gen_rand_0_and_1()};
4  for (int i{}; i < number_of_layers; i++)
5  {
6    if (rand < efficiencies[i])
7    {
8      detection_in_layers.push_back(true);
9      layers_detected++;
10   }
11 }
12 // if not detected in more than 1 layer then particle is not
       detected
13 if ((number_of_layers - layers_detected > 1) == false)
14 {
15   detected_ids_and_energy.insert(std::pair<int, double>(particle->
         get_id(), particle->get_energy()));
16   // Particle was detected
17   return true;
18 }
```

Listing 4: A snippet of the fucntion `bool_detection_with_efficiencies`, which is part of the `boolean_detector` class. Function simulates the detection of a particle in a generic boolean detector.

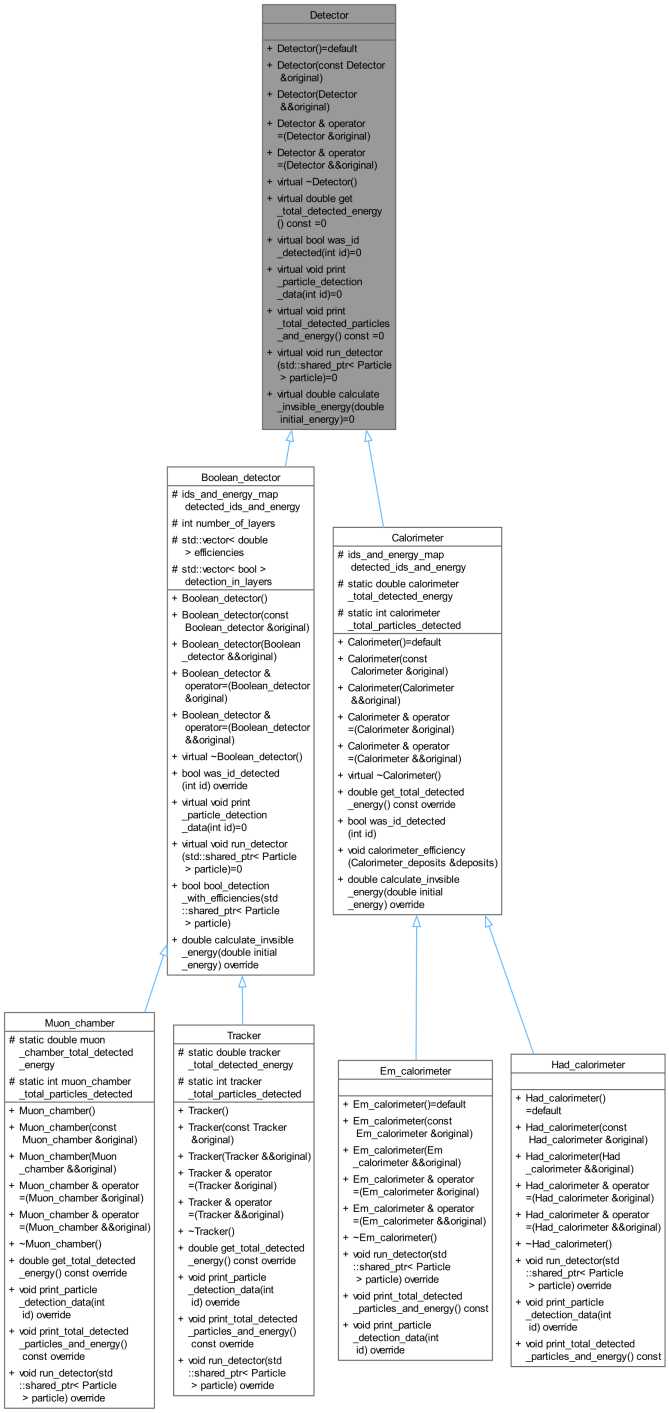

Figure 2: Image displaying the full inheritance chain for the detectors in a UML format. Greyed out class indicates that it is abstract. This image is generated using Doxygen, which reads in the entire code and generates graphs based on it [5].

Lines 4 to 10 focus on randomly simulating the efficiency of each layer, then recording that the particle was detected in this layer. Next, in line 13, we check if the particle was not detected in more than one layer. If so, the particle is declared not detected. If it is detected, the id and energy of the particle are inserted into the map via an `std::pair`, and the function returns true, indicating the particle was detected.
In the tracker, the number of layers is set to three. The tracker has static variables to record the energy and number of particles detected, such that totals of these values can be recorded between different events, when the class destructor will be called. Static variables like these are present in all the sub detectors, but cannot be defined in the base class as statics are not inherited, and thus we would only record a combined total of all the detector. The `run_detector` function is overloaded in the tracker. Here, we first check if the charge of the particle is zero, as the tracker only detects charged particles. Then, the `bool_detection_with_efficiencies` function is called to check if the particle is detected.
In the muon chamber, the number of layers is set to two. The only difference to the tracker is in `run_detector()`. The code used to check if a particle is detected in the muon chamber is shown in Listing 5.

```
1     // Attempt to cast particle ptr as a muon ptr
2     auto Muon_chamber_interacting_particle = std::dynamic_pointer_cast
          <Muon>(particle);
3     // If this fails, will give a nullptr, so throw these as they aren
          't muons
4     if (Muon_chamber_interacting_particle == nullptr) {
          detection_in_layers = {false, false};return;}
```

Listing 5: Snippet of `run_detector` overridden by the `muon` class. Code attmepts to cast generic particle pointer as a muon, thus determining if it is a muon.

In line 2, we utilise the function `std::dynamic_pointer_cast`, which allows us to recast a pointer to a class to one of a different class in the inheritance chain. If this is unsuccessful, a `nullptr` is returned. Hence, in line 4, we remove any particles that aren't muons. Then, the same steps as in the tracker are followed. The muon chamber also contains its own static variables for total detected particles and energy.
The calorimeter functions in a different way to both the tracker and muon chamber. We need to store the calorimeter deposits separately, as this allows us to distinguish the protons and neutrons, from electrons and photons which don't deposit in the HAD section. There is a function to simulate the efficiency of the calorimeter, `calorimeter_efficiency`. It uses `gen_decimal_in_range`, outlined in Section 2.1. We generate random numbers from 0.95 to 1, modelling the calorimeter as always detecting 95% of the energy. Static variables for the total number detected and energy are also present. The calorimeter is split into the electromagnetic and hadronic parts, thus separating the storage of what is detected in each part, allowing us to distinguish the electrons from protons et cetera. Both the EM calorimeter and HAD calorimeter override the `run_detector`

function, which is shown for the EM calorimeter in Listing 6.

```
1  auto calorimeter_interacting_particle = std::dynamic_pointer_cast<
       Calorimeter_interacting>(particle);
2  if (calorimeter_interacting_particle == nullptr) {return;}
3  Calorimeter_deposits particles_deposits =
       calorimeter_interacting_particle->get_calorimeter_deposits();
4  calorimeter_efficiency(particles_deposits);
5  std::vector<double> em_deposits{particles_deposits.get_em1(),
       particles_deposits.get_em2()};
6  // If all particles detected in calorimeter get detected in em part,
        so make em part record total detected
7  calorimeter_total_particles_detected++;
8  calorimeter_total_detected_energy += (em_deposits[0] + em_deposits
       [1]);
9  detected_ids_and_energy.insert(std::pair<int, std::vector<double>>(
       particle->get_id(), em_deposits));
10 ;}
```

Listing 6: Code shows how `em_calorimeter` overrides `run_detector`. Function checks if a particle is detected in the calorimeter by casting, then extracts the particle's deposits in the EM calorimeter, and stores them in a map.

In lines 1 and 2, we employ a similar method to the muon chamber, attempting to cast the particle as a calorimeter interacting one. In line 3, the particle's calorimeter deposits are extracted, and the EM parts are stored in a vector in the map. The reason the maps do not store an object of calorimeter deposits is such that we don't store two copies of the deposits in both the EM and HAD parts. These two copies would be different due to the random nature of the detector. The total number of particles detected in the calorimeter is added to here, as in the HAD part, not all particles get detected. The only major difference in the HAD calorimeter, is that we check if the hadronic deposits are 0, indicating the particle is an electron or photon.

3 Output and Use of Code

3.1 Running the Simulation and Printing Results

The simulation is run in a wrapper class named `Toy_simulation`, which is shown in Figure 3. It can run the simulation, which is done by iterating through the vector of detectors, nested inside a loop through the vector of particles. Next, the function `deduce_what_was_detected` can be called, which iterates through the vectors of detectors and particles, and calling `was_id_detected` for each detector. The information for where a given particle was detected is stored in a vector of booleans. Using Table 1, we narrow down what the particle is. The information for the particle detected in the event is then printed. The template function, `what_particle_is_this`, shown in

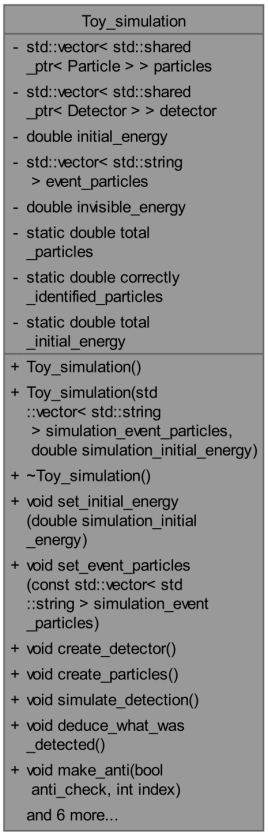

Figure 3: Image member variables and functions of `Toy_simulation` in a UML format. Greyed out class indicates that it is abstract. This image is generated using Doxygen, which reads in the entire code and generates graphs based on it [5].

Listing 7, is used to determine if the particle was correctly identified or not. This was done by the same method of casting to a generic particle type, shown in line 1.

```
1 template <class c_type> bool Toy_simulation::what_particle_is_this(std
     ::shared_ptr<Particle> particle)
2 {
3   auto real_particle_type = std::dynamic_pointer_cast<c_type>(particle
       );
4   // Will not give a nullptr if we can cast to the particle's type
5   if (real_particle_type != nullptr) {return true;}
6   return false;
7 }
```

Listing 7: Code for template function `what_particle_is_this`, a memeber function of `Toy_simulation`. Code recasts input particle pointer to a given particle class, allowing us to

determine if the input particle is that type of particle.

Static variables are used to store information on the total number of particles, the number detected, and the total energy, such that it totals up for all events. This information is printed after all events

3.2 Using the Code

The code is designed to be used in the terminal, with the user being able to input the names of particles, as well as the initial energy an the event. The user can then choose to input data for another event, or finish and display data for all events. Error checking on the particle names happens inside `Toy_simulation`, when the particles are created. This is to prevent code duplication. Physical checks on the initial energy happen inside the `Toy_simulation` constructor. The `input_checks` namespace contains functions to validate the other inputs made by the user. After an event, the true and detected information is displayed for each particle. For example, a single two photon event with initial energy 125 GeV produces the output shown in Figure 4.

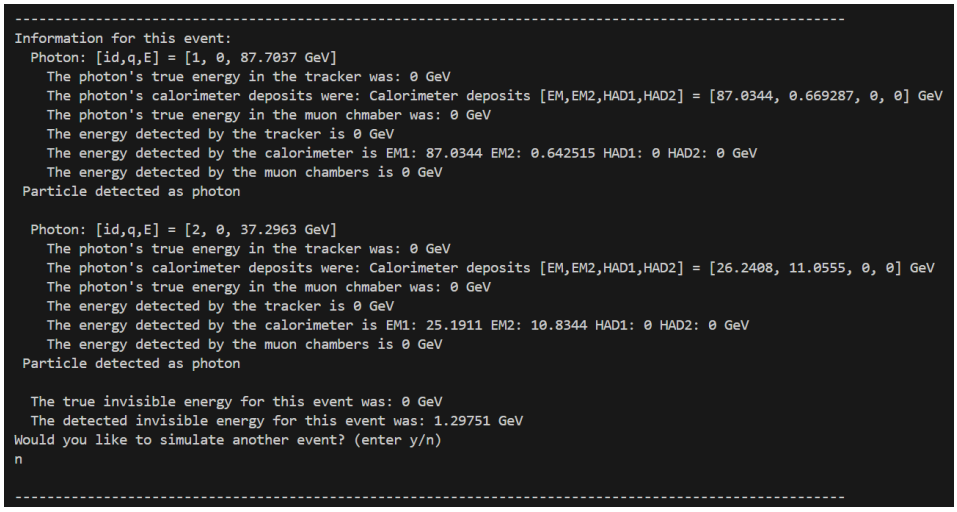

Figure 4:

4 Conclusions

One feasible extension could be to extend to detection efficiencies beyond a flat rate for all particles. In reality, detectors reconstruct particles at different efficiencies. This could have been done by storing efficiencies in a map, pairing the efficiencies to the name of particle. The particle's type could be found using by casting and hence its associated efficiencies could be extracted from the map.

Another useful extension would be to include the four momentum of the particles, which could be done with a class similar to the calorimeter deposits. General particle physics detectors mea-

sure the transverse momentum of particles, which is used to be able to infer the presence of neutrinos, and distinguish them from inefficiencies in the detector. If we knew what neutrinos were detected, the presence of a tau would also be able to be reconstructed.
In conclusion, the code functions as desired, simulating a simplified version of a particle detector. The code adheres to principles of object oriented programming, containing inheritance chains for particles and detectors. It simulates the detection of particles one at a time, using random number generators to simulate the detection efficiencies. Possible extensions include a four momentum object, allowing the transverse momentum to be detected. This would aid in the reconstruction of neutrinos and taus.

The number of words in this document is 2548.

References

[1] Aad, G. et al, "Measurement of W± and Z-boson production cross sections in pp collisions at s=13 TeV with the ATLAS detector," *Physics Letters B*, vol. 759, p. 601–621, 2016.

[2] B. Martin, *Particle physics*. Chichester, UK: John Wiley Sons, 2017.

[3] Tanabashi, M. et al, "Review of Particle Physics," *Phys. Rev. D*, vol. 98, p. 030001, Aug 2018.

[4] R. Lischner, *Exploring C++11*. New York City, US: Apress, 2013.

[5] https://doxygen.nl/, *Doxygen*. van Heesch, D., May 2024.