Particle Detector

PHYS30762 C++ Project

Zak Trivedi 10841354
Department of Physics and Astronomy, University of Manchester

May 8, 2025

ABSTRACT

This project aims to simulate the detection of particles from a collision event by a Compact Muon Solenoid (CMS) like detector. A class for a modifiable generic detector consists of sub-detectors classes, including trackers, muon chambers and calorimeters. Different particle classes are detected by this detector, which measures their energy (if detected) and infers the particle type. Detectors are not assumed to be ideal and their accuracy is dictated by detector efficiency and resolution. An experiment demonstrating the relationship between electromagnetic calorimeter efficiency and incorrect detections for lead tungstate and another material, using the simulation setup, is shown.

1. INTRODUCTION

Particle detection is essential for understanding fundamental interactions, validating theoretical models and discovering new particles [1]. Hence, efficient and robust software is necessary to compute the vast detector data rapidly [2]. This project was inspired by software for the Compact Muon Solenoid (CMS) at the Large Hadron Collider (LHC), which aims to investigate the interactions of high-energy particles [3].

The CMS contains many sub-detectors to detect various particles [4]. Such a design is implemented in this project, with a generic detector composing of different sub-detectors. The available sub-detectors chosen for this simulation were trackers, muon chambers and calorimeters. The muon chambers were split into two types based on whether they use drift tubes (DT) and cathode strip chambers (CSC) to detect particles, both of which detect only muons. Similarly, the calorimeters were split into electromagnetic calorimeters (ECAL) and hadron calorimeters (HCAL), which detect electromagnetic particles and hadrons respectively. The tracker sub-detector detects any charged particle.

Consequently, neutrinos are not directly detected by any sub-detector. However, their presence can be determined indirectly from the interaction's overall missing transverse energy (MTE), the imbalance of total momentum in the transverse plane before and after the collision [5]. Any significant MTE implies the presence of undetected neutrinos carrying away momentum and can be calculated using the transverse momenta of the detected particles after the collision (p_x and p_y) as

$$MTE = \sqrt{(\sum p_x)^2 + (\sum p_y)^2}. \tag{1}$$

The objective of this simulation is to utilise an object-oriented approach in C++ to model particle detection in realistic detectors. This includes taking into account detector resolution and efficiency, which can change as a result of different detector properties, to implement particle detection misses and an error in the measured value. It also aims to determine the particle type and MTE based on these detections.

2. CODE DESIGN AND IMPLEMENTATION

2.1. Introduction to Code Structure

The code consists of two hierarchical class structures for particles and sub-detectors, where classes and their implementation separated for abstraction. Additionally, there is a generic detector class, initiated in the main function within the project file that runs the simulation (`main()`), that sub-detector classes can be added to and removed from as desired. This allows for the user to create a custom detector. Furthermore, there is a four momentum class which is used as a member in the particle hierarchy. The user is also able to construct the particles that they want to simulate detections for in the main function and set values for their four momentum, rest mass, charge and flavour/type (where applicable).

Advanced C++ features, such as smart pointers, the STL, static data members, exceptions and lambda functions, are also used to enhance this project and add extra functionality.

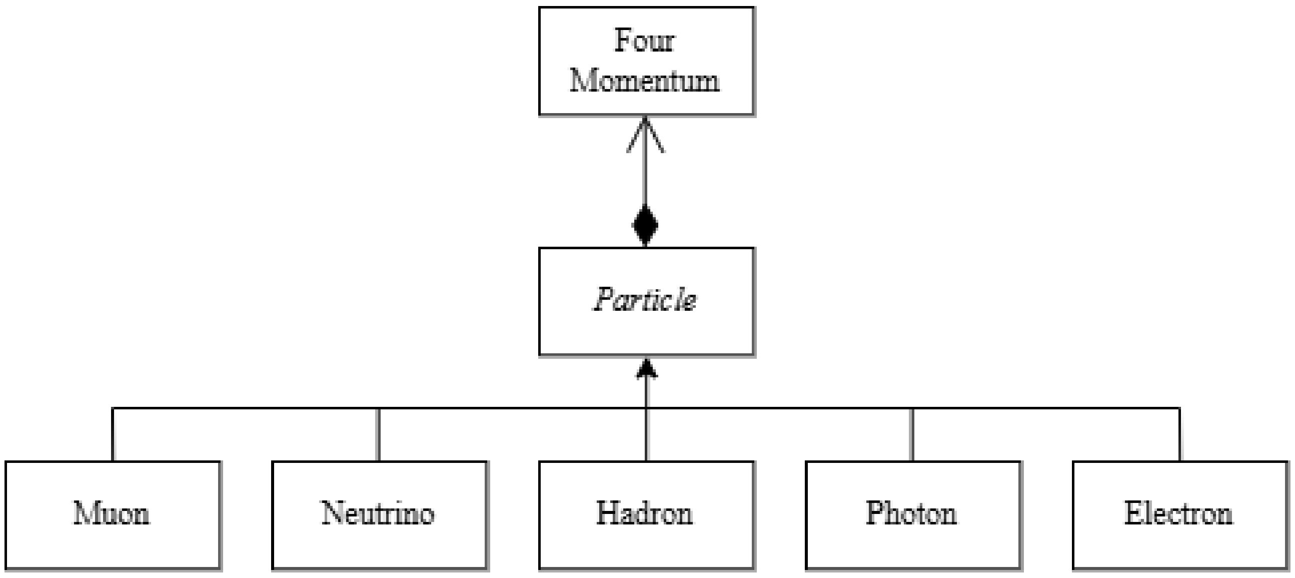

FIG. 1. UML diagram displaying the high level hierarchy for particles. Four momentum is used as a data member in the particle abstract base class.

2.2. *Class hierarchy for particles*

The class hierarchy for particles used in this project is simply displayed through the high level UML diagram in Figure 1. As shown, a particle abstract base class is used to provide an interface to store the common four momentum, rest mass and charge data members. The four momentum has its own class for further abstraction and to simplify the particle interface. It also includes a pure virtual print function (`virtual void print_data() const=0`), which is subsequently overridden in the derived classes. This enforces polymorphism and allows specialised print functions to be used for each particle type. The derived classes for neutrinos and hadrons also add an extra data member for flavour and type respectively as these are specific for their derived classes.

2.3. *Class Hierarchy for Detectors*

A high level UML diagram of the class hierarchy structure for detectors is depicted in Figure 2. For similar reason as for particles, it was chosen to have a sub-detector abstract base class as an interface for common class features. This contains data members for its status (ON/OFF), name and detection efficiency. Alike to the particle abstract base class, this class also includes a pure virtual print function, which is overridden in the subsequent derived classes. There is also a pure virtual detection function of `bool detect_particle(const Particle &particle)` in the abstract base class, which is also overridden in each derived class and simulates a detection event specific for each sub-detector. Each derived class (not abstract) has its own static data member for the detection count (`static int s_detection_count`) so that the count is kept for all of each detector if multiple are initiated.
The tracker class is derived from `Sub_Detector` and includes the extra data members of `std::stringm_material` and `int m_layers` for the material and layers of the tracker respectively. Its overriding `detect_particle` function utilises these factors to compute a resolution and an adjusted efficiency for the tracker. The adjusted efficiency logic due to tracker material is shown in Listing 1.

Listing 1. Snippet of the `detect_particle` function for the tracker sub-detector, showing how the material property is used to simulate detector efficiency and missed particles.

```
// Apply material-dependent efficiency modifier.
double material_factor = 1.0;
if(lowercase_material == "silicon")
{
  material_factor = 1.0;
}
```

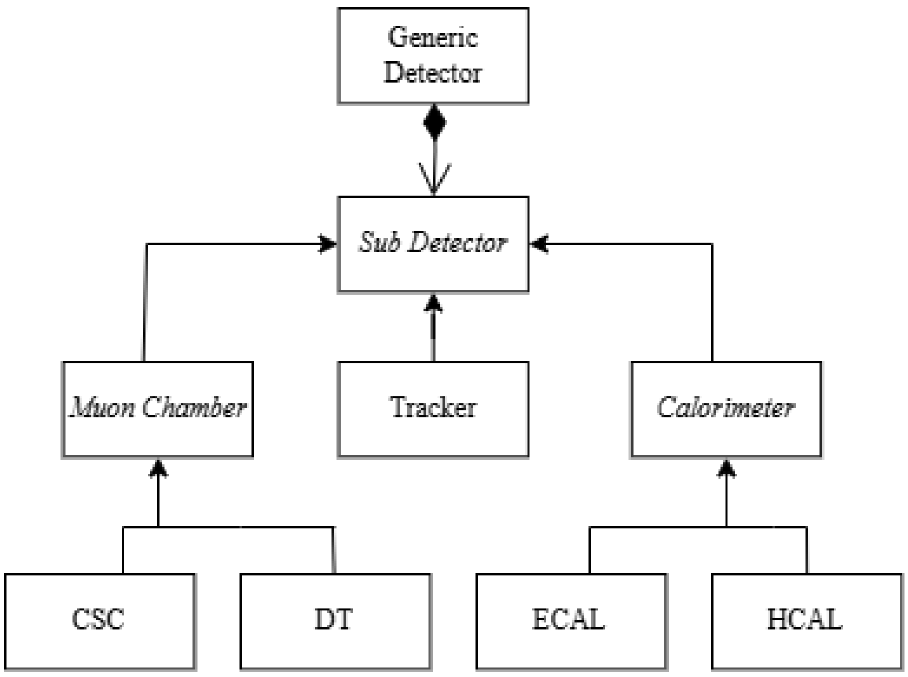

FIG. 2. UML diagram displaying the high level hierarchy for detectors. The generic detector is included as sub-detectors are combined in the generic detector.

```
else
{
  // Assume other materials are slightly less efficient
  material_factor = 0.8;
}
double effective_efficiency = m_detection_efficiency * material_factor;

// Simulate detection inefficiency.
static std::mt19937 random(std::random_device{}());
std::uniform_real_distribution<double> uniform_distribution(0.0,1.0);
if(uniform_distribution(random) > effective_efficiency)
{
  std::cout<<"Particle missed by tracker due to detector inefficiency."
     <<std::endl;
  return false;
}
```

As shown in Listing 1, the material and detector efficiency are combined to give an effective detector efficiency, which then uses a random number generator in a normal distribution from the STL to simulate missed detections due to this. This was implemented to mimic a realistic detector where particles are occasionally missed due to its material dependent efficiency.
The logic to compute tracker resolution, also contained in the `detect_particle` function, is shown in Listing 2.

Listing 2. Snippet of the `detect_particle` function for the tracker sub-detector, showing how the number of layers is used to simulate an error on the measured energy.

```
// Smear the energy assuming tracker resolution is 1/sqrt(layers)
double resolution = 1.0 / std::sqrt(static_cast<double>(m_layers));
std::normal_distribution<double> smear_distribution(
  true_energy, true_energy * resolution
);
double measured_energy = smear_distribution(random);
```

```
if(measured_energy < 0.0)
{
  measured_energy = 0.0;
}
```

Listing 2 illustrates how the resolution is computed from the inverted square root of the number of layers and how this is then used with a normal distribution to smear the measured energy away from the true energy. Such a design choice was made to reflect how a more layered detector would decrease the error in the measure energy.
The other two inherited classes of the `Sub_Detector` are two intermediate abstract classes for calorimeters and muon chambers. This intermediate abstraction was chosen to simplify their derived classes by providing an extra interface of common logic and members by applying good object-oriented programming design with inheritance.
The calorimeter intermediate abstract class store data members for material (`std::stringm_material`) and number of cells (`int m_number_of_cells`) as these are common for both types of calorimeter. The material property is used in the derived classes' detection functions in the same way as the tracker's, simulating missed particles wit an `effective_efficiency` as shown in Listing 1. The class also contains the pure virtual print and detect functions present in the base abstract sub-detector class so that these can be overridden in the derived classes.
`Calorimeter` has two derived classes `ECAL` and `HCAL`, representing electromagnetic and hadron calorimeters respectively. As well as its static detection count member, `ECAL` also includes a data member for resolution (`m_resolution`) as this is the dominant limitation of energy measurement for ECALs due to its operating procedure. This is combined with `int m_number_of_cells` to create an error in the measured energy in a similar way as in Listing 2 but with the slightly altered effective resolution formula shown in Listing 3.

Listing 3. Snippet of the `detect_particle` function for the ECAL sub-detector, showing how the ECAL's resolution is determined.

```
double effective_resolution = m_resolution/std::sqrt(static_cast<double
   >(m_number_of_cells))
```

This change was made to allow the intrinsic resolution of the individual detector to be scaled by its `int m_number_of_cells`.
On the other hand, `HCAL` provides an additional sampling fraction data member (`doublem_sampling_fraction`). This is used instead of resolution as for HCALs the sampling fraction is the limiting factor for energy measurement, because the active layers in the HCAL only see the visible fraction of the true shower energy. In the HCAL detection function, this is achieved by adjusting the resolution and true energy by this fraction to give the visible energy, as shown in Listing 4.

Listing 4. Snippet of the `detect_particle` function for the HCAL sub-detector, showing how the HCAL's measured energy is determined using the sampling fraction.

```
// Calculates visible energy based on sampling fraction
double visible_energy = true_energy*m_sampling_fraction;

// Resolution of 1/sqrt(Ncells)
double effective_resolution = 1.0/std::sqrt(static_cast<double>(
   m_number_of_cells));
std::normal_distribution<double> smear_distribution(visible_energy,
   visible_energy*effective_resolution);
double measured_energy = smear_distribution(random);
```

Alternatively, the muon chamber intermediate abstract class stores an additional common data member for component count (`int m_component_count`. Alongside carrying through the pure virtual print and detect functions, it also provides a protected `bool detect_common(const Particle &particle) const` helper function to be used by the detect functions in the derived classes. This was implemented to minimise code repetition. `Muon_Chamber` has two derived

classes for (CSC) and (DT) respectively to differentiate between the different types of muon chamber using this additional layer of inheritance. Their detection functions are similar to that of the tracker described in Listings 1 and 2 but using `int m_component_count` instead of `int m_layers` to give a value for the resolution.
In `main()` a generic detector is constructed and derived sub-detector classes are added to and removed from it as desired by the user. The vector of `std::unique_pointers` to particles described in sub-section 2.2 was then run through this generic detector and particle type and MTE were inferred from the results.

2.4. Additional Functionality and Advanced C++ Features

Additional functionality was included beyond the brief to further refine this simulation. Firstly, detectors were made more realistic by including detector efficiencies and resolutions so that data regarding how material properties and detector composition affect results could be collected. This was achieved by exploiting the STL library.
Additionally, it was chosen to infer the particle type after attempting detections from all detectors, to provide significance to the detection results. This was done in the main file using a static helper `static std::string infer_particle` function outside of `main()`. This function took arguments of the boolean outputs of each detection function (eg. `bool tracker_detection`) and deduced the particle type based on which detection functions had returned `true`. Another static helper function was also implemented to get the true particle type so that the detected type could be verified, to check for incorrect results due to detector inefficiencies.
Furthermore, Equation 1 is utilised in `main()` to check for the presence of neutrinos from the MTE. This was implemented to collect all possible data from the collision, allowing researchers to form accurate conclusions.
It was also decided to include `<algorithm>` from the STL library for use in the getter for sub-detectors within the `Generic_Detector` class to allow easy access to contained sub-detectors. Consequently, the generic detector can be easily modified as demonstrated in `main()`.
Other advanced C++ features were also included throughout the project. For instance, the use of static data for detection counts so that the count is recorded over multiple of the same class of detector. Unique pointers were chosen wherever a pointer was required to optimise memory management by keeping the heap as free as possible as they are automatically freed when they go out of scope. Also, the project used lambda functions to reduce code length and improve readability. Exceptions were also used in setter functions to ensure valid inputs. For example, the validation in the tracker layer setter is shown in Listing 5.

Listing 5. Example of a use of an exception in the setter function for the layers in a tracker.

```
void Tracker::set_layers(int layers)
{
  if(layers >=1)
  {
    m_layers = layers;
  }
  else
  {
    throw std::invalid_argument("Tracker layers must be >= 1");
  }
}
```

3. RESULTS

3.1. How to use

To use the code in `main()`, the user starts by constructing a generic detector and adding unique pointers to sub-detectors to it as desired using the `voidadd_sub_detector(std::unique_ptr<Sub_Detector>sub_detector)` function. This is shown in Listing 6.

Listing 6. Snippet of the `main()` showing how the user can add sub-detectors to the generic detector.

```
// Add sub-detectors
cms.add_sub_detector(std::make_unique<Tracker>("Tracker","Silicon"
    ,5,0.95,true));
cms.add_sub_detector(std::make_unique<ECAL>("ECAL","Lead Tungstate"
    ,75000,0.02,0.90,true));
cms.add_sub_detector(std::make_unique<HCAL>("HCAL","Steel Scintillator"
    ,40000,0.10,0.85,true));
cms.add_sub_detector(std::make_unique<Muon_Chamber_DT>("DT Chambers"
    ,250,0.9,true));
cms.add_sub_detector(std::make_unique<Muon_Chamber_CSC>("CSC Chambers"
    ,540,0.85,true));
cms.add_sub_detector(std::make_unique<HCAL>("HCAL extra","Steel
    scintillator",3000,0.15,0.85,true));
cms.add_sub_detector(std::make_unique<Muon_Chamber_CSC>("CSC Chambers
    extra",1040,0.9,true));
```

Here, the specifications for each sub-detector can be set, which are specific to each sub-detector class. These specifications (such as status, material, number of layers etc.) are then validated by exceptions similar to the one given in Listing 5. Afterwards, a vector of `std::unique_pointers` to particles is declared in `main()` that desired particles can be appended to using `emplace_back`. Similarly, features like the four momentum, flavour/type, rest mass and charge can be set here and these are also verified with exceptions in the setters. This is shown in Listing 7.

Listing 7. Snippet of the `main()` showing how the user can add particles to a vector which will be passed through the generic detector.

```
// Create sample particles
std::vector<std::unique_ptr<Particle>> particles;
particles.emplace_back(std::make_unique<Electron>(Four_Momentum(50.0,
    10.0, 15.0, 5.0)));
particles.emplace_back(std::make_unique<Photon>(Four_Momentum(60.0,
    -10.0, 5.0, 2.0)));
particles.emplace_back(std::make_unique<Muon>(Four_Momentum(80.0, 0.0,
    -20.0, 1.0)));
particles.emplace_back(std::make_unique<Hadron>(Four_Momentum(100.0,
    5.0, 0.0, -3.0),"proton", 938, 1.0));
particles.emplace_back(std::make_unique<Neutrino>(Four_Momentum(30.0,
    -5.0, 0.0, -4.0),"muon"));
```

Subsequently, the user can easily turn detectors ON or OFF or remove them from the configurations using `turn_off`, `turn_on` and `remove_sub_detector` functions respectively. The code can then be compiled and run to simulate the detection of every particle through every sub-detector in the generic detector and output the result of these detections, the inferred particles and the MTE. The output from a photon passing through a generic detector comprised of one of each sub-detector is show in Listing 8.

Listing 8. Snippet of the `main()` showing how the user can add particles to a vector which will be passed through the generic detector.

```
--- New Particle ---
Photon: Energy=60.00 MeV, Px=-10.00 MeV/c, Py=5.00 MeV/c, Pz=2.00 MeV/c,
   Rest mass=0.00 MeV/c^2, Charge=0.00 e
No particle detected by tracker (particle is neutral).
Particle detected by ECAL!
  True Energy=60.00 MeV, Measured Energy=59.66 MeV
No hadron detected by HCAL.
```

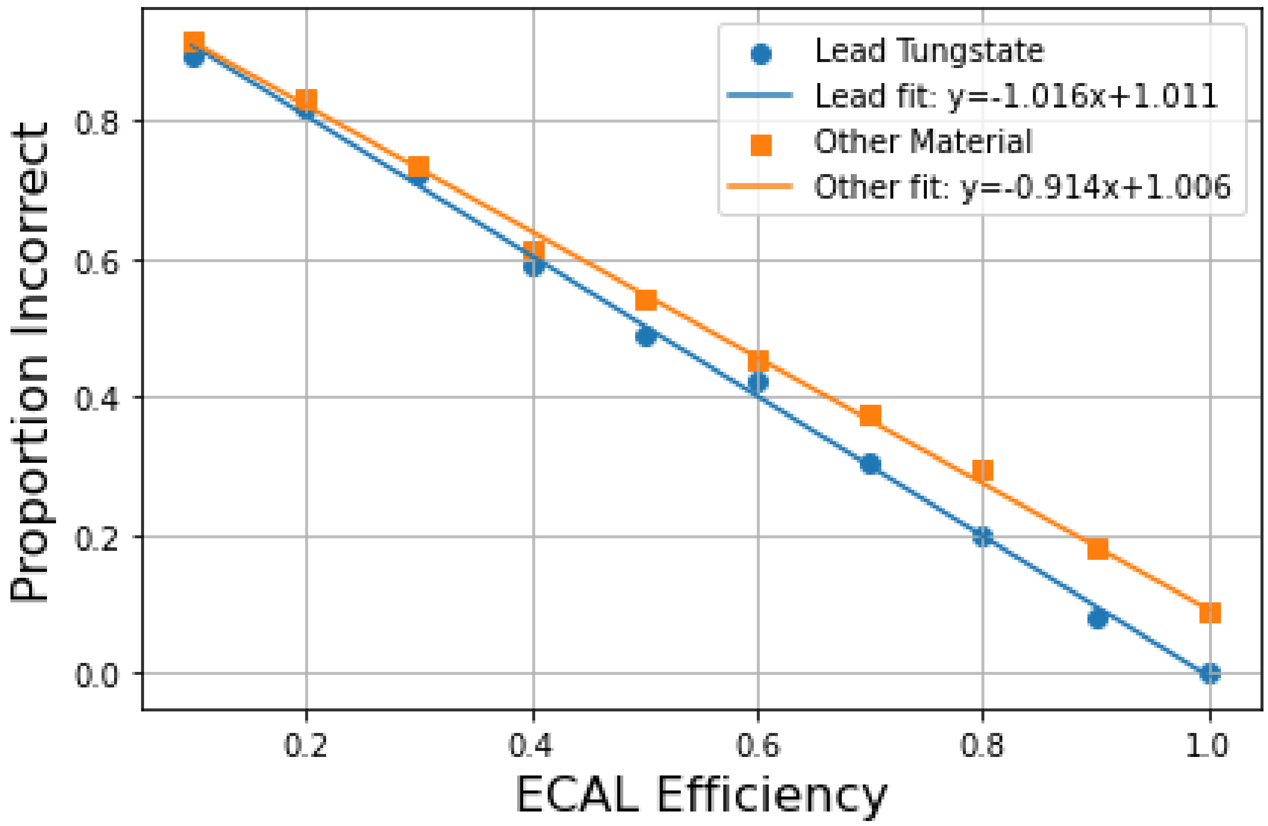

FIG. 3. Plot of ECAL efficiency against proportion of wrongly identified photons during a 1000 photon simulation for both a lead tungstate ECAL and one of an other material. Both data sets were fitted lineally with equations of $y = -1.016x + 1.011$ and $y = -0.914x + 1.006$ for lead tungstate and the other material respectively.

```
No muon detected by DT Chambers.
No muon detected by CSC Chambers.
Inferred particle type: Photon
Correctly detected Photon.
```

3.2. *Dependence of ECAL Efficiency on Correct Identification of Photons*

As an example of data that can be collected from this project, the simulation was run for 1000 identical photons through a generic detector comprised of all types of sub-detector at different efficiencies of the ECAL. A simple counter was implemented into `main()` as shown in Listing 9 to count the number of times a photon was wrongly detected due to the ECAL's inefficiency.

Listing 9. Adaption to `main` to count photons that were wrongly identified as something else during this example experiment.

```
 int wrong = 0;
 if(inferred != "Photon")
     ++wrong;
```

This was done for a lead tungstate ECAL and other material ECAL, allowing ECAL efficiency to be plotted against proportion of incorrect detections for both compositions as shown in Figure 3. This illustrates the advantage of using lead tungstate over other materials as it allows a lower detector efficiency to achieve the same level of successful detections. For instance, if a miss rate of 0.2 was considered acceptable, a lead tungstate detector would only require approximately a 0.8 ECAL efficiency, whereas another material would require approximately 0.9.
This simple analysis demonstrates the potential of this project to analyse how detector materials affect accuracy and can easily include other materials. Furthermore, this project could also be easily adjusted

for alternative tests, such as testing the dependency of measured energy error on the number of layers of the tracker. Combined with information regarding the cost per layer, this could be used to find the optimal number of layers for desired detection rate with minimal cost.

4. DISCUSSION

Use of detector parameters, such as material and layers, to determine detector efficiencies and resolutions allows for the simulation of realistic CMS like detectors. Additionally, this facilitates extraction of useful data from simulations, such as the dependencies of successful detections and measured energy error on the detector properties. An example of the simulation being used for this purpose was shown in Figure 3, proving how a more effective material places fewer requirements on detector efficiency.

However, there are also some key limitations to this project. Whilst consideration has been taken to make detectors realistic, some effects such as background noise are not accounted for. Also, features such as detector material and its associated factor to calculate the effective efficiency are hard coded into the program, making it difficult to add another material as an option with a different specific material factor. To improve the project further, all other realistic considerations should be systematically accounted for. Additionally, features such as materials should have their own classes so that future users can easily construct new materials with specific material factors for detectors.

Moreover, the structure of `main()` is somewhat monolithic, making it difficult to extend without causing errors. Classes for `Detector_Builder`, `Event_Simulator` and `Analyser` would further abstract `main()`, making it easier for future users to adapt.

Word Count From Overleaf: 2483

[1] D. Griffiths, *Introduction to elementary particles* (John Wiley & Sons, 2020).

[2] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, *et al.*, Computer Physics Communications **180**, 2499 (2009).

[3] C. Collaboration *et al.*, Journal of instrumentation **3**, 1 (2008).

[4] C. , Detector — cms experiment.

[5] S. Ellis, R. Kleiss, and W. Stirling, Physics Letters B **158**, 341 (1985).