Particle Detector Simulation in C++

PHYS30762 Project Report

Amrit Bath 1089910
Department of Physics and Astronomy
University of Manchester

May 21, 2025

Abstract

This project simulates a particle detector using object oriented design principles, and is built around a polymorphic class architecture, with an abstract `Particle` base class, concrete subclasses, and individual detector classes. The framework incorporates custom manager classes such as `DetectorManager` and `InferenceEngine` that provide methods for running batch detections, logging true versus measured energies, and inferring unseen neutrinos from missing energy. Specialised features include tau-decay branching logic, probabilistic detection efficiencies, and rule-based particle identification based on hit patterns.

1 Introduction

Modern particle physics experiments rely on sophisticated detector systems to reconstruct properties of particles produced in high-energy collisions. Among the most prominent are the ATLAS and CMS detectors at the Large Hadron Collider (LHC), designed to explore the Standard Model and search for new physics including the Higgs boson and dark matter. These detectors use layered subsystems including trackers, calorimeters, and muon chambers to capture a particle's charge, momentum, energy, and trajectory [1]. The ATLAS detector, located 100 metres underground near Meyrin, Switzerland, employs six concentric subsystems to identify and measure particles emerging from collisions by using a large magnet system to bend charged particles, ATLAS is the largest-volume particle detector constructed [2]. The CMS detector uses a superconducting solenoid generating a 4-tesla magnetic field, enabling precise momentum measurements. It shares physics goals with ATLAS but follows different engineering principles [3]. These detectors are designed to explore the Standard Model and to search for phenomena such as the Higgs boson, whose discovery was confirmed in 2012 [4].

Modern particle physics is described by the Standard Model, a quantum field theory that unifies the electromagnetic, weak, and strong interactions. It organises all known elementary particles into three generations of quarks (up, down; charm, strange; top, bottom), three generations of charged leptons (electron, muon, tau) with their corresponding neutrinos, and the gauge bosons (photon, $W^{\pm}$, Z^0, gluons) that mediate forces, additionally the Higgs boson responsible for mass generation. The Standard Model predicts

phenomena from nuclear decays to the properties of hadrons and electroweak processes. Its agreement with observations makes it the highly accepted in modern day particle physics and is the benchmark against which any new theoretical or experimental discovery must be compared [5].

2 Code Design and Implementation

This section outlines the architectural structure and simulation features of the particle detector system. This project simulates detector architectures through the framework, the `Tracker` class uses a multi-layer geometry and 90% detection efficiency. The software implements abstract base classes and derived components for different detector types. Features include probabilistic detection, energy smearing, and particle identification logic based on detector responses. The design of the code was developed with modularity, encapsulation, and extensibility in mind, with Table 1 summarising key assumptions and physical modelling features embedded in the simulation, including detector efficiencies, particle-detection, and energy-handling procedures. The framework models particle creation, propagation, energy deposition, and classification. It also accounts for practical constraints like incomplete energy capture and neutrino invisibility, requiring post-detection reasoning [6].

Figures 1 and 2 show the inheritance changes and runtime compositions, respectively. The codebase was structured with attention paid to ensuring each module cleanly maps to physical phenomena or experimental procedures observed in particle physics. Each particle inherits from a `Particle` base class, enforcing accessors for energy, mass, type, and four-momentum. Detectors are modular and encapsulated. Smart pointers manage memory cleanly, avoiding leaks in branching logic like tau decays. Event energies and scattering angles are generated using STL random distributions to reflect real-world variability. Custom exceptions enforce constraints such as limiting velocity below the speed of light.

The remainder of this report discusses design and implementation, with results illustrated through sample outputs, followed by reflections on how the model could be extended to simulate more complex experimental setups.

Table 1: Summary of Simulation Assumptions

Category	**Assumption / Model Description**
Tracker	Detects only charged leptons: electrons, muons, and their antiparticles. Detection occurs probabilistically using a fixed efficiency of 90%. Material is silicon with 5 layers.
Calorimeter	Detects all particles depositing energy (electrons, photons, hadrons). Smeared energy measurement based on Gaussian distribution. Material: Lead tungstate crystals.
Muon Chambers	Detects only muons and anti-muons. Energy recorded if matched and within efficiency.
Particle Mass and Charge	Each subclass is initialized with rest mass, energy, charge, and optional antiparticle status. Photon has zero mass and charge; neutrino is neutral with no detector response.
Four-Momentum Handling	All particles are assigned a `FourMomentum` object which stores (E, p_x, p_y, p_z). Used by detectors to retrieve true energy values.
Detection Efficiency	Applies a probabilistic check using the STL random number generator. Simulates realistic inefficiencies.
Particle Identification	Infers particle type based on the combination of detector hits. Tracker + calorimeter but not muon chamber : likely electron. Calorimeter-only hits : photons.
Neutrinos	Not detected by any sub-detector. Missing energy inferred from the imbalance of detected energy.
Antiparticles	Treated identically to matter counterparts in detection logic. Distinguished by a boolean flag internally.

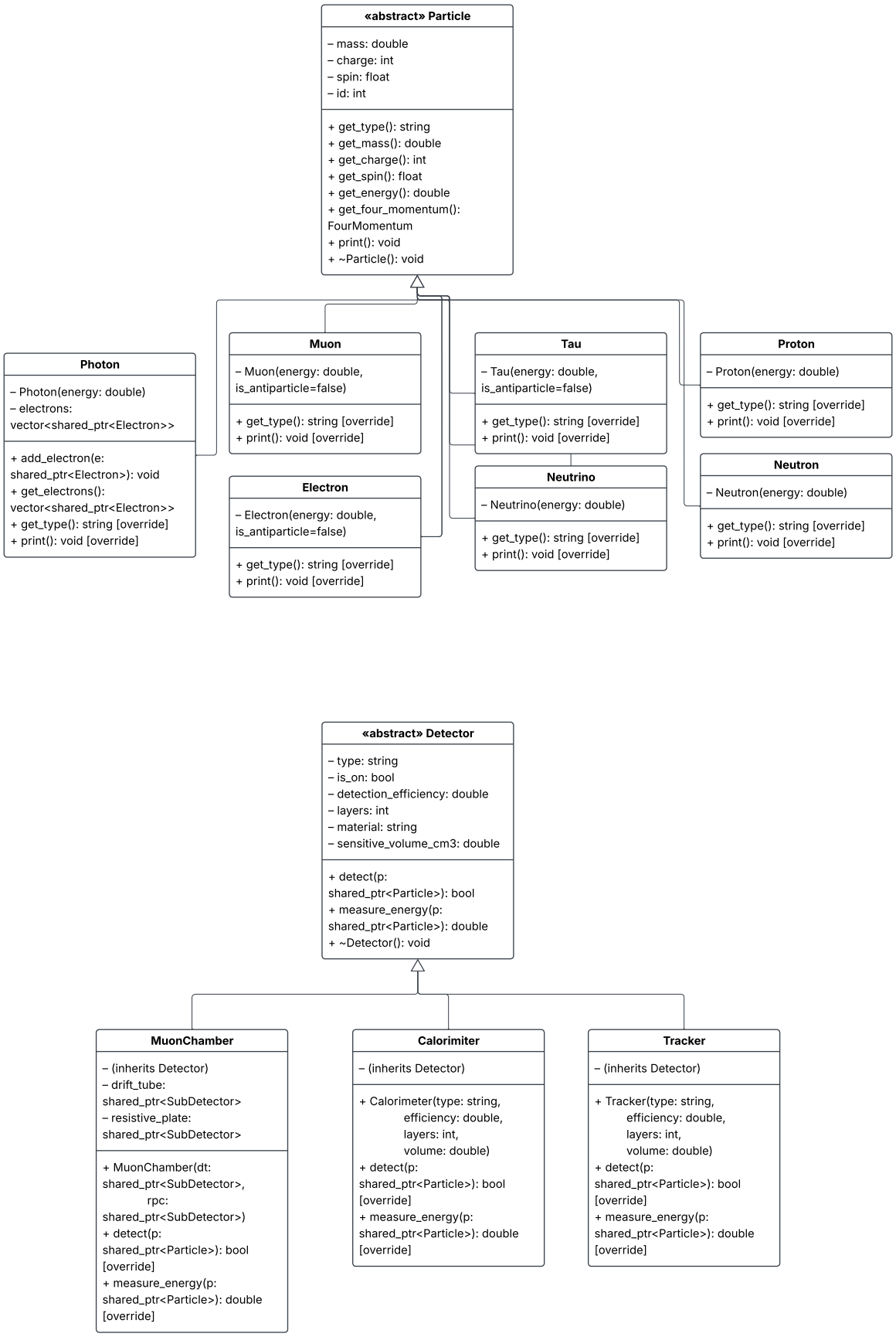

Figure 1: UML diagram showing the inheritance chain of classes, including particle species and detectors. Made using Lucidchart [7]. Solid arrows with unfilled heads denote inheritance relationships, showing a class derives from an abstract parent.

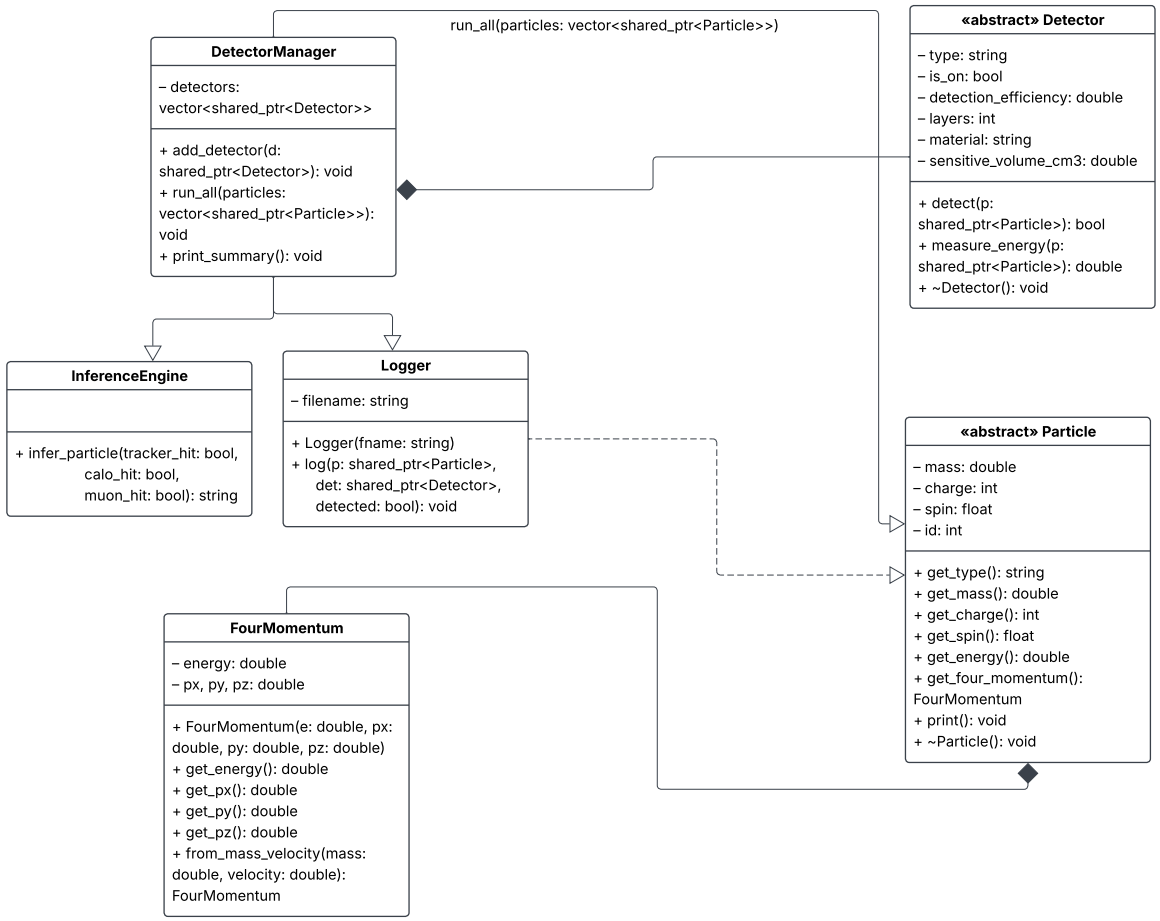

Figure 2: UML diagram showing the runtime composition and associations classes. Made using Lucidchart [7]. Solid arrows with unfilled heads denote inheritance relationships, while solid lines with diamonds represent composition or aggregation, where one class contains or references another.

The framework integrates polymorphism, inheritance, smart pointers, STL random distributions, and file input/output. These are used throughout to enhance realism and support clean memory management such as tau decay branching, Compton scattering angles and probabilistic detection [8].

To prevent multiple inclusion of header files, I used `#pragma once` rather than the traditional `#ifndef`, `#define`, and `#endif` include guards. Although `#pragma once` is technically not part of the ISO standard, it is widely supported by major compilers including GCC, Clang, and MSVC. Following core guidelines, this improves code clarity, reduces the risk of errors from manual macro definitions, and can slightly enhance compilation efficiency [9]. Given that the project requirements specify Cpp 17 compatibility with Visual Studio Code, and all target compilers support `#pragma once`, it is a suitable alternative. [10]. Lambda functions are used to simplify operations over STL containers, such as particle generation and detector iteration, to improve compactness and readability.

2.1 Particle Architectures

The `Particle` class enforces a common interface across all particle types. It defines virtual methods such as `get_type()`, `get_energy()`, and `print_data()`, ensuring each derived type - `Electron`, `Muon`, `Photon`, `Neutrino`, etc. - implements its own behaviour while maintaining consistency with detectors. Each particle also stores intrinsic properties such as charge, four-momentum, and spin: a fundamental quantum number used to distinguish fermions and bosons. Spin is initialised in derived classes (e.g., 0.5 for electrons, 1.0 for photons) and displayed via the overridden `print_data()` method. Particles are dynamically allocated and stored via `std::shared_ptr<Particle>`, to enable runtime polymorphism and reduce the risk of memory leaks in branching processes like tau decays. This choice is especially beneficial for handling decay trees, where particles may recursively generate new child particles.

```
class Particle {
protected:
    double rest_mass;
    double energy;
public:
    Particle(double rm, double e);
    virtual ~Particle() = default;
    virtual std::string get_type() const = 0;
    virtual void print_data() const = 0;
};
```

Listing 1: Simplified Particle base class

Each particle contains an instance of the `FourMomentum` class, which encapsulates energy and three-momentum components for relativistic calculations [11].

```
four_momentum = FourMomentum::from_mass_and_velocity(rest_mass, 0.0, 0.0,
    velocity);
```

Listing 2: Four-momentum usage in particle

2.2 Detector Hierarchy

All detectors inherit from a common `Detector` base class. Derived classes include `Tracker`, `Calorimeter`, and `MuonChamber`, each overriding the `detect()` function. For example, the `Tracker` only responds to charged leptons, while the `Calorimeter` absorbs energy stochastically based on a Gaussian smear. The `MuonChamber` identifies muons and simulates a layered architecture composed of drift tubes (DT) and resistive plate chambers (RPC), each with separate detection probabilities. This reflects the structure of real LHC muon subsystems.

```
if (particle->get_charge() != 0 && is_lepton(particle)) {
    return rng() < detection_efficiency;
}
```

Listing 3: Detection logic in Tracker

```
bool Calorimeter::detect(std::shared_ptr<Particle> p) {
    double original_energy = p->get_energy();
    double absorbed_fraction = distribution(gen);
    p->set_energy(original_energy * (1.0 - absorbed_fraction));
```

```
    return absorbed_fraction > 0.2;
}
```

Listing 4: Calorimeter absorption logic

Each detector is initialised with metadata like material, number of layers, and intrinsic efficiency. In `MuonChamber`, the overall detection probability combines DT and RPC efficiencies. The `DetectorManager` acts as a container for detector components. It provides batch operations including adding detectors, toggling detector states, running detection across all particles, and reporting measured energies.

```
class DetectorManager {
private:
    std::vector<std::shared_ptr<Detector>> detectors;
public:
    void add_detector(const std::shared_ptr<Detector>& detector);
    void detect_all(const std::shared_ptr<Particle>& particle);
    void print_all() const;
    void turn_all_on();
    void turn_all_off();
};
```

Listing 5: DetectorManager header

2.3 Branching Physics

Photons in the simulation may undergo Compton scattering, the photoelectric effect, or pair production, depending on their energy. Although currently effects are cycled deterministically to demonstrate system capabilities, full probabilistic selection could be added.

```
if (photon_energy > 1.022) {
    spawn_electron_pair(photon_energy);
}
```

Listing 6: Pair production with photon energy threshold

Tau leptons can decay via leptonic or hadronic paths. The `TauDecay` module selects decay outcomes probabilistically and returns multiple new particles [12].

```
if (r < 0.5) {
    auto mu = std::make_shared<Muon>(100.0);
    auto nu = std::make_shared<Neutrino>(50.0);
    products.push_back(mu);
    products.push_back(nu);
} else {
    auto p = std::make_shared<Proton>(500.0);
    auto n = std::make_shared<Neutron>(300.0);
    auto nu = std::make_shared<Neutrino>(100.0);
    products.push_back(p);
    products.push_back(n);
    products.push_back(nu);
}
```

Listing 7: Tau Decay Possibilities

2.4 Simulation Pipeline

Post-detection, particle identity is inferred from detector-hit patterns. Neutrinos, invisible to all detectors, contribute to missing energy calculations.

```
if (tracker && calorimeter && !muon) return "electron";
if (!tracker && !calorimeter && !muon) return "neutrino";
```

Listing 8: Rule-based inference logic

The `Logger` records true/measured energies, detector responses, and inferences to CSV files for analysis. The `main()` function follows a modular sequence: particle creation, detector setup, tau decays, photon interactions, detection, inference, and logging. Random values like energies and scattering angles are generated using STL distributions, seeded by `std::random_device`. Particles can also be initialised from an external `Particles.txt` file for reproducibility.

```
initialise_particles(particles, gen, energy_dist);
setup_detectors(tracker, calorimeter, muon_chamber, detector_manager);

simulate_tau_decay(particles, logger);
simulate_photon_interactions(std::dynamic_pointer_cast<Photon>(particles
    [2]), particles);
simulate_extra_photon_effects(particles, gen);

run_simulation(particles, tracker, calorimeter, muon_chamber, logger,
    total_energy, visible_energy);
```

Listing 9: Simulation pipeline in main.cpp

The output includes true vs measured energies, particle inference, missing energy estimates, detector counts, and calorimeter error percentages. The system is fully extensible for adding detectors or interaction types.

3 Results

The simulation output reflects a realistic sequence of particle creation, decay, interactions, and detection across multiple subsystems. The results are displayed in a structured terminal output, with optional CSV logging for post-run analysis. The output sequence is summarised below, showcasing the system's end-to-end functionality. At the beginning of the simulation, a set of particles is initialised with energies from a uniform distribution between 400-1600 keV. Their properties, including rest mass, charge, velocity, and four-momentum, are printed to the terminal. Derived quantities such as relativistic β and γ are calculated using the `FourMomentum` class.

```
Particle Type: electron
Mass: 0.511 MeV
Charge: -1e
Velocity: 0 m/s
Beta: 0
Gamma: 1.26e+03
Four-Momentum: (E = 0.511, px = 0, py = 0, pz = 0)
```

Listing 10: Particle Creation

Additional particles including neutrinos, protons, neutrons, and tau leptons are added to expand interactions. The `TauDecay` module probabilistically selects a decay path for

the tau lepton. In the example shown, a leptonic decay is selected, resulting in a muon and a neutrino. These products are printed and logged, and the tau is destructed. Smart pointers ensure correct memory handling.

```
Decaying tau...
Leptonic decay -> muon + neutrino
Decay products:
Particle Type: muon
Particle Type: neutrino
```

Listing 11: Leptonic Tau Decay

Photon interactions are stochastically triggered, with high-energy photons producing electron pairs via pair production. Additional photons undergo either Compton scattering or the photoelectric effect. The system prints interaction details, including energy deposition and angle of scattering.

```
[Effect] Applying photoelectric effect...
  Energy deposited: 1.51e+03 keV
[Effect] Applying Compton scattering...
  Scattered angle: 36.2 degrees
```

Listing 12: Photon Effects

Each particle is passed through the three detector components (tracker, calorimeter, muon chamber), where probabilistic detection is applied. Measurement values per detector are printed. Using the pattern of hits, the `InferenceEngine` proposes a most-likely particle classification.

```
electron detected in tracker.
electron deposited 386 keV in calorimeter.
electron not detected in muon chamber.
Inference: electron (for particle type: electron)
```

Listing 13: Electron Detection

Notably, neutrinos and neutral particles (like photons below detection threshold) are correctly inferred as "neutrino or neutral" based on zero hits. At the end of the run, the total true energy, measured energy, and inferred missing energy are calculated. This mimics real detector workflows where non-interacting particles (especially neutrinos) lead to energy imbalance.

```
Total event energy:      5e+08 keV
Visible detected energy: 2.6e+08 keV
Missing energy:          2.4e+08 keV
```

Listing 14: Event energies

The calorimeter also prints a comparison of true deposited energy versus measured output, with an error estimate:

```
Calorimeter true deposited energy: 1.5e+03 keV
Calorimeter measured energy:       1.63e+03 keV
Calorimeter error (%):             8.82 %
```

Listing 15: Calorimiter Energies

This aligns with realistic measurement uncertainties due to energy smearing and statistical variation. A final summary of detection counts and subsystem configurations is printed. This confirms that the polymorphic behaviour of the detector hierarchy has

functioned correctly, and that detector efficiency values have been continued throughout the simulation. Additionally, as seen below, the tracker's 90% efficiency leads to four charged-lepton hits in the sample run which is followed through from the previous discussion of Table 1.

```
Detector Type: tracker
Particles Detected: 4
Material: Silicon
Layers: 5
Detection Efficiency: 0.9
...
Detector Type: muon chamber
Particles Detected: 1
Material: Drift tubes
Layers: 8
Detection Efficiency: 0.92
```

Listing 16: Detector Behaviour

The simulation concludes with explicit destruction of detector instances and clean-up of particle objects. Smart pointers and modular control ensure that memory management is correctly handled across all branches.

4 Discussion

This project demonstrates object oriented code usage in modelling high-energy particle interactions. The simulation reflects the behaviour of detector systems like ATLAS and CMS while remaining clear. Inheritance from a `Particle` base class enables shared physical properties while supporting type-specific behaviours. Similarly, detectors inherited from a `Detector` base class, allowing polymorphic response and central control via the `DetectorManager`. While these choices added initial design complexity, they also allow for easy extension (e.g., adding pair production or tau decay) without rewriting existing logic. Use of `shared_ptr` across all dynamically created objects ensured memory safety, particularly during complex decay branching and detector interaction chains. `Logger` is currently located within the Interactions module, this reflects its use across simulation stages including decays and detector recording. It could also logically sit within a Core or Utility namespace. The simulation includes several features that enhance realism, for example, probabilistic detection efficiencies and energy smearing, realistic branching ratios for tau decays, particle inference based on detector-hit patterns and missing energy estimation due to invisible neutrinos.

While the simulation captures many realistic aspects of particle detection, several simplifications remain. Particle propagation through the detector is treated instantaneously without modelling spatial trajectories or angular distributions, limiting the system's ability to simulate track curvature or spatial separation of decay vertices. Similarly, magnetic field effects, such as bending of charged particles, are omitted, and relativistic corrections for time dilation or energy loss through secondary processes are not considered. Future extensions could address these physical limitations to better match the detailed response of real high-energy detectors like ATLAS and CMS. Despite these simplifications, the model successfully reproduces key physics signatures (e.g. electron detection, tau decay cascades, photon invisibility), demonstrating the validity of the pipeline. Handling tau

decay presented a non-trivial branching problem with dynamic object creation. This was resolved via polymorphism and careful memory management. Photon interactions introduced domain-specific constraints, such as energy thresholds for pair production and angle limits for Compton scattering-requiring explicit checks and validation logic. Balancing output verbosity was another challenge. While verbose outputs aid debugging, they can overwhelm users. Use of modular logging and a configurable `VERBOSE` macro helps manage this trade-off. While the use of `std::shared_ptr` throughout the codebase simplified memory management, it introduced non-negligible reference-count overhead and risk of cyclic dependencies. In future iterations, switching to `std::unique_ptr` combined with explicit factory functions could reduce runtime costs and improve ownership clarity. Similarly, the inheritance hierarchy offered code reuse at the expense of coupling; a component-based design using composition might better accommodate new detector types without proliferating subclasses.

5 Conclusion

This project successfully demonstrates an object oriented simulation of particle interactions with a simplified detector system, inspired by high-energy physics experiments like those at the LHC. Using design, inheritance, and smart pointers, the simulation models realistic processes such as tau decays, photon interactions, and subsystem-based detection. The framework mirrors physical simulations, particles inherit shared behaviours via a polymorphic base class, while detectors operate independently through encapsulated logic and probabilistic efficiency models. The system supports dynamic branching, energy inference, and end-to-end analysis, capturing essential features of particle detection workflows. Results demonstrate that particles are classified effectively using detection trail patterns, and missing energy (typically from neutrinos or non-interacting photons) is correctly inferred. The calorimeter's error margin and total detection counts validate both measurement realism and software functionality. Future improvements could include addressing physical limitations such as magnetic field effects and relativistic corrections for time dilation or energy loss through secondary processes. Overall, this work provides a technical demonstration of object oriented code in scientific simulations and allows for an understanding of particle detector logic through code.

References

[1] Frank Close. *Particle Physics: A Very Short Introduction*. Oxford University Press, 2004.

[2] CERN. *ATLAS Experiment*. Accessed: 2025-04-23. CERN. 2024. URL: https://home.cern/science/experiments/atlas.

[3] CERN. *CMS Experiment*. Accessed: 2025-04-23. CERN. 2024. URL: https://home.cern/science/experiments/cms.

[4] Georges Aad et al. "Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC". In: *Physics Letters B* 716.1 (2012), pp. 1–29.

[5] David Tong. *The Standard Model*. https://www.damtp.cam.ac.uk/user/tong/sm/standardmodel.pdf. Accessed: 2025-05-21.

[6] David J. Griffiths. *Introduction to Elementary Particles*. 2nd, Revised. Wiley-VCH, 2008.

[7] Lucid Software Inc. *Lucidchart*. Accessed: 2025-05-02. 2008. URL: https://lucid.app.

[8] Glen Cowan. *Statistical Data Analysis*. Oxford Science Publications, 1998.

[9] Thamara. *Pragma Once vs Header Guards*. Accessed: 2025-05-21. 2023. URL: https://thamara.dev/posts/pragma-once-vs-header-guards/.

[10] Changqi. *What is the #pragma once in C++?* Accessed: 2025-05-20. 2023. URL: https://medium.com/@changqi_37311/what-is-the-pragma-once-in-c-237bdf4edca8.

[11] Lisa Randall. *Warped Passages: Unraveling the Mysteries of the Universe's Hidden Dimensions*. HarperCollins, 2005.

[12] Particle Data Group. *Review of Particle Physics: Tau Branching Fractions*. 2024. URL: https://pdg.lbl.gov/2024/reviews/rpp2024-rev-tau-branching-fractions.pdf.

According to Overleaf's Word Count, this Document Contains 2197 Words.