Simulating Particle Detection in Detector Systems Using C++

PHYS30762 C++ Project

Rosa Roberts

Department of Physics and Astronomy, University of Manchester

11th May 2025

This project simulates a particle detector inspired by experiments such as ATLAS and CMS, using object-oriented programming principles including inheritance, polymorphism, and encapsulation. Sub-detectors and particles inherit from abstract base classes. A detector class manages multiple sub-detectors and simulates particle interactions, calculating 'true' and 'detected' energies based on detector resolution. Detector resolution effects are simulated using the C++ Standard Library's random number generator. Templates are used to construct detector configurations. The simulation successfully models Higgs, Z boson, and anti-top quark decays, yielding physically reasonable results.

1. INTRODUCTION

High-energy particle physics experiments rely on sophisticated detector systems to identify particles produced in collisions and measure their properties. These systems consist of multiple sub-detectors, each designed to interact with different particle types. Understanding how different particles interact with these components is crucial for identifying them and interpreting collision events [1]. This project simulates such a particle detector with sub-detector components - tracker, electromagnetic (EM) calorimeter, hadronic calorimeter, and muon spectrometer - through which photons, electrons, positrons, hadrons, muons, and neutrinos are passed. Particle identification is inferred from which sub-detectors 'register' a signal [2].

Abstract base classes define particles and sub-detectors, with derived classes specifying behaviours and types. Each particle holds a four-momentum object, used by sub-detectors to read energy. The program was extended to simulate a realistic detector using the Standard Library's random number generator (STL RNG). A generic detector class manages sub-detectors. A separate header, *DetectorConfig.h*, and implementation file, defines pre-built detector configurations resembling experiments like ATLAS. Users can compute values like missing transverse energy (MET) and invariant mass (IM) from detector outputs.

2. DESIGN AND IMPLEMENTATION

The code is structured around three main class hierarchies - *Particle*, *SubDetector*, and *DetectorConfig* - which model particle behaviour, sub-detector responses and detector layouts, respectively. Polymorphism enables generic handling of abstract base classes while retaining the specific functionality of derived classes.

To improve modularity and prevent future naming conflicts, the code is organised into three namespaces, each grouping related classes. Shared functionality is implemented in non-virtual base class methods to reduce code duplication, while pure virtual functions allow derived classes to define custom behaviour. Some validation and exception handling are delegated to derived classes, with static validation functions in base classes (some protected) enforcing constraints during object construction or property setting. Setters throw exceptions to prevent invalid data assignment.

2.1. The Four Momentum Class

The *FourMomentum* class represents a particles' relativistic four-momentum, with components for momentum and energy. It provides methods to compute key quantities such as mass, transverse momentum, momentum magnitude, pseudorapidity and system invariant mass. A *print()* method supports output in derived *Particle* classes.

The class includes *validate_components(),* a static function enforcing physical constraints: non-negative energy, valid momentum components (checked using *std::numeric_limits*), and the mass-shell condition. The *set_momentum_components()* method throws exceptions if these checks fail. The *calculate_system_invariant_mass()* function throws if called on an empty vector. The Rule of 5 is implemented to ensure proper correct copying and moving of objects.

2.2. The Particle Class

The *Particle* class is an abstract base class defining the interface for all particle types. Derived classes (e.g. *Electron*, *Photon*) inherit from this class and implement type-specific behaviour. *Particle* contains a *FourMomentum* object for energy-based calculations used in the *Detector* class. This class hierarchy is shown in Figure 1.

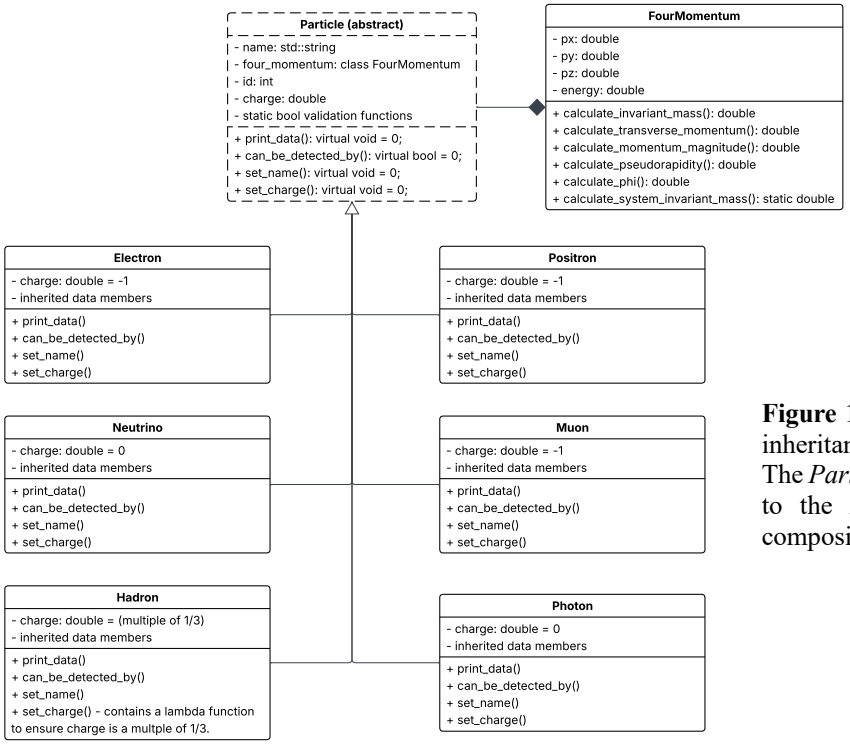

Figure 1. A UML diagram showing the inheritance chain of the *Particle* class. The *Particle* abstract base class is related to the *FourMomentum* class through composition.

The derived particle classes support the modelling of decays, such as Higgs to diphoton, a Z boson decaying to an electron-positron pair and an anti-top quark decaying to a b quark, muon and neutrino. The id property distinguishes between particles of the same type.

The Hadron class includes a locally defined lambda function, *is_valid_hadron_charge()*, to ensure hadron charges are valid multiples of 1/3 (reflecting quark composition). This lambda function is shown in Figure 2. This logic is specific and not reused elsewhere.

```
// Helper lambda to validate if charge is a multiple of 1/3
  auto is_valid_hadron_charge = [](double charge)
  {
    // Scale the value by 3.0 and round it to the nearest integer
    double scaled = charge * 3.0;
    double rounded = std::round(scaled);
    // Check if the absolute difference is within a small epsilon
    double epsilon = std::numeric_limits<double>::epsilon() * 100;
    // Check if the scaled value is close to the rounded value
    return std::abs(scaled - rounded) < epsilon;
  };
```

Figure 2. Lambda function defined within the Hadron class to verify that hadron charges are valid multiples of 1/3, reflecting underlying quark composition. This function is only used locally within the Hadron class.

The pure virtual function *can_be_detected_by()* allows each particle to define its sub-detector response. For example, electrons/positrons leave tracks and deposit energy in EM calorimeters, muons leave tracks and are detected in muon spectrometers, and photons are only detected in EM calorimeters. Hadrons leave tracks and interact with hadronic calorimeters. Neutrinos pass undetected through all sub-detectors. The Rule of 5 is implemented so that particle resources are handled correctly during copying and moving.

2.3. The Sub-detector Class

The *SubDetector* class is an abstract base class for sub-detector components. Derived classes (e.g. *Tracker*, *EMCalorimeter*) implement specific detection logic. An intermediate abstract class, *Calorimeter,* provides shared properties for *EMCalorimeter* and *HadronicCalorimeter*. This class hierarchy is shown in Figure 3.

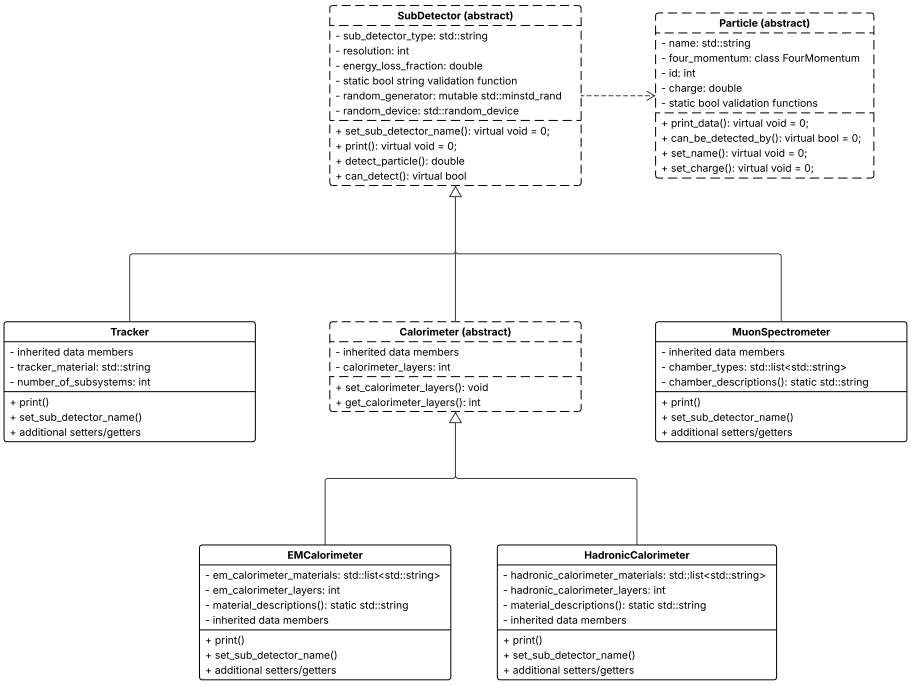

Figure 3. A UML diagram showing the inheritance chain of the *SubDetector* class and it's derived classes. The *SubDetector* abstract base class is dependent on the *Particle* class.

Detection is simulated realistically using a mutable *std::minstd_rand* RNG to apply energy smearing based on resolution. The RNG is seeded with *std::random_device* to ensure different outputs across program executions. It is made mutable to allow modification in *const* methods like *detect_particle(),* which calculates the energy of a particle in a sub-detector (assuming some small fraction of the particle energy is lost in the sub-detector) using a normal distribution with standard deviation proportional to the resolution of the sub-detector. The RNG is used to sample from this distribution and *std::abs()* is used to ensure non-negative energy values. This is shown in Figure 4. This simulation returns energy values from all sub-detectors, including trackers, for consistency – even though real trackers measure position and momentum. This ensures a uniform interface for all sub-detectors and simplifies particle identification.

```
// For realistic detection, apply resolution effects by generating a distribution
  double mean = energy_loss_in_detector;
  double std_dev = energy_loss_in_detector * (detector_resolution / 100.0);
  // Create a normal distribution with mean and standard deviation
  std::normal_distribution<double> distribution(mean, std_dev);
  // Generate a random measured energy based on the detector resolution
  double measured_energy = distribution(const_cast<std::minstd_rand&>(random_generator));
  // Use absolute value to ensure the measured energy is not negative
  return std::abs(measured_energy);
```

Figure 4. Code snippet of the *detect_particle()* method which simulates energy measurements in a sub-detector by applying energy smearing based on detector resolution. The output energy is sampled from a normal distribution with a mean equal to the post-energy-loss value and a standard deviation proportional to the detector resolution. The result is passed through *std::abs()* to ensure non-negative energy values.

The *std::minstd_rand* linear congruent engine is moderately fast and has a very small storage requirement - suitable for this short project [3]. It is deterministic once seeded. The *std::random_device* used provides a source of non-deterministic random numbers (using hardware entropy if available) [4]. This ensures that the seed used to initialise the RNG is as random as possible.

STL containers are used throughout: *std::list* stores chamber types in *MuonSpectrometer* to allow easy addition of chamber types, while *std::set* and *std::map* manage material types in calorimeters for fast lookup and value associations. The *Tracker* class uses <*climits*> to constrain the number of tracker sub-systems within *INT_MAX*.

The hierarchy leverages polymorphism: *can_detect()* delegates to the *Particle* class using *can_be_detected_by(),* allowing easy addition of new particle types without altering sub-detectors.

Copy operations are omitted: *std::random_device* is non-copyable, and sub-detectors are fixed in real experiments. Move operations are also excluded, as sub-detectors are not relocated in real experiments, leaving the past location in an initialised state. Sub-detectors are owned by the *Detector* class, instantiated once and used throughout the program.

2.4. The Detector Configuration

The configuration system uses a small class hierarchy of static structs, with a base *DetectorConfig* struct. Derived structs like *ATLASConfig* and *CMSConfig* define specific layouts by overriding the *configure()* method, adding sub-detectors via the templated *add_sub_detector()* function. This design enables consistent, easily extendable setups without altering detection logic. This hierarchy is shown in Figure 5.

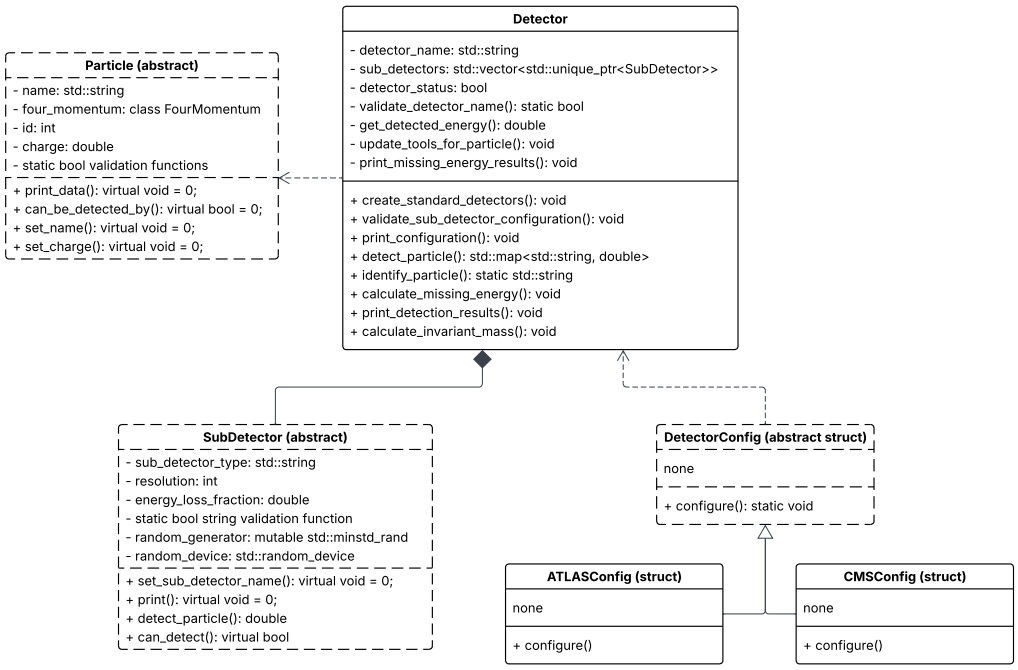

Figure 5. A UML diagram showing the inheritance chain of the *DetectorConfig* class and it's derived classes. The *Detector* class is dependent on the *DetectorConfig* and *Particle* class hierarchies and is related to the *SubDetector* class hierarchy through composition.

Structs are used instead of classes for brevity, as they default to public members. To configure a detector, the *Detector* class calls the templated *configure_detector()* function, which invokes the appropriate static *configure()* method to populate the *sub_detectors* vector. The templated functions are shown in Figure 6. Containers like *std::list* pass material or chamber types to sub-detectors for easy modification of these ordered collections. New configurations can be added by defining a new derived struct, preserving existing logic. Currently, the *Detector* class implementation enforces that each configuration includes exactly one of each sub-detector type. This is explained further in Section 2.5.

```cpp
// Template function to add a sub-detector of any type to the sub_detectors vector
template<typename T, typename... Args>
void add_sub_detector(std::vector<std::unique_ptr<DetectorSubsystems::SubDetector>>&
sub_detectors, Args&&... args)
{
  // Creates a unique pointer to a sub-detector and adds it to the sub_detectors vector
  sub_detectors.emplace_back(std::make_unique<T>(std::forward<Args>(args)...));
}

// Template function to configure a detector using a specific configuration class
  // This function forwards the configuration process to the provided DetectorConfig class.
  template<typename DetectorConfig>
  void configure_detector(std::vector<std::unique_ptr<DetectorSubsystems::SubDetector>>&
  sub_detectors)
  {
    DetectorConfig::configure(sub_detectors);
  }
```

Figure 6. Templated functions used in the detector configuration system. The *add_sub_detector()* function accepts any sub-detector type and its constructor arguments, forwarding them to *std::make_unique* to construct and store the sub-detector in the *Detector* class's *sub_detectors* vector. The *configure_detector()* function calls the static *configure()* method of a given configuration struct.

2. 5. The Detector Class

The *Detector* class manages the full detection system using a *std::vector<std::unique_ptr<SubDetector>>*. It provides key methods such as *detect_particle()* and *identify_particle()*, along with functions to compute IM and MET from a system of particles. Copy and move operations are disabled, as real detectors are fixed and cannot be meaningfully duplicated or transferred. Copying risks multiple *unique_ptr* ownership, while moving would invalidate the detector state.

Sub-detectors are configured through *create_standard_detectors(),* which calls the templated *configure_detector()* function (see Section 2.4.), followed by *validate_sub_detector_configuration()*, which throws if multiple instances of the same sub-detector type are detected.

The *detect_particle()* function simulates a particle's passage through all sub-detectors, updating its energy after each interaction and returning a *std::map* of detector readings for easy data retrieval. This checks the detector is active and each sub-detector's *detect_particle()* method is called.

The *identify_particle()* function interprets these readings, using non-zero readings to construct a bitmask of triggered sub-detectors. A *switch* statement maps patterns to particle types. MET is computed from the vector difference between true and detected momenta, and IM is calculated using the *FourMomentum::calculate_system_invariant_mass()* function and compared to known masses. Figure 7 shows the bitmask logic.

```cpp
// Encode the detection pattern using a 4-bit bitmask
  // Each bit corresponds to a sub-detector:
  // For example, bitmask = 0b0011 means Tracker + EM Calorimeter were triggered
  int bitmask = 0;
  if(detected_by_tracker) {bitmask |= 1;}  // 0001
  if(detected_by_em_calorimeter) {bitmask |= 2;}  // 0010
  if(detected_by_hadronic_calorimeter) {bitmask |= 4;}  // 0100
  if(detected_by_muon_spectrometer) {bitmask |= 8;}  // 1000
  // Use the bitmask to identify the particle based on known interaction signatures
  switch (bitmask)
  {
    case 0b0010: return "Photon"; // EM Calorimeter only
    case 0b0011: return "Electron or Positron"; // Tracker + EM Calorimeter
    case 0b0101: return "Hadron"; // Tracker + Hadronic Calorimeter
    case 0b1001: return "Muon"; // Tracker + Muon Spectrometer
    case 0b0000: return "Nothing Detected (Possible Neutrino)"; // No sub-detector triggered
    default: return "Unknown"; // Any other pattern not explicitly handled
```

Figure 7. Bitmask-based encoding of detector responses for particle identification. Each bit in the 4-bit bitmask represents whether a specific sub-detector was triggered. The final bitmask is matched against known interaction patterns using a *switch* statement to classify the particle.

2.6. The Main Program

The main program simulates three particle decay events - Higgs, Z boson, and anti-top quark - each encapsulated in its own function that returns a *std::vector<unique_ptr<Particle>>*. For example, *simulate_higgs_decay()* creates two oppositely directed photons to conserve momentum.

The *process_physics_event()* function activates the detector, collects readings, identifies particles, and outputs results, including IM and MET. The *run_complex_simulation()* function coordinates the full workflow, and *main()* wraps execution in a *try-catch* block to handle errors gracefully.

3. RESULTS

The detector used a standard ATLAS configuration. The following results were gathered from a single execution, and the exact values vary across executions due to the randomness of the RNG used to generate energy values. However, the analysis of the results is the same regardless of the exact values. In Higgs decay, the EM calorimeter recorded 56.46 GeV and 62.24 GeV for the two photons (true energies: 60 and 65 GeV, respectively). The diphoton invariant mass was 124.86 GeV, close to the expected 125 GeV, confirming accurate mass reconstruction, despite resolution effects. The missing energy showed a small discrepancy between true and detected energy (6.30 GeV) due to the detector resolution. The MET was detected as 5.40 GeV, compared to the true MET of 5.83 GeV.

The 10 GeV MET threshold, seen in the anti-top decay output, flags significant deviations that may indicate the presence of weakly interacting particles or significant momentum errors. The energy loss in each detector is exaggerated compared to reality for demonstration. The relatively large MET values (5-6 GeV) observed even in the Higgs decays - where no neutrinos are present - results from using simplified, non-physical momentum vectors for illustrative purposes.

For the Z boson decay, the detector readings showed 42.81 GeV in the tracker and 40.42 GeV in the EM calorimeter for the electron (true energy: 45 GeV) and 33.55 GeV in the tracker and 32.03 GeV in the EM calorimeter for the positron (true energy: 35 GeV). The IM was 79.53 GeV, lower than the theoretical 91.2 GeV, due to detector resolution and initial momentum setup.

In the anti-top quark decay, the true particle energies were 80 GeV, 40 GeV, and 45 GeV, respectively. The readings for the b quark showed 79.63 GeV in the tracker and 60.85 GeV in the hadronic calorimeter. The readings for the muon showed 38.95 GeV in the tracker and 37.17 GeV in the muon spectrometer. The anti-neutrino left no signal, as expected. The IM was 132.38 GeV, lower than the theoretical anti-top quark mass of 173 GeV. This is expected given that the neutrino energy is not detected, resulting in significant missing energy. The detected MET was 64.33 GeV, large due to the neutrino's undetected momentum.

4. DISCUSSION

This project successfully demonstrates object-orientated programming and key high-energy physics concepts through a simple particle detector simulation. While effective, the current implementation includes only basic particles and uses a simplified interaction model. The current detection pattern in the *Detector* class's *identify_particle()* function works well for basic particle identification but will struggle with more complex scenarios found in actual experiments. The energy resolution model is simplified compared to real detector responses which may not follow Gaussian distributions.

Future improvements could introduce more particle types (e.g. more hadrons, tau leptons) and their own methods. The Hadron class could be expanded into its own hierarchy, with distinct classes for mesons and baryons, reflecting their different quark compositions and decay properties. Quantum numbers such as lepton number and baryon number conservation could be implemented as validation checks for interactions.

The simulation does not account for the full detector geometry and magnetic field effects. Adding additional hierarchical structures, such as muon chambers, and creating new detector configurations would enhance particle identification. More detailed models of particle interactions with detector materials could be implemented for greater realism. The simulation does not account for processes such as bremsstrahlung, pair production and annihilation, which impact particle detection in real experiments. This would require more complex energy loss models in each sub-detector.

This simulation passes individual particles through the detector one at a time, unlike real experiments where bunches pass through simultaneously. This simplification helps track individual interactions with sub-detectors but ignores the effects of overlapping events.

5. CONCLUSION

This project successfully demonstrates how object-oriented programming techniques in C++ can be used to simulate the behaviour of a particle detector. While simplified compared to real detectors like ATLAS, it captures essential concepts while providing a foundation for future extensions, including additional sub-detectors, detector configurations and particle types. The use of an RNG allowed for more physically meaningful results. The simulation accurately reconstructed quantities such as invariant mass and missing transverse energy.

6. DECLARATIONS

This project was inspired by my recent lab work, 'Analysing Events with the Z Boson with the ATLAS Experiment'. The calculations implemented through the FourMomentum class reflect the quantities I worked with during that experiment. The simulated decay processes featured in this project were selected based on the interactions I studied as part of the lab.

The template functions described in Section 2.4. were developed with the use of AI to add and configure sub-detectors so that configurations can be easily extended and modified without altering the core code. The bitmask logic described in Section 2.5. was also developed with the use of AI with the aim of enabling simple and efficient particle identification. All other code is my own.

REFERENCES

[1] ATLAS collaboration, *About the ATLAS Experiment*, CERN, https://atlas.cern/about, last accessed 23.04.2025

[2] ATLAS collaboration, *Detector and Technology*, CERN, https://atlas.cern/Discover/Detector last accessed 23.04.2025

[3] CPP Reference, *Pseudo-random number generation*, https://en.cppreference.com/w/cpp/numeric/random, last accessed 24.04.2025

[4] CPP Reference, *std::random_device*, https://en.cppreference.com/w/cpp/numeric/random/random_device, last accessed 24.04.2025

Word count: 2196

(Using the word count in Word, not including figure captions, references or declarations.)

Code was formatted using the *minted* packaged in LaTeX.