Particle Catalogue
C++ Project-2

Luca Vicaria
Department of Physics and Astronomy, University of Manchester
(Dated: May 12, 2024)

This project presents a comprehensive C++ simulation and catalogue of Standard Model particles by utilising object-oriented programming principles to encapsulate the characteristics and behaviors of fundamental particles. By leveraging a class hierarchy, the simulation effectively models the interactions and properties of quarks, leptons, bosons, including specialised attributes such as charge, spin, and four-momentum. Advanced C++ features such as templates, exception handling, and the Standard Template Library (STL) are integral to the project, enhancing its robustness and scalability.

The project introduces a custom container, wrapped around existing STL containers, to manage a dynamic collection of particle objects. This facilitates operations like searching for particles by type and calculating cumulative properties, such as the total four-momentum across all particles in the catalogue. The challenge features added to the project include a physically consistent implementation of the four-momentum vector, ensuring that the calculated invariant mass matches the expected values for each particle, thereby increasing the simulation's accuracy. Additionally, the project employs a lambda functions for streamlined operations on particle data, and exception handling mechanisms to robustly manage runtime errors.

1. INTRODUCTION

The Particle Catalogue project is an endeavour aimed at developing a comprehensive digital catalogue for the Standard Model of particle physics, a theory that has revolutionised our understanding of the fundamental constituents of the universe. The Standard Model, established in the latter half of the $20th$ century, consolidates our knowledge of elementary particles and the three fundamental forces of nature (excluding gravity). The framework for this model was largely solidified in the $1970s$, following the independent contributions of physicists, including Sheldon Glashow, Steven Weinberg, and Abdus Salam, who collectively advanced the electroweak theory, a cornerstone of the Standard Model [1]. This theoretical construct not only describes particles such as quarks, leptons, and bosons but also predicts their interactions via strong, weak, and electromagnetic forces. The particles outlined by the Standard Model are detailed in Table I.

Type	**Particle**	**Mass**	**Charge**	**Spin**
Quarks	Up	$\sim 2.3\,\mathrm{MeV}/c^2$	'+2/3	1/2
	Down	$\sim 4.8\,\mathrm{MeV}/c^2$	$-1/3$	1/2
	Charm	$\sim 1.275\,\mathrm{GeV}/c^2$	$+2/3$	1/2
	Strange	$\sim 95\,\mathrm{MeV}/c^2$	$-1/3$	1/2
	Top	$\sim 173.07\,\mathrm{GeV}/c^2$	$+2/3$	1/2
	Bottom	$\sim 4.18\,\mathrm{GeV}/c^2$	$-1/3$	1/2
Leptons	Electron	$0.511\,\mathrm{MeV}/c^2$	-1	1/2
	Muon	$105.7\,\mathrm{MeV}/c^2$	-1	1/2
	Tau	$1.777\,\mathrm{GeV}/c^2$	-1	1/2
	Electron Neutrino	$< 2.2\,\mathrm{eV}/c^2$	0	1/2
	Muon Neutrino	$< 0.17\,\mathrm{MeV}/c^2$	0	1/2
	Tau Neutrino	$< 15.5\,\mathrm{MeV}/c^2$	0	1/2
Gauge Bosons	Photon	0	0	1
	Gluon	0	0	1
	Z Boson	$91.2\,\mathrm{GeV}/c^2$	0	1
	W Boson	$80.4\,\mathrm{GeV}/c^2$	±1	1
Scalar Boson	Higgs Boson	$\sim 126\,\mathrm{GeV}/c^2$	0	0

TABLE I. This table provides a detailed list of the fundamental particles described by the Standard Model of particle physics, specifying each particle's type, mass, charge, and spin. It includes three major categories: Quarks, Leptons, and Gauge Bosons, alongside the Higgs boson. Note that each particle also has a corresponding antiparticle (except for the photon, Z Boson, and Higgs), which typically has the inverse charge and, for quarks, the inverse colour charge.

A particle catalogue serves as an essential resource for physicists and educators, providing a structured database of particle properties such as mass, charge, and spin, along with their interactions and decay processes. Such a catalogue is pivotal not just for academic and instructional purposes but also for ongoing and future experimental and theoretical research. By systematising particle information in a robust, interactive format, the catalogue can enhance the accessibility of particle physics, supporting simulations, data analysis, and potentially aiding in the discovery of new particles or unexplained.
As the field of particle physics continues to evolve, the particle catalogue could be expanded, potentially incorporating real-time data from particle physics experiments around the world, such as those conducted at the

Large Hadron Collider (LHC).

2. CODE DESIGN AND IMPLEMENTATION

2.1. Hierarchical Structural Overview of Classes

Within the codebase, the abstract base class ‘Particle‘ starts the hierarchical chain, with ‘GenericParticle‘ inheriting from it. From ‘GenericParticle‘, the classes ‘Lepton‘, ‘Boson‘, and ‘Quark‘ extend this basic framework. ‘Lepton‘ is further branched into derived classes specifically for electrons, muons, taus and neutrinos, while ‘Boson‘ divides into subclasses for photons, gluons, W bosons, Z bosons and the higgs boson. The ‘FourMomentum‘ class is composed within these classes, allowing each particle to manage its four-momentum and energy.

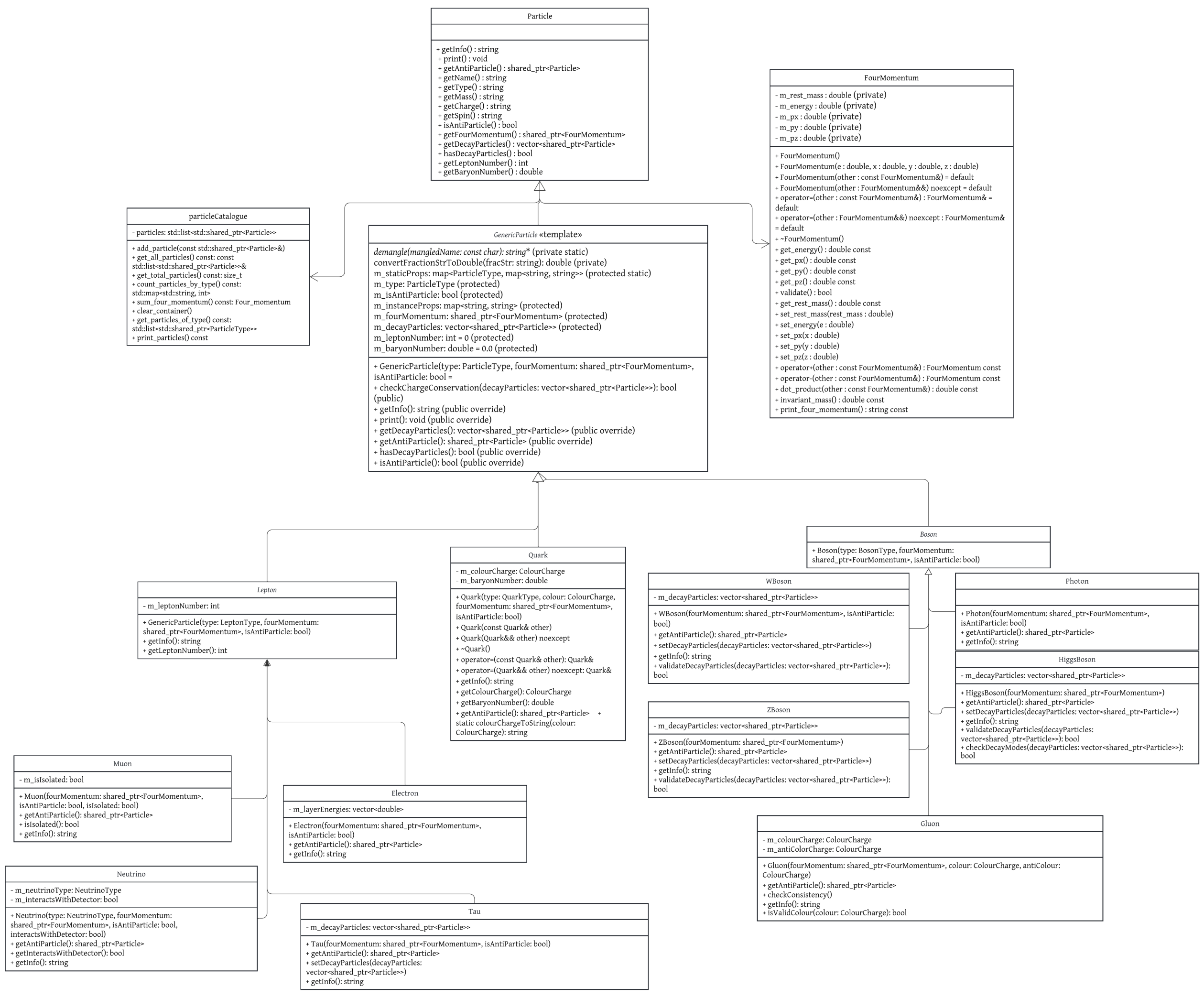

FIG. 1. UML class diagram representing code hierarchy [2].

Throughout the code, enumeration classes (enum classes) and mappings is instrumental. Enum classes such as ‘LeptonType‘, ‘QuarkType‘, ‘BosonType‘, ‘ForceType‘, and ‘ColourCharge‘ are utilised to enforce strong type checking and scope control. This approach minimises errors by ensuring that each type of particle and characteristic can only assume explicitly defined values, thereby preventing the assignment of incorrect or undefined types.

The project further utilises the strengths of enum classes by associating specific properties such as name, mass, charge, and spin—with each particle type through the use of mappings. This is implemented via a template specialisation for a ‘map‘ container that holds properties for each enum class type. For instance, properties of leptons are stored in a map where each ‘LeptonType‘ (e.g., Electron, Muon) is associated with its corresponding attributes and are therefore accessible through simple queries. An example of this is shown in Listing 1.

```
// Define properties for Leptons
template <>
std::map<LeptonType, std::map<std::string, std::string>> GenericParticle<LeptonType>::
    m_staticProps {
    {LeptonType::Electron, {{"name", "Electron"}, {"mass", "0.511"}, {"charge", "-1"}, {"spin"
    , "0.5"}}},
    {LeptonType::Muon, {{"name", "Muon"}, {"mass", "105.66"}, {"charge", "-1"}, {"spin", "0.5"
    }}},
    {LeptonType::Tau, {{"name", "Tau"}, {"mass", "1776.8"}, {"charge", "-1"}, {"spin", "0.5"
    }}},
    {LeptonType::Neutrino, {{"name", "Neutrino"}, {"mass", "0"}, {"charge", "0"}, {"spin", "
    0.5"}}}
};
```

Listing 1. Example implementation of a map container for the Lepton class.

The advantages of integrating enum classes with mappings are that they ensure type safety while also centralising particle properties, simplifying updates and maintenance. This allows for scalability, as adding new particle types or properties requires minimal modifications.

2.2. Abstract Base Particle Class

The class 'Particle' serves as an abstract base class from which all specific particle types can inherit. By designating this class as abstract, it explicitly prevents direct instantiation, reinforcing a structured approach where specific particle types—like leptons, quarks, or bosons—must be defined through concrete subclasses. Additionally, the Particle class provides a common interface for all types of particles, enabling functions to operate on collections of particles, regardless of their specific types, and interact with them through the interface defined by Particle. Each subclass deriving from the 'Particle' class must implement a set of pure virtual functions such as 'getInfo()', 'print()', 'getAntiParticle()', and others related to particle properties like mass, charge, and spin. This requirement ensures consistent behavior and interface across all particle types, which is essential for maintaining code reusability and simplifying maintenance. Furthermore, the use of virtual functions enables dynamic binding, a core aspect of polymorphism, where the specific method executed depends on the type of the object at runtime, not the type of reference or pointer. This feature is crucial for functions designed to handle multiple particle types dynamically, allowing them to operate on collections of particles irrespective of their specific types. The generic setup of the 'Particle' class also supports type safety and flexibility. Users can create collections of particles, such as '$std :: vector < std :: shared_ptr < Particle >>$', facilitating the management of mixed particle types within simulations. This approach enhances the extensibility of the system, as adding new particle types becomes straightforward without necessitating changes to the existing code base. Overall, the abstract base class 'Particle' creates a robust and flexible architecture for the particle catalogue, aligning with best practices for achieving a modular and maintainable codebase.

2.3. The Generic Particle

The '$GenericParticle < T >$ ' template class extends the generic interface provided by the abstract 'Particle' class, and serves as an intermediate class in the hierarchy. This design enables behaviors and properties specific to a category of particles, such as leptons or quarks, to be defined and managed efficiently, reducing redundancy and enhancing code maintainability. Within this framework, 'GenericParticle' encapsulates shared attributes like mass, charge, spin, and four-momentum. This template-based approach ensures type-specific handling while maintaining type safety, leveraging compile-time checks to manage different particle properties and behaviors efficiently. For instance, the 'GenericParticle' class can handle different static properties or implement methods that vary based on the particle type, like 'LeptonType' or 'QuarkType'. In terms of performance, templates are resolved at compile time, which eliminates the overhead associated with runtime polymorphism when it is unnecessary. This leads to more efficient code execution. Additionally, the template can be specialised for certain types to provide unique implementations where needed, optimising performance or altering behavior without compromising the template's broader applicability. In summary, the combination of 'Particle' as an abstract base class and '$GenericParticle < T >$ ' as a template provides a robust framework for managing a diverse array of particle types in a particle physics simulation. This structure supports code reusability, maintainability, and scalability, effectively balancing runtime polymorphism for flexibility with compile-time type safety for enhanced performance and correctness.

2.4. Leptons

The 'Lepton' class, deriving from 'GenericParticle', serves as a base for specialised lepton classes including 'Electron', 'Muon', 'Tau', and 'Neutrino'. Each derived class introduces unique functionalities specific to their respective particles. For example, the 'Electron' class incorporates a random number generator, utilising to distribute total energy across calorimeter layers. As well as this, the 'Tau' class features an automated decay particle function, utilising the built-in 'push back' method to populate decay modes while ensuring lepton, baryon, and charge conservation. This is demonstrated in Listing 2.

```
void setDecayParticles(const std::vector<std::shared_ptr<Particle>>& decayParticles) {
    m_decayParticles.clear();
    if(validateDecayParticles(decayParticles))
        std::copy(decayParticles.begin(), decayParticles.end(), std::back_inserter(
    m_decayParticles));
    else
        std::cerr << "\nInvalid decay particles for Tau" << std::endl;
}
```

Listing 2. Decay particle validation checks.

2.5. Quarks

The ‘Quark‘ class, derived from ‘GenericParticle‘, specifically manages quark particles, allowing for particle creation based on type and colour charge. The class is structured to allow for all quark types, each characterised by unique properties such as mass, charge, and spin stored in a static property map. Quarks are further defined by a colour charge, critical for simulating particle interactions, with possible values including Red, Green, Blue, and their antiparticles. The class features the ‘colourChargeToString‘ method, which converts the colour charge enum into a human-readable string format. This functionality allows for the clear and concise output of quark characteristics. The implementation of this method is depicted in Listing 3.

```cpp
// Convert colour charge to string representation
static std::string colourChargeToString(ColourCharge colour) {
  switch (colour) {
    case ColourCharge::Red: return "Red";
    case ColourCharge::Green: return "Green";
    case ColourCharge::Blue: return "Blue";
    case ColourCharge::AntiRed: return "AntiRed";
    case ColourCharge::AntiGreen: return "AntiGreen";
    case ColourCharge::AntiBlue: return "AntiBlue";
    default: return "Unknown";
  }
}
```

Listing 3. Converting colour charge of enum type to a string type.

2.6. Bosons

The ‘Boson‘ class, derived from ‘GenericParticle‘, represents elementary particles such as Photons, W and Z Bosons, Gluons, and the Higgs Boson, each characterised by distinct properties specified in a static properties map. Classes like ‘ZBoson‘, ‘WBoson‘, and ‘HiggsBoson‘ are designed to handle decay particles in a similar way to the ‘Tau‘ class, ensuring attributes like charge conservation are preserved. As well as this, the ‘Gluon‘ class distinctively manages colour charges, ensuring valid colour and anti-colour charges in each instance to accurately simulate the strong force.

2.7. Four-momentum

The ‘FourMomentum‘ class is designed to manage the energy and momentum of particle instances and includes overloaded operators for addition and subtraction operations as well as methods for calculating the invariant mass, which is used throughout the code to ensure physical consistency and is shown in Listing 4.

```cpp
double invariant_mass() const {
    return std::sqrt(std::max(0.0, this->m_energy * this->m_energy - (this->m_px * this->m_px
   + this->m_py * this->m_py + this->m_pz * this->m_pz)));
}
```

Listing 4. Invariant Mass Calculation

2.8. Advanced Features and C++ Techniques

As well as templates and STL map containers which have already been discussed, advanced features within this project include the use of exception handling and lambda functions. The project utilises exceptions extensively to manage unexpected conditions gracefully, particularly in the sections where particles are assigned decay products. In the example below, a Tau particle is initialised and its decay products are set and in order to handle any potential exceptions stemming from invalid particle properties, these operations are encapsulated within a ‘try‘ block, followed by a ‘catch‘. The implementation of this is shown Listing 5.

```cpp
try {
    tau1->setDecayParticles({
        std::static_pointer_cast<Particle>(std::make_shared<Electron>(
            std::make_shared<FourMomentum>(0.511, 0, 0, 0))),
        std::static_pointer_cast<Particle>(std::make_shared<Neutrino>(
            NeutrinoType::ElectronNeutrino, std::make_shared<FourMomentum>(0, 0, 0, 0), true,
   false)),
        std::static_pointer_cast<Particle>(std::make_shared<Neutrino>(
            NeutrinoType::TauNeutrino, std::make_shared<FourMomentum>(0, 0, 0, 0), false,
   false))
    });
    tau1->print();
} catch(const std::exception& e) {
    std::cerr << "Error: " << e.what() << '\n';
}
```

Listing 5. Handling exceptions when setting decay particles for a Tau particle

This project also utilises argument throwing in order to enforce conservation laws. In the function ‘*checkConsistency*‘, an exception is thrown if the gluon particles do not have valid colour and anti-colour charges. This preemptive check prevents the application from proceeding with an invalid state, thereby avoiding more complex errors later on. The specific implementation is shown in Listing 6.

```
void checkConsistency() const {
    if(!isValidColour(m_colourCharge) || !isValidColour(m_antiColorCharge)) {
        throw std::invalid_argument("Gluon must have both colour and anti-colour charges, with
        valid values.");
    }
}
```

Listing 6. Checking consistency of gluon charges and throwing an exception if invalid

Lambda functions in this project serve as a means to write succinct and efficient in-line code that can execute operations within a localised area. The '*printAllParticles*' lambda captures the '*particleCatalogue*' by reference, allowing for an efficient traversal and operation on the container without the overhead of passing parameters. This function is demonstrated in Listing 7.

```
auto printAllParticles = [&particleCatalogue]() {
    for(const auto& pair : particleCatalogue) {
        pair.second->print();
    }
};
```

Listing 7. Using a lambda function to print all particles in the catalogue

In summary, the use of exceptions, throwing arguments, and lambda functions not only enhances the code's robustness and error-handling capacity but also optimises performance and improves the clarity and maintainability.

2.9. User Interface and Implementation in Main

The 'main()' function serves as the central entry point for the application. It orchestrates the initialisation of the particle catalogue, handles user interactions, and manages the particle lifecycle. Initialisation of the particle catalogue is performed by the '*initializeParticles*()' function. This function employs an STL map, keyed by string identifiers, to associate each particle with a '$std :: uniqueptr < Particle >$ ' and is demonstrated in Listing 8. These smart pointers provide automatic memory management and exclusive ownership, which helps prevent memory leaks by automatically deallocating particles when they are no longer needed.

```
std::map<std::string, std::unique_ptr<Particle>> initializeParticles() {
    std::map<std::string, std::unique_ptr<Particle>> particleCatalogue;
    // Initialisation logic here
    return particleCatalogue;
}
```

Listing 8. Initialise Particle Catalogue

The 'loop()' function is designed to facilitate interactive user sessions. It prompts users to query specific particles and uses '*setConsoleColour*()' to enhance the readability of console text through dynamic colour changes, depicted in Listing 9.

```
void loop(std::map<std::string, std::unique_ptr<Particle>>& particleCatalogue) {
    std::string input;
    while(true) {
        std::cout<<"Enter a particle name or 'quit' to exit: ";
        std::getline(std::cin, input);
        if(input == "quit")
            break;
        // Additional interaction logic here
    }
}
```

Listing 9. Loop Function for User Interaction

To manage subsets of data efficiently, sub-containers such as vectors of '$std :: sharedptr < Particle >$ ' are utilised. This approach optimises memory usage and processing by limiting operations to relevant subsets, allowing us to calculate the number of particles of a specific type within the catalogue. This is shown in Listing 10.

```
std::vector<std::shared_ptr<Particle>> getParticlesOfType(const std::map<std::string, std::
    unique_ptr<Particle>>& catalogue, const std::string& type) {
    std::vector<std::shared_ptr<Particle>> particles;
    for(const auto& pair : catalogue) {
        if(pair.second->getType() == type)
            particles.push_back(std::shared_ptr<Particle>(pair.second.get(), [](Particle*){}))
    ;
    }
    return particles;
}
```

Listing 10. Get Particles of a Specific Type

Additionally, the project includes the functionality to simulate and visualise the decay processes of various particles by leveraging smart pointer utilities such as '$std :: makeshared$' and '$std :: staticpointercast$'. These are employed to construct and safely cast shared pointers, ensuring memory management and efficient control over the lifecycle of particle instances and their associated data. Below is an example of how decay products, an Electron and two Neutrinos of different types, are created with '$std :: make_shared$' and then added to a Tau particle's decay particles using '$std :: staticpointercast$', shown in Listing 11.

```
auto tau1 = std::make_unique<Tau>(std::make_shared<FourMomentum>(1776.8, 0, 0, 0));
try {
    // Set a valid set of decay particles
    tau1->setDecayParticles({
        std::static_pointer_cast<Particle>(std::make_shared<Electron>(std::make_shared<
    FourMomentum>(0.511, 0, 0, 0))),
        std::static_pointer_cast<Particle>(std::make_shared<Neutrino>(NeutrinoType::
    ElectronNeutrino, std::make_shared<FourMomentum>(0, 0, 0, 0), true, false)),
        std::static_pointer_cast<Particle>(std::make_shared<Neutrino>(NeutrinoType::
    TauNeutrino, std::make_shared<FourMomentum>(0, 0, 0, 0), false, false))
    });
} catch(const std::exception& e) {
    std::cerr<<"Error setting decay particles: "<<e.what()<<std::endl;
}
```

Listing 11. Management of Tau Particle Decay

3. RESULTS

The initial display presents users with a snapshot of the particle catalogue and shows the total number of particles stored within the catalogue as well as the number of each particle type within their respective sub-containers. The sum of the four-momentum for all particles catalogued is also printed to the screen. Following this summary, the output provides detailed entries for each particle within the container and is depicted in Listing 12.

```
Available particles: all particles within the standard model
Contains: 31 particles.
Number of each particle type:
Leptons: 12
Quarks: 12
Bosons: 7
Total four-momentum of all particles: (E=738112, Px=0, Py=0, Pz=0)

All particle information:
Name=W Boson, Type=Boson, Mass=80360, Charge=+1, Spin=1, FourMomentum=(E=80360, Px=0, Py=0, Pz
    =0)
Name=Z Boson, Type=Boson, Mass=91190, Charge=0, Spin=1, FourMomentum=(E=91190, Px=0, Py=0, Pz
    =0)
Name=Anti-W Boson, Type=Boson, Mass=80360, Charge=-1, Spin=1, FourMomentum=(E=80360, Px=0, Py
    =0, Pz=0)
```

Listing 12. Particle catalogue information output.

The system then allows users to interactively query specific particle information by entering the name of a particle. Upon input, the system outputs detailed attributes of the particle such as type, mass, charge, spin, and other relevant properties. This functionality provides a means to access detailed data, simulating a real catalogue and is shown in Listing 13.

```
Enter a particle name to get its information or 'quit' to exit: electron
Name=Electron, Type=Lepton, Mass=0.511, Charge=-1, Spin=0.5, FourMomentum=(E=0.511, Px=0, Py
    =0, Pz=0), Lepton Number=1, Calorimeter Energies=[0.0630675, 0.1989, 0.249032, 0]

Enter a particle name to get its information or 'quit' to exit: tau
Name=Tau, Type=Lepton, Mass=1776.8, Charge=-1, Spin=0.5, FourMomentum=(E=1776.8, Px=0, Py=0,
    Pz=0), Lepton Number=1, Decay Particles: Muon, Anti-Muon-Neutrino, Tau-Neutrino
```

Listing 13. Demonstration of user querying catalogue information.

Finally, the code illustrates an instance when a set of invalid decay particles are set and non-real four momentum to demonstrate error handling, shown in Listing 14.

```
Invalid decay particles for Tau
Physical inconsistency: Energy cannot be negative and the invariant mass must be equal to the
    rest mass of the particle.

Four-momentum is invalid.
```

Listing 14. Error handling when an invalid decay vector is set as well as a non-physical four-momentum.

4. DISCUSSION

Within the project there are several areas for improvement that could significantly improve usability and computational efficiency. One improvement would be to reformat user outputs to be more intuitive in order to ensure that the user understands the data being presented. This could be done by implementing a graphical user interface (GUI) for a more interactive experience. Additionally, refactoring the code to centralise common functionalities, like conservation checks and particle decay functions at a higher level in the class hierarchy would streamline operations and increase clarity. To further optimise the performance of the code, adopting a structured approach to properties management, such as implementing a map of 'ParticleProperties', would reduce redundancy and accelerate property access during runtime.

5. CONCLUSION

In conclusion the Particle Catalogue Project effectively models Standard Model particles through a well-structured class hierarchy that incorporates complex C++ features such as inheritance and polymorphism to capture the unique characteristics and behaviors of quarks, leptons, and bosons. Key features such as encapsulation, templates, mappings and the extensive use of the Standard Template Library have played pivotal roles in enhancing the project's robustness and scalability. By integrating advanced functionalities like custom containers, exception handling, and lambda functions, this simulation not only facilitates dynamic operations such as type-based searches and cumulative property calculations but also ensures physical consistency in the computations of properties like four-momentum. The simulation has proven to be a valuable scientific and educational tool, demonstrating the power of object-oriented designs in scientific computations. However, there are identified areas for improvement that could improve the project's utility and efficiency. Enhancing the intuitiveness of user outputs and centralising common functionalities like conservation checks and particle decay processes could significantly improve both user experience and code maintainability. Additionally, optimising performance through a more structured management of particle properties, such as implementing a 'ParticleProperties' map, would minimise redundancy and speed up the retrieval of an object's properties during runtime. These improvements will not only streamline operations but also augment the simulation's accuracy and user-friendliness.

[1] C. Sutton, electroweak theory, `https://www.britannica.com/science/electroweak-theory` (2006), accessed: 2024-05-09.
[2] L. M. LLc", `https://lucid.app/lucidchart` (2024), accessed: 2024-05-12.

This document contains 2500 words as calculated by Overleaf.