C++ Report - Particle Catalogue

Christopher Andrews[1]

[1] *Department of Physics and Astronomy*
University of Manchester
Manchester, UK

Github Repository

ABSTRACT

In modern physics research, advanced programming languages like C++ play a crucial role, particularly in computational physics and numerical simulations where performance efficiency is vital. This report presents the development of a particle catalogue system, designed to simulate particle decays whilst strictly adhering to conservation laws, such as energy, momentum and various quantum numbers. The architecture is designed with a 3-tier generational hierarchy with polymorphic class inheritance, for example Particle → Lepton→ Electron. Particles are characterised by their momenta (p_x, p_y, p_z) and other particle specific attributes. The code is capable of outputting detailed properties on a particle specific basis, and simulating the decay sequence of unstable particles, with decay particle properties also outputted.

Contents

1. INTRODUCTION

Advanced programming languages, such as C++, have an ever growing role in modern physics and computational physics. One such field which heavily relies on computational techniques is modern particle physics. This field pushes the boundaries of the Standard Model, analysing and modelling particle data from high-energy physics facilities, such as the LHC (Large Hadron Collider). The complexity of high-energy particle interactions necessitates the development of sophisticated and efficient software capable of dealing with particle behaviours with high accuracy (Sjostrand et al. 2015).

A crucial aspect of particle physics research involves simulating particle interactions and decays, which are fundamental for theoretical development and the potential discovery of new particles. The accurate tracing of energy and momentum in these processes, as well as the analysis of decay products to verify conservation laws, demands advanced and robust programming solutions.

C++ is particularly suited to scientific computing in physics due to its high efficiency, memory management, object orientated features and its extensive scientific library (Stroustrup 2013). All these reasons make C++ ideal for large mathematical computations and many particle simulations.

This project aims to build a comprehensive catalogue that utilises modern C++ features for accurate representation, manipulation and analysis of particle data. This choice of this project was motivated by the opportunity apply and

expand upon advanced C++ features, such as polymorphism, templates and class hierarchy, which are fundamental to modern C++ development in physics.

2. CODE DESIGN AND IMPLEMENTATION

This code implements a class for each individual particle, which also handles their respective antiparticle and are all implemented with rule-of-five deep copy functionality. The classes are grouped (to reduce the number of files) into their base-class file. For example, all the lepton particles are in one file (lepton.h), which also holds the base class Lepton. Similarly for the quarks (quark.h) and bosons (boson.h). The base class Particle, which all particles inherit from, is in its own file (particle.h). This class prints all the general information about each particle, such as type, mass, charge, etc. The specific information about leptons, quarks and bosons are printed within their own classes (such as lepton number in the lepton class). The further specific particle properties are printed within the individual class for which that property is specific to (e.g. colour charge for gluons, and decay products for unstable particles).

Each particle type (leptons, quarks, bosons) have their respective .cpp files which implements their execution, including their particle decay (if unstable). Particle.cpp handles generic functionality, such as the various getters for each particle (e.g. mass, charge, spin). It also handles the conservation checks for decaying particles, such as conservation of energy, momentum (in each component), lepton number, baryon number, charge and colour charge, explained further in §2.2.

The four-momentum is also split into its own class, and file, with its own executing file, fourmom.cpp. The particle_catalogue.h and particle_factory.h files are explained further in §2.1. The class hierarchy can be seen on Figure 1, which shows the 3 generational inheritance structure of the code using the Unified Modelling Language (UML) to visually represent the structure. Virtual functions defined in the 'Particle' class are overridden in derived classes, allowing for polymorphic behaviour where particles can be manipulated in their base classes while exhibiting their unique behaviours. This code was designed so it can be easily editable or modified as needed, with a strict adherence to the splitting of interface and implementation (except in particle_catalogue.h, which utilises templates, thus the implementation is kept within its header file, see §2.1).

The energy is calculated from the momentum as follows:

$$E^2 = m^2 + p_x^2 + p_y^2 + p_z^2 \quad \text{with} \quad \mathbf{p} = (p_x, p_y, p_z) \tag{1}$$

where we have taken the speed of light as 1, $c = 1$.

The enum class ColourCharge allows the allocation of colours/anticolours to particles whilst also providing safety against accidental conversions/comparisons due to them being strongly typed.

2.1. *Specific Functions*

In the particle_catalogue.h file is the ParticleCatalogue class which uses templates, generalising the framework, allowing for more dynamical manipulation. This design choice allows the catalogue to manage any particle type, as long as it adheres to the interface expectations (i.e., having a get_type() method that returns a string). The implementation of a template has several advantages, including:
(i) Flexibility of adding new particle types without altering the code infrastructure
(ii) Ensuring operations on the catalogue, such as adding/removing particles, are type-checked at compile time, reducing runtime errors.

Another implementation in this header file is the use of 'std::unordered_map', which maps string identifiers to vectors of shared pointers to particles, streamlining particle management. It supports efficient lookups, insertions, and deletions, facilitating rapid access and modification of particle data. It also organises particle by type, which is crucial for printing on a particle type basis, and also counting the number of a specific particle type and also provides a clean separation for particle types if operations need to be applied on specific groups of particles. Further benefits of using unordered_map include automation of memory management which reduces the risks of memory leaks. It is implemented into the private part of the class:

```
std::unordered_map<std::string, std::vector<std::shared_ptr<T>>> particles_by_type;
```

The header file "particle_factory.h" is utilised to efficiently add particles to the catalogue using 'try' and will succeed if there are not any non-physical and non-recoverable 'throw' errors, otherwise the error is caught in 'catch' and the particle is not added to the catalogue. For example, if the inputted momentum is too large (for an extreme upper limit of 10^9 MeV/c), the FourMomentum constructor will throw the error, and the particle will be returned as a 'nullptr' in the catch:

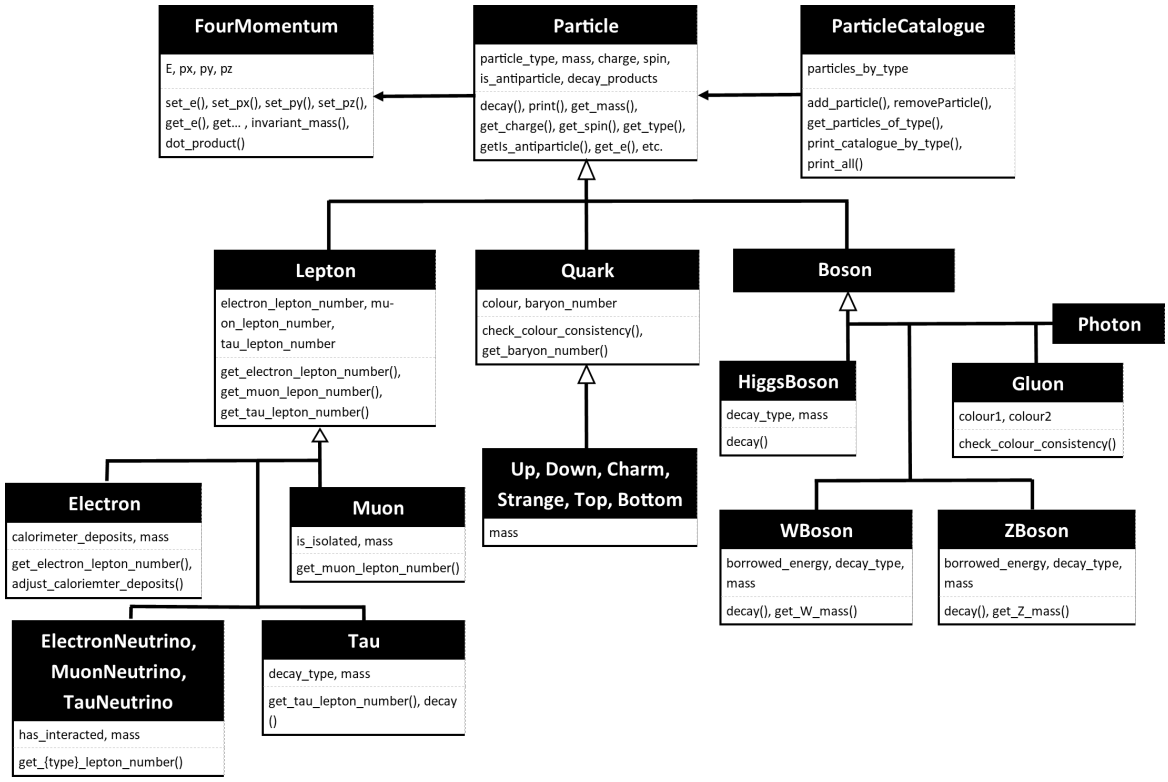

Figure 1: UML class diagram illustrating the inheritance structure of the particle catalogue. The base 'Particle' class is shown at the top, with derived classes 'Lepton', 'Quark', 'Boson'. There are further derived classes from each respective derived classes, such as 'Electron', 'Muon', 'Tau', 'ElectronNeutrino', 'MuonNeutrino' and 'TauNeutrino' from 'Lepton'.

```
FourMomentum::FourMomentum(double E, double px, double py, double pz)
{
  const double max_mom = 1e10;
  if(std::abs(px) > max_mom || std::abs(py) > max_mom || std::abs(pz) > max_mom)
  {
    throw std::invalid_argument("Momentum component is out of the allowed range.");
  }
```

and the function create_add_particle() in particle_factory.h uses the 'try', 'catch' mechanism:

```
    try
    {
      auto particle = std::make_shared<ParticleType>(std::forward<Args>(args)...);
      catalogue.add_particle(particle);
      return particle;
    }
    catch(const std::exception& e)
    {
      std::cerr<<"Error creating particle: "<<e.what()<<std::endl;
      return nullptr;
    }
```

Both of these allows for the concise addition of a particle to the catalogue in the main function in main.cpp, as seen below:

```
auto tau = create_add_particle<Tau>(catalogue, 24, 256, 34, false);
tau->decay();
```

along with the printing of particles by particle-type:

```
catalogue.print_catalogue_by_type("Tau");
```

which will print all the particles in the catalogue of type "Tau", and the decay products associated with that tau particle.

Both gluon colour-anticolour and quarks colour are checked and swapped to adhere to the fact that gluons have to carry one colour and one anticolour, and quarks (antiquarks) have to carry a colour (anticolour). For quarks, a lambda function was utilised to automatically swap the colour to adhere to physical laws:

```
void Quark::check_color_consistency()
{
  // Lambda for swapping colours to their anticolours and vice versa
  auto swap_colour = [this]() -> ColourCharge
  {
    switch(this->colour)
    {
      case ColourCharge::Red: return ColourCharge::AntiRed;
      case ColourCharge::Green: return ColourCharge::AntiGreen;
      case ColourCharge::Blue: return ColourCharge::AntiBlue;
      case ColourCharge::AntiRed: return ColourCharge::Red;
```

The lambda function increases readability; captures 'this' pointer, which allows it to act on the member variable 'colour' directly; and reduces the need to define another private member function, or free function, to achieve the same functionality.

Another lambda function is utilised in the ParticleCatalogue class to keep the code logic local to where its needed (in function remove_particle), negating the need to define a new function elsewhere whilst also increasing efficiency:

```
void remove_particle(const std::string& type, std::shared_ptr<T> particle)
{
  auto& particles = particles_by_type[type];
  auto new_end = std::remove_if(particles.begin(), particles.end(),
                                [&particle](const std::shared_ptr<T>& p) { return p == particle; });
  particles.erase(new_end, particles.end());
}
```

Deep copy functionality is implemented for all particles. For particles which can decay, a boolean flag to copy their decay products (and the decay products of the decay products etc, if there are multi-generational decays) is available when deep copying. For example:

```
auto z_boson_copy_with_decay_products = std::make_shared<ZBoson>(*Z, true);
auto W_minus_copy = std::make_shared<WBoson>(*W_minus1, false);
```

where the 'true' flag indicates the decay products should also be copied, and the 'false' flag indicates they should not. Implementing the decay product copying in the deep-copy required the addition of a clone function in all the derived classes which recursively calls itself to clone multi-generational decays.

2.2. *Decays*

The particles which were considered unstable in this project were: Tau, Higgs, Z-Boson and W±-Boson; as per the instructions. Each particle has a "void decay() override" in their specific classes which overrides the Particle class decay. Each particle's decay is managed in its base class particle-type (e.g. Tau decay is in lepton.cpp, and Z-Boson decay is in boson.cpp). The decay routes are with varying probabilities, accurate to real decays (where appropriate). Using the W-Boson as an example, it has a 33% chance for a leptonic decay, and 66% for a hadronic decay. All allowed (main route) decays (i.e. when $M_W \geq \sum_i M_{\text{decay},i}$) are modelled, with consistency checks on the conservation laws.

Physical conservation checked with "check_baryon_number_conservation", "check_charge_conservation", "check_lepton_number_conservation", "check_invariant_mass", and the energy and momentum iterative assignments and checking of "distribute_energy_momentum" and "check_conservation" respectively. The first 4 in the list compare the before and after states of the decay products, and if conservation is violated, will print an error statement with the type of violation. The "check_invariant_mass" function compares the invariant mass of the decay products to their rest mass. The last 2 in the list are to assign and check the energy and momentum states of the decay products relative to the parent particle.

The distribute_energy_momentum" iterates through many various four-momentum with each step closer towards conservation, until the "check_conservation" has returned a "true", for which the for loop is broken. The "check_conservation" has a 0.1% tolerance for conservation for energy and for each component of the momentum (p_x, p_y, p_z). There is a print statement stating how many iterations it took for each particle to achieve conservation. If conservation over many iterations is not achieved, a statement is printed stating for which particle decay it was not achieved for. Multiple decays such as those seen in the Feynman diagram in Figure 2 are possible in the developed code, with each decay product ensuring conservation of energy and momentum. Due to that the rest mass of the initial parent decaying particle, in this case the Higgs boson, is much larger than the rest mass of the final decay products (e.g. $M_H >> M_\mu$), extreme momenta of the decay products is necessary to achieve the required conservation. For this reason, the many particle decays such as the 3 generational decay of $H \to ZZ \to \tau\bar{\tau}\nu_e\bar{\nu}_e \to \nu_\tau\mu\bar{\nu}_\mu u\bar{d}\bar{\nu}_\tau\nu_e\bar{\nu}_e$ may take more iterations to achieve conservation. The implemented random sampling of this method means that identical decaying particles with the same initial energies and momenta will have decay products with four-momentum unlikely to be the same.

The Higgs Boson can decay to virtual particles of ZZ and W±, which is allowed by allocating "borrowed_energy" to the ZZ or $W^{\pm}$ Bosons (valid under Heisenberg's Uncertainty Principle). This extra energy is associated with the mass of the Z/W bosons, hence the virtual Z/W bosons have invariant mass which is different from their actual mass. The virtual W± Bosons were omitted from the τ decays due to unnecessary complications, and because this decay was not specified in the instructions. Consequently, the τ particle decays directly into three decay products, as illustrated in the right Feynman diagram in Figure 2. This scenario frames the decay of a Tau particle as a three-body problem, rather than a simpler two-body problem. The complexity of managing a three-body decay is reflected in the increased number of iterations required to achieve conservation of momentum and energy among the Tau particles, as detailed in §3.

A code example of the decay of a $W^- \to \tau\bar{\nu}_\tau$ is shown below:

```
auto tau_W = std::make_shared<Tau>(0, 0, 0, charge > 0 ? true : false);
auto tauNeutrino_W = std::make_shared<TauNeutrino>(0, 0, 0, false, charge > 0 ? false : true);
this->add_decay_product(tauNeutrino_W);
this->add_decay_product(tau_W);
distribute_energy_momentum(this->decay_products, this->get_e(), this->get_px(), ...);
tau_W->decay();
```

3. RESULTS

All particles require (p_x, p_y, p_z) inputs. For particles which have antiparticles, a "true/false" boolean parameter on the **last** input is required, with "true" returning their antiparticle. For $W^{\pm}$ bosons however, their charge (±1) should be specified on the first input. Other inputs include colour charge for quarks, and colour-anticolour inputs for gluons, boolean statements for is_isolated for muons, has_interacted for neutrinos and calorimeter deposits for electrons (which is automatically changed to match the electron's energy). Example inputs are shown below:

```
auto electron = create_add_particle<Electron>(catalogue, 1.0, 2.0, 3.0,
    std::vector<double>{0.1, 0.2, 0.15, 0.05}, false);
auto anti_up_quark = create_add_particle<UpQuark>(catalogue, 1.0, 2.46, 75, ColourCharge::AntiRed, true);
auto gluon = create_add_particle<Gluon>(catalogue, ColourCharge::Green, ColourCharge::AntiGreen, 4, 7, 2);
auto W_minus = create_add_particle<WBoson>(catalogue, -1, 10, 76, 82);
W_minus->decay()
```

Each particle is printed with its own particle specific properties, and the virtual W/Z bosons are printed with the mass energy that was borrowed to remain compliant to conservation laws. A decay similar to that in Figure 2, with $H \to ZZ \to d\bar{d}\tau\bar{\tau} \to d\bar{d}e\bar{\nu}_e\nu_\tau u\bar{d}\bar{\nu}_\tau$, is shown below:

```
Energy and momentum conservation for HiggsBoson decay achieved in 14 iterations.
Energy and momentum conservation for ZBoson decay achieved in 169 iterations.
Energy and momentum conservation for ZBoson decay achieved in 316 iterations.
Energy and momentum conservation for Tau decay achieved in 4025 iterations.
Warning: Total energy deposited in calorimeter does not match the Electron's energy (0.511 instead of 10393.3). Adjusting.
Energy and momentum conservation for AntiTau decay achieved in 6521 iterations.
Type: HiggsBoson
  Mass: 125110.00 MeV/c^2
  Invariant mass: 125110.00 MeV/c^2
  Charge: 0.00
```

```
  Spin: 0.00
  Four-Momentum: (125128.77, 2004.00, 334.00, 754.00) MeV/c
Decay Type: Virtual ZZ
Decay Products:
Virtual ZBoson with borrowed energy: 28632.60 MeV
Type: ZBoson
  Mass: 91187.60 MeV/c^2
  Invariant mass: 62555.00 MeV/c^2
  Charge: 0.00
  Spin: 1.00
  Four-Momentum: (62774.38, 6556.31, 22769.89, 9993.35) MeV/c
Decay Type: Hadronic
Decay Products:
Type: DownQuark
  Mass: 4.70 MeV/c^2
  Invariant mass: 4.70 MeV/c^2
  Charge: -0.33
  Spin: 0.50
  Four-Momentum: (31387.19, 26713.26, 11904.19, -11395.08) MeV/c
  Colour Charge: Red
  Baron number: 0.33
  Antiparticle: No
Type: AntiDownQuark
  Mass: 4.70 MeV/c^2
  Invariant mass: 4.70 MeV/c^2
  Charge: 0.33
  Spin: 0.50
  Four-Momentum: (31334.18, -20156.95, 10865.69, 21388.42) MeV/c
  Colour Charge: AntiRed
  Baron number: -0.33
  Antiparticle: Yes
Virtual ZBoson with borrowed energy: 28632.60 MeV
Type: ZBoson
  Mass: 91187.60 MeV/c^2
  Invariant mass: 62555.00 MeV/c^2
  Charge: 0.00
  Spin: 1.00
  Four-Momentum: (62359.93, -4552.31, -22435.89, -9239.35) MeV/c
Decay Type: Leptonic
Decay Products:
Type: Tau
  Mass: 1776.86 MeV/c^2
  Invariant mass: 1776.86 MeV/c^2
  Charge: -1.00
  Spin: 0.50
  Four-Momentum: (31179.97, -3964.18, -100.65, -30875.69) MeV/c
  Electron Lepton Number: 0
  Muon Lepton Number: 0
  Tau Lepton Number: 1
  Antiparticle: No
Decay Type: Leptonic
Decay Products:
Type: Electron
  Mass: 0.51 MeV/c^2
  Invariant mass: 0.51 MeV/c^2
  Charge: -1.00
  Spin: 0.50
  Four-Momentum: (10393.32, -1083.09, 303.47, -10332.28) MeV/c
  Electron Lepton Number: 1
  Muon Lepton Number: 0
  Tau Lepton Number: 0
  Antiparticle: No
  Calorimeter Deposits: 10393.32 0.00 0.00 0.00
Type: AntiElectronNeutrino
  Mass: 0.00 MeV/c^2
  Invariant mass: 0.00 MeV/c^2
  Charge: 0.00
  Spin: 0.50
```

```
  Four-Momentum: (6928.88, -479.98, 226.50, -6908.52) MeV/c
  Electron Lepton Number: -1
  Muon Lepton Number: 0
  Tau Lepton Number: 0
  Antiparticle: Yes
  Has Interacted: No
Type: TauNeutrino
  Mass: 0.00 MeV/c^2
  Invariant mass: 0.00 MeV/c^2
  Charge: 0.00
  Spin: 0.50
  Four-Momentum: (13859.05, -2401.11, -630.62, -13634.89) MeV/c
  Electron Lepton Number: 0
  Muon Lepton Number: 0
  Tau Lepton Number: 1
  Antiparticle: No
  Has Interacted: No
Type: AntiTau
  Mass: 1776.86 MeV/c^2
  Invariant mass: 1776.86 MeV/c^2
  Charge: 1.00
  Spin: 0.50
  Four-Momentum: (31152.80, -588.13, -22335.24, 21636.34) MeV/c
  Electron Lepton Number: 0
  Muon Lepton Number: 0
  Tau Lepton Number: -1
  Antiparticle: Yes
Decay Type: Hadronic
Decay Products:
Type: UpQuark
  Mass: 2.20 MeV/c^2
  Invariant mass: 2.20 MeV/c^2
  Charge: 0.67
  Spin: 0.50
  Four-Momentum: (10384.27, 116.38, -7713.80, 6951.03) MeV/c
  Colour Charge: Red
  Baron number: 0.33
  Antiparticle: No
Type: AntiDownQuark
  Mass: 4.70 MeV/c^2
  Invariant mass: 4.70 MeV/c^2
  Charge: 0.33
  Spin: 0.50
  Four-Momentum: (6685.06, -2.31, -4335.21, 5088.80) MeV/c
  Colour Charge: AntiRed
  Baron number: -0.33
  Antiparticle: Yes
Type: AntiTauNeutrino
  Mass: 0.00 MeV/c^2
  Invariant mass: 0.00 MeV/c^2
  Charge: 0.00
  Spin: 0.50
  Four-Momentum: (14085.19, -702.20, -10286.23, 9596.51) MeV/c
  Electron Lepton Number: 0
  Muon Lepton Number: 0
  Tau Lepton Number: -1
  Antiparticle: Yes
  Has Interacted: No
Total number of decay products for HiggsBoson (including subsequent decays): 12
```

3.1. *User Inputs*

The user has various input prompts, guided through by easy-to-follow statements, these are as follows:

1. The first prompt is asking whether the user would like to print the whole catalogue [y/n] ([yes/no] also functions as an input). After all the information about all the particles is printed using the print_all() function in ParticleCatalogue class, there is another print statement:

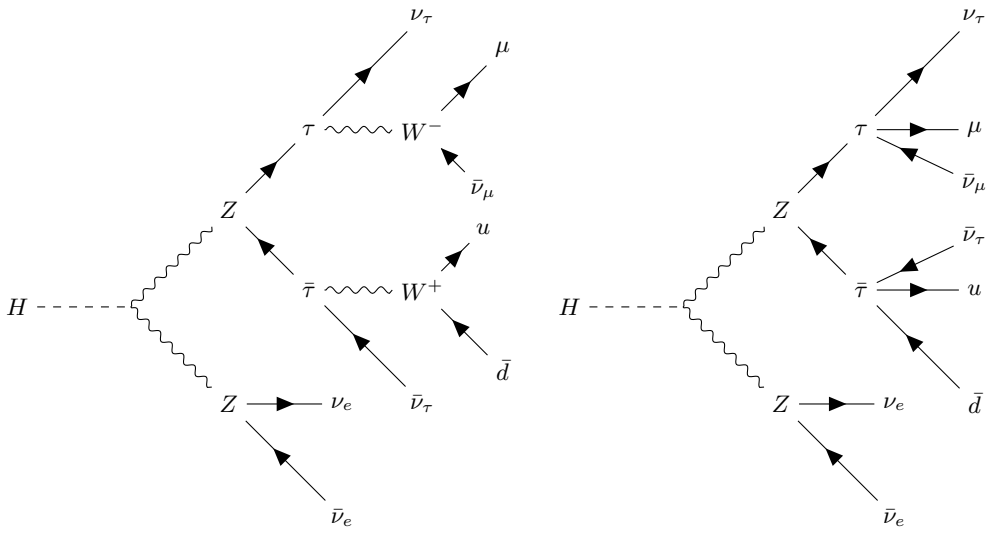

Figure 2: Feynman diagram depicting the decay of a Higgs boson (H) into two virtual Z bosons, which subsequently decay into a pair of τ and $\bar{\tau}$, and further decay processes involving W^- and W^+ bosons. The Feynman equation on the right is the decay process implemented into this project, which omits the virtual $W^\pm$ from the tau decay.

```
Total number of base particles printed: 3
Total number of decay particles printed: 16
```

which tells the user the number of base particles (particles directly added to the catalogue by the user and particles which haven't been created from a decay) that have been printed, and also the total number of decay particles which have been printed, which also includes all subsequent decays.

There is also another function which calculates the sum of all the four-momentum of all the base particles, decay particles, as well as the invariant mass associated with their four-momenta:

```
Total Four-Momentum of all base particles in the catalogue: (1024654.69, 7207.00, 2797.76, 3223.00) MeV/c
Total Invariant Mass of all base particles in the catalogue: 1024620.46 MeV/c^2
Total Four-Momentum of all decay particles in the catalogue: (882902.74, 7373.00, 3373.00, 4358.00) MeV/c
Total Invariant Mass of all decay particles in the catalogue: 882854.76 MeV/c^2
```

2. The second user input prompt is the ability to choose which specific particle-type they want printed, eg an input of 'W-' will print all of the associated W^- bosons in the catalogue. This is not case sensitive, so an input of 'higgsboson' or 'hiGgsBosOn' will print the HiggsBoson. An example of W^- is shown below:

```
Printing 3 particles of type W- and its decay products:
Type: W-
  Mass: 80377.00 MeV/c^2
  Invariant mass: 80377.00 MeV/c^2
  .
  .
  .

Type: W-
  Mass: 80377.00 MeV/c^2
  Invariant mass: 80377.00 MeV/c^2
  .
  .
  .


Type: W-
  .
  .
  .
```

```
Number of particles of type W-: 3
```

This is repeated until the user inputs 'n' or 'no'.

3. The third input is to demonstrate the deep-copy functionality, where the user is prompted [y/n], which prints an electron, Z (with original decay products), and W- (without original decay products).

4. The final input [y/n] is to save the print_all() and sum_all() ouputs to a .txt file. This text file will be named 'particle_catalogue_output_{date}_{time}'.

4. CONCLUSION

This report has presented a comprehensive particle catalogue system developed using modern C++ programming techniques, which simulates the decay of various known particles and antiparticles while strictly adhering to conservation laws. The implementation successfully demonstrates the use of advanced C++ features, including polymorphism, templates, and lambda functions, in creating a highly modular and scalable system.

The architecture of the system, designed with a 3-tier inheritance structure facilitates clear organisational logic and enhances the maintainability of the code. This structure allows for the easy addition of new particle types and decay interactions without extensive modifications to the existing structure. This project was deigned such that it can be heavily adapted for further improvements, and also for the addition of any more particles, or any changes/developments in the Standard Model.

The system could be improved by expanding the range of particle interactions, incorporating further decay chains and enhancing the efficiency of the four-momentum assignment algorithm. Expanding the number of conservation checks could also be utilised to ensure rigorous compliance with physical laws. The function which iterates through energy and momentum could be improved to incorporate a more efficient process, and to more accurately represent real particle decays. Further work could be carried out to introduce detectors to the simulated decay chains with varying detecting probabilities.

Total Words: 2496 (from overleaf wordcount)

REFERENCES

Sjostrand, T., Ask, S., Christiansen, J., et al. 2015, Computer Physics Communications, 191, 159, doi: 10.1016/j.cpc.2015.01.024

Stroustrup, B. 2013, The C++ Programming Language, 4th edn. (Addison-Wesley Professional)