Particle Catalogue Report
PHYS30762 Project Report

Euan Baldwin 10818421
Department of Physics and Astronomy
University of Manchester

(Dated: May 11, 2024)

This project aims to develop a particle catalogue to store data on Standard Model particles. The solution is designed around a polymorphic class architecture, with an abstract particle base class, derived classes for different particle species, and individual classes for sufficiently distinct particle types. The catalogue integrates custom container classes that offer user-implemented methods for counting particles, operating on four-momentum, and retrieving sub-containers. The design supports specialised particle properties, such as calorimeter layers for electrons, isolation variables for muons, and decay vectors. Advanced features such as templates, maps, lambda functions and STL containers are utilised to ensure that the final implementation offers an accurate, scalable, and efficient way to catalogue particle data.

1. INTRODUCTION

The Standard Model of particle physics is the most widely accepted and well-tested theory that describes three of the four known fundamental forces — the electromagnetic, weak, and strong interactions. It classifies all known elementary particles into four distinct categories, specifically quarks, leptons, guage bosons and scalar bosons. These categories are shown in Figure 1.

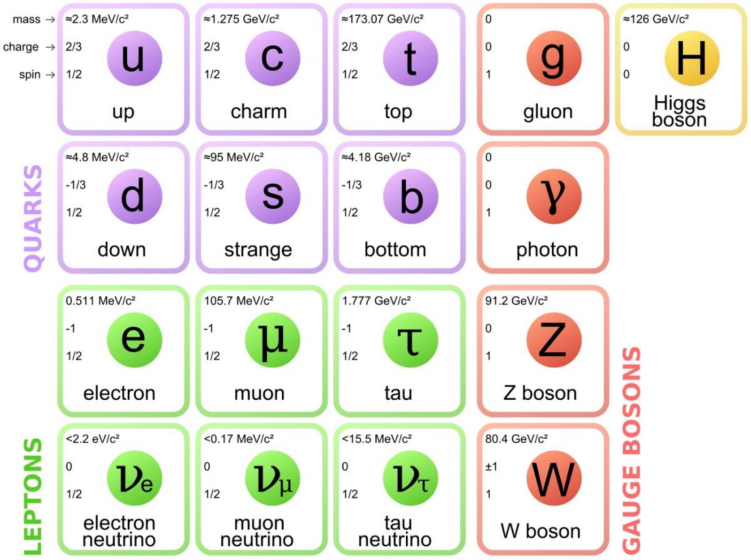

FIG. 1: Detailed list of the fundamental particles described by the Standard Model of particle physics [1].

The model's development spans several decades, beginning in the mid-20th century. The discovery of the Higgs boson in 2012 at CERN [2] provided experimental confirmation of the final missing piece of the model, reinforcing the theoretical framework established in the 1960s and 70s. The quark model brought order to the particle zoo by proposing that particles like protons and neutrons are composed of fundamental quark subunits. Likewise, the unification of electromagnetic and weak forces into the electroweak theory by Glashow, Weinberg, and Salam marked another milestone, earning them the 1979 Nobel Prize in physics [3].

The need for precise experimentation and modelling drives the requirement for an accurate and structured digital catalogue to store and simulate particles detected in experiments. This catalogue is crucial for encapsulating the classifications and interactions presented in the Standard Model. This project aims to create a robust system that utilises object-oriented programming principles to organise particles and their antiparticles within a hierarchical class structure. This approach not only allows for efficient particle storage but also provides a flexible and scalable solution for managing and simulating particle physics research.

2. CODE DESIGN AND IMPLEMENTATION

2.1. Particle Hierarchy and Class Structure

As shown in Figure 2, the codebase for the particle catalogue project is structured to support a detailed simulation and management of various Standard Model particles. The hierarchy arranges particle families into

classes, namely `Boson`, `Quark`, and `Lepton`, which all inherit from the common abstract base class, `Particle`. Each specific type of particle extends these classes, inheriting common properties such as mass, charge, spin, antiparticle status and decay handling, while also implementing or overriding specific behaviours, such as `print data()`.

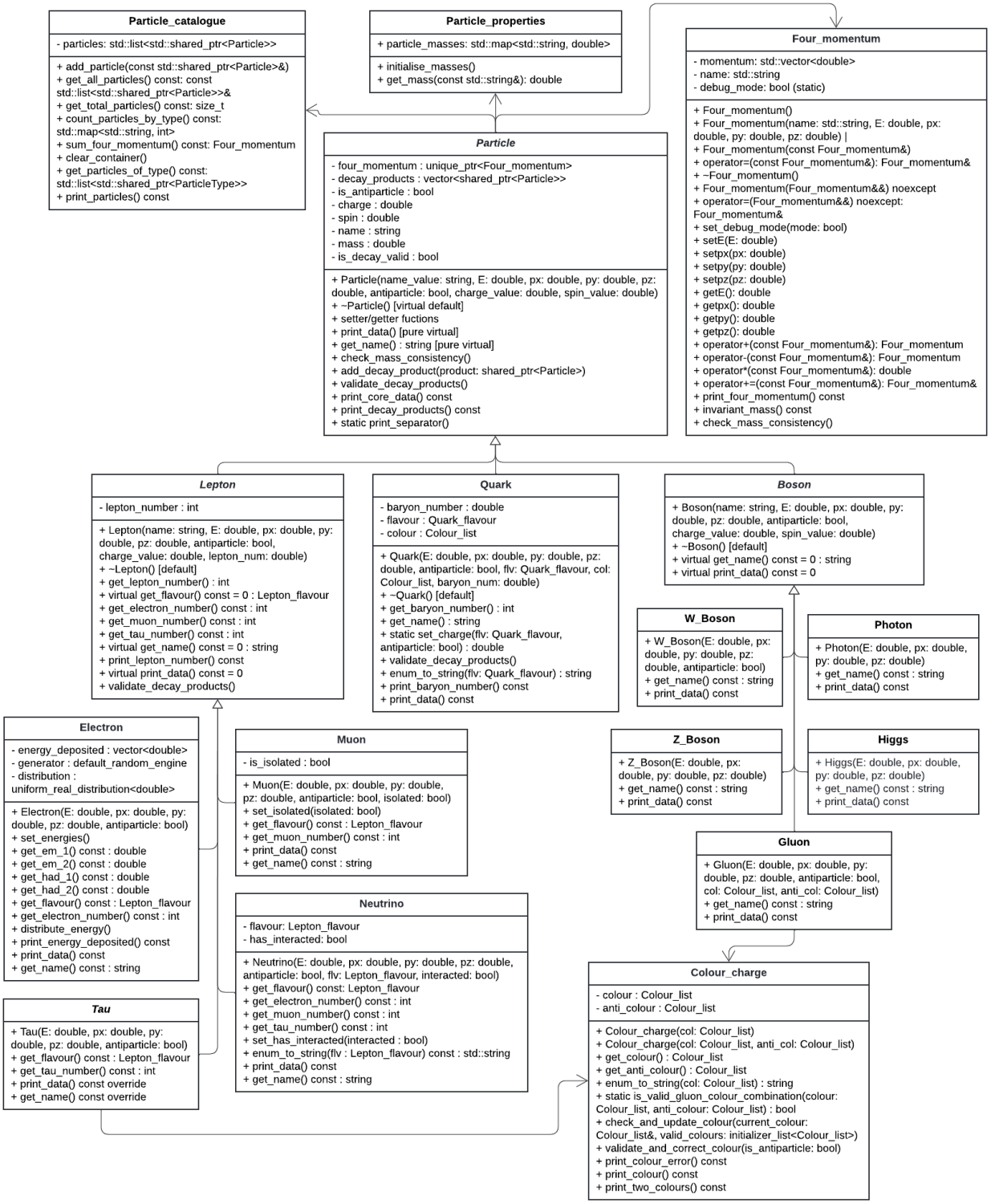

FIG. 2: UML diagram showing the inheritance chain of particle species, particle utilities and individual flavours. Made using [4].

A core part of the `Particle` class is the handling of decay validation, shown in Listing 1. A boolean flag for decay validity is defined, and decays with invalid charge conservation set this flag to the invalid, or false, state. To ease debugging a print statement is also included for invalid decays, allowing the user to easily identify which decay was invalid and why.

```
// Validate that the decay products conserve charge
void validate_decay_products()
{
  double sum_charge = 0;
  for(const auto& product : decay_products)
    sum_charge += product->get_charge();

  // If charges don't match, mark decay as invalid
  if(sum_charge != this->get_charge())
  {
    std::cerr<<"\033[0;31mCharge inconsistency detected in "<<get_name()<<" decay.\033[0m"<<
    std::endl;
    is_decay_valid = false;  // Mark the decay as invalid
  }
  else
    is_decay_valid = true;  // Decay is valid if charge is consistent
}
```

Listing 1: Method in the `Particle` class to validate the charge conservation in decays.

The classes for particle families are implemented as abstract classes (except for the `Quark` class) from the `Particle` class. The `Lepton` class, for example, includes specific properties and methods related to leptons, such as lepton number validation in decay chains. The individual particle classes are then derived from the particle family classes. This is shown in the `Electron` class, which further specialises `Lepton`, adding or customising methods to handle its unique attributes, namely the printing of electron data and the energy the electron deposits as it passes through the four layers of a calorimeter. Other specialised lepton flavour-specific attributes include isolation status for muons and interaction status for neutrinos.

Quarks and Bosons follow a similar hierarchical structure, with the `Quark` class extending `Particle` and adding quark-specific properties, such as flavour and colour and printing implementation. The `Quark` class also implements the handling of baryon number. As quarks were not determined to be sufficiently distinct from one another the `Quark` class did not require abstraction and the implementation of individual quark flavour classes was deemed inefficient. The abstract `Boson` class and its subclasses (`W Boson`, `Photon`, `Z Boson`, `Higgs`, and `Gluon`) override the virtual particle printing function to print their unique values of spin, charge and decay products. Gluons in particular also inherit a colour and an anticolour charge from the `Colour charge` class.

2.2. Utility Classes

Supporting the particle classes are various utility classes. One example is the `Four momentum` class, which handles a particle's four-momentum, essential for physics calculations. This class includes methods for performing four-vector operations, including addition, subtraction, and multiplication. An implementation of how the inner product of four-momenta is calculated is shown in Listing 2. Additionally, the class includes checks to verify that the particle's energy is greater than zero and that its four-momentum is consistent with its invariant mass.

```
// Four-vector multiplication (inner product)
double operator*(const Four_momentum& other) const
{
  return getE() * other.getE() - (getpx() * other.getpx() + getpy() * other.getpy() + getpz()
    * other.getpz());
}
```

Listing 2: Implementation of four-vector inner porduct calculation in the `Four momentum` class.

The `Particle properties` class functions as a centralised repository for particle's invariant masses, which are mapped to particle names using `std::map`. This design allows for quick lookup and retrieval of invariant masses, used to validate a particle's four-momenta, in a way that maintains the readability of the code. As shown in Listing 3, the implementation for returning particle masses utilises a try/catch block to allow for graceful degradation and error-specific handling logic, preventing a valid four-momentum operation (which returns a nameless particle) from throwing unnecessary errors.

```
// Search for the particle's mass by its name
double get_mass(const std::string& particle_name)
{
  try
  {
    // Check if the particle name is empty (valid use case to return four-momentum operations)
    if(particle_name.empty())
      // Return NaN as a representation of undefined mass
      return std::numeric_limits<double>::quiet_NaN();

    auto it = particle_masses.find(particle_name); // Attempt to find particle name from map

    if(it != particle_masses.end()) // If found, return the corresponding mass
```

```
      return it->second;

    // If not found, throw an exception
    throw std::invalid_argument("Particle name '"+particle_name+"' not found in particle masses.");
  }
  catch(const std::exception& e) // Print error message and return a default value
  {
    std::cerr<<"Error: "<<e.what()<<std::endl;
    return 0;
  }
}
```

Listing 3: Implementation of try/catch to return the invariant mass of a particle in the `Particle properties` class.

The `Colour charge` class handles and validates particle colour charges using the `Colour list` enumerator, shown in Listing 4. This enumerator defines all possible colour charges for particles and their corresponding anti-colours and by using enumerators, the class ensures that each particle's colour and anti-colour are represented with distinct, readable values. The class constructors use these enumerators to initialise either a single colour state (used for quarks) or a colour and anti-colour state (used for gluons). The enumerator values are translated into readable strings through the implementation of an enumerator-to-string switch statement. Additionally, as shown in Listing 5, validation methods utilising lambda functions (which make the code cleaner and the localised scope intent more transparent) check and correct the enumerated colour properties for particles and antiparticles, setting invalid colours to the "Unknown" status and producing helpful debugging outputs for the user.

```
// Enum class representing possible colour charges
enum class Colour_list { Red, Green, Blue, Antired, Antigreen, Antiblue, Unknown };
```

Listing 4: Enumerator for colour and anti-colour charge in the `Colour charge` class.

```
// Checks and updates a colour if it is not in the list of valid colours
void Colour_charge::check_and_update_colour(Colour_list &current_colour, const std::
    initializer_list<Colour_list> &valid_colours)
{
  // Use a lambda to check if the current_colour is in valid_colours
  auto is_valid_colour = [&valid_colours](Colour_list colour)
  {
    return std::find(valid_colours.begin(), valid_colours.end(), colour) != valid_colours.end
    ();
  };

  // Update colour if it's not valid
  if(!is_valid_colour(current_colour))
  {
    if(current_colour != Colour_list::Unknown)
    {
      current_colour = Colour_list::Unknown;
      print_colour_error();
    }
  }
}
```

Listing 5: Implementation of colour validitity checking using a lambda function in the `Colour charge` class, ensuring that colours are valid from the enumerated options shown in Listing 4.

The core of the codebase is structured around the `Particle catalogue` class, which acts as the central repository and manager for all particles. It uses advanced data structures, for example, `std::list` and `std::map`, to store and manage shared pointers to particle objects, enabling efficient memory management and data retrieval. Specifically, it allows adding particles individually or in bulk, accessing particles by their properties, and performing aggregate operations such as counting particles by type or summing their momenta. The class also provides functionality to print a list of particles, enhancing its utility for debugging and reporting.

The implementation of returning particles of only a certain type, shown in Listing 6, is done using a template function, `get particles of type()`, that retrieves all particles of the specified type from the particle container. By using a template the function remains flexible and can adapt to different particle types without requiring separate implementations. The template parameter, `Particle type`, is inferred when the function is called, enabling the function to identify the intended particle type at compile time. The function iterates over the particle container and attempts to cast each particle to `Particle type` using a dynamic cast to a `std::shared ptr`. If the cast succeeds, the particle is added to a sublist that stores only particles of this particular type, hence leveraging polymorphism and ensuring that only compatible particles are collected.

```cpp
// Retrieves all particles of a specific type
template<typename Particle_type>
std::list<std::shared_ptr<Particle_type>> get_particles_of_type() const
{
  // Create a list to store the particles of the specific type
  std::list<std::shared_ptr<Particle_type>> sublist;
  for(const auto& particle : particles)
  {
    // Try to dynamically cast to the specific type
    auto casted_ptr = std::dynamic_pointer_cast<Particle_type>(particle);
    if(casted_ptr)
      // If cast is successful add particle to the sublist
      sublist.push_back(casted_ptr);
  }
  return sublist; // Return the sublist containing all particles of the specific type
}
```

Listing 6: Template used to retrieve particles of a specific type from the `Particle container` class

2.3. Main Application Flow

The main application logic is in `main`, where instances of various particle types are created and handled. This file serves as the project's central hub, bringing together particle classes and utility functions to form a unified simulation environment.

The `Particle container` object is initialised to hold and manage the particles. Various types of particles are created using `std::make shared`, which creates these particles dynamically and returns a shared pointer to them. Further details on how to initialise particles can be found in the project's `README` file. After adding particles to the container, the program prints out their properties using `print particles()`. It can then clear the container for new operations with `clear container()`.

```cpp
// Hadronic tau decay
auto tau_hadronic = std::make_shared<Tau>(1.98e+00, -3.96e-01, 4.94e-01, 5.94e-01, false);
container.add_particle(tau_hadronic);
tau_hadronic->add_decay_product(std::make_shared<Quark>(4.80e-03, 0, 0, 0, false,
    Quark_flavour::Down, Colour_list::Red));
tau_hadronic->add_decay_product(std::make_shared<Quark>(2.30e-03, 0, 0, 0, true, Quark_flavour
    ::Up, Colour_list::Antired));
tau_hadronic->add_decay_product(std::make_shared<Neutrino>(1.55e-02, 0, 0, 0, false,
    Lepton_flavour::Tau, true));
tau_hadronic->validate_decay_products();
```

Listing 7: Example of how to add decay products to a particle in the container. The decay added is a hadronic tau decay into a down quark, an up anti-quark and a tau neutrino. The decay output is shown in Listing 9.

Decaying particles, such as the tau lepton, are added along with their decay products. As shown in Listing 7, these products are added to the parent particle using `add decay product()`, followed by validation with `validate decay products()` to ensure the decay adheres to physical laws (for example conservation of lepton number and charge conservation). The program also intentionally adds several particles that represent invalid states or decay processes to demonstrate input validation. The ability to compute operations on particles' four-momenta using overloaded operators is also demonstrated.

3. RESULTS

The particle physics simulation program demonstrates various functionalities for managing particle properties, interactions, and validations. The output from executing the program demonstrates numerous operations, including four-momentum addition, particle property validation, and printing particle states and interactions.

Initially, the program adds an invalid quark (an up quark with a colour of antired) to the container. This triggers two types of errors: an invariant mass inconsistency, where the calculated mass significantly differs from the known value, causing the four-momentum to be scaled, and an invalid colour charge, resulting in the colour being set to "Unknown". This example, shown in Listing 8, demonstrates the program's ability to detect and handle inconsistencies in particle properties. Following this, an invalid decay process for an Anti-Tau particle is simulated, showing the program's ability to handle invalid particle decays.

```
-----------------------------------------------------------------
Invariant mass inconsistency for Up.
  Known mass: 2.30e-03 GeV
  Calculated mass: 9.27e+00 GeV
Invalid colour(s). Setting to Unknown.
Up Quark
Colour:             Unknown
Spin:               0.50
Charge:             0.67e
```

```
Baryon number:      0.33
Four-momentum:      [2.48e-03, 2.48e-04, 4.96e-04, 7.44e-04] GeV
---------------------------------------------------------------------
```

Listing 8: Example of an invalid particle created with a four-momentum inconsistent with its invariant mass. The quark was also incorrectly assigned an anti-colour so its colour charge has been set to the default value of "Unknown".

The container's functionality is showcased through operations such as clearing its contents, printing particle details (shown in Listing 9), and returning the total number of particles and their types. The simulation verifies that 38 particles have been added, including valid particles, antiparticles, and decays. Listing 10 shows the total four-momentum of all particles, demonstrating the container's container's ability to aggregate properties across its contents.

```
---------------------------------------------------------------------
Tau
Spin:               0.50
Charge:             -1.00e
Lepton number:      1
Four-momentum:      [1.98e+00, -3.96e-01, 4.94e-01, 5.94e-01] GeV
Decay Products:     Down Quark, Up Anti-Quark, Tau Neutrino
---------------------------------------------------------------------
```

Listing 9: Example of a particle output to the terminal. The tau is decaying via the hadronic channel, as detailed in Listing 7.

```
---------------------------------------------------------------------
Total Particles: 38
Type: Bottom              Count: 2
Type: Charm               Count: 2
Type: Down                Count: 2
Type: Electron            Count: 2
Type: Electron Neutrino   Count: 2
Type: Gluon               Count: 2
Type: Higgs Boson         Count: 4
Type: Muon                Count: 2
Type: Muon Neutrino       Count: 2
Type: Photon              Count: 1
Type: Strange             Count: 2
Type: Tau                 Count: 3
Type: Tau Neutrino        Count: 2
Type: Top                 Count: 2
Type: Up                  Count: 2
Type: W Boson             Count: 3
Type: Z Boson             Count: 3
Total Four-momentum:      [1.95e+03, -5.59e+01, 4.03e+02, 2.81e+02] GeV
---------------------------------------------------------------------
```

Listing 10: Output showing total particle count, individual particle counts and total four momenta of particles in the container.

Lastly, as shown in 11, specific particle operations such as calculating the dot product of four-momenta and adding or subtracting four-momenta of particles illustrate the program's utility in performing particle physics calculations. These operations are crucial for studies in particle dynamics and interactions within physics simulations.

```
---------------------------------------------------------------------
Dot product of Four-momentum:   5.84e-06 GeV^2
Addition of Four-momentum:      [4.79e-03, 3.07e-04, -4.27e-04, -6.16e-04] GeV
Subtraction of Four-momentum:   [1.70e-04, 1.89e-04, -5.65e-04, -8.72e-04] GeV
---------------------------------------------------------------------
```

Listing 11: Output showing four-momentum operation examples (dot product, addition and subtraction) using the values of the first two particles in the catalogue.

Overall, the program output indicates a sophisticated system capable of simulating, validating, and managing complex particle interactions, reflecting real-world scenarios in particle physics research.

4. DISCUSSION

Several enhancements to the codebase could be made in order to extend its features and improve its usability. A notable improvement would involve expanding the particle catalogue beyond the Standard Model to include,

for example, the bosons predicted by the Grand Unified Theory [5]. Developing a Graphical User Interface (GUI) to include features such as decay chain visualisation could also improve usability.

While the existing error-handling mechanisms are robust, expanding these checks to other parts of the code could improve the program's fault tolerance, thereby providing more detailed error messages that include context to help developers diagnose issues quickly. Additionally, implementing unit tests for each class and method would improve the program's reliability and ease future refactoring.

Furthermore, enhancing data validation routines across all particle types would ensure consistency. For example, implementing stricter checks on particle attributes before including them in the catalogue would reduce the risk of invalid data. Finally, although the use of smart pointers is effective, implementing custom allocators or improving object reuse strategies might reduce memory overheads when dealing with large datasets. This would be particularly valuable for real-world particle physics applications, which often require highly optimised computing to deal with high volumes of data.

5. CONCLUSION

In conclusion, the development of this particle catalogue system demonstrates the advantages of an object-oriented approach to managing and simulating Standard Model particles. The hierarchical class structure, with an abstract `Particle` base class and specialised subclasses, provides a versatile and efficient framework. The polymorphic design ensures that different particle types retain their unique properties while sharing a consistent interface, thus enabling user-friendly data management and operations. The addition of utility classes, such as `Four momentum`, `Particle properties`, and `Colour charge`, further add to the catalogue's functionality. Despite the strengths of the code, improvements could be considered for future iterations. These include extending the catalogue beyond the standard model, developing a GUI and implementing additional decay validation checks. Overall, this project uses templates, maps, lambda functions, and STL containers to build a scalable, modular platform for cataloguing particle data, offering a valuable tool for advancing our understanding of the Standard Model in particle physics research.

---

[1] R. L. Workman and Others (Particle Data Group), Review of Particle Physics, PTEP **2022**, 083C01 (2022).
[2] G. Aad, T. Abajyan, B. Abbott, J. Abdallah, S. A. Khalek, A. A. Abdelalim, R. Aben, B. Abi, M. Abolins, O. AbouZeid, *et al.*, Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc, Physics Letters B **716**, 1 (2012).
[3] Nobel Prize Outreach AB, The nobel prize in physics 1979, `https://www.nobelprize.org/prizes/physics/1979/summary/` (2024), accessed: 2024-05-10.
[4] Lucid Software Inc, Lucidchart, `https://lucid.app` (2008), accessed: 2024-05-09.
[5] G. Altarelli, The standard electroweak theory and beyond (1998), arXiv:hep-ph/9811456 [hep-ph].

According to Overleaf's word count, this document contains 2076 words.