ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP010

AtomicAndPhysicalConstants.jl: A FAST JULIA PACKAGE TO ACCESS PARTICLE PROPERTIES AND FUNDAMENTAL CONSTANTS*

L. Li†, D. Sagan, CLASSE, Cornell University, Ithaca, NY, USA
D. Abell, University of Maryland, College Park, College Park, MD, USA
A. Coxe, Virginia Commonwealth University, Richmond, VA, USA

Abstract

To support constants lookup for the SciBmad project, we introduced `AtomicAndPhysicalConstants.jl`, a Julia package that provides physical constants, subatomic particle properties, and atomic isotope data. It aggregates datasets from NIST (CODATA) and the Particle Data Group, offering a unified interface. The package supports configurable unit systems and data types, and integrates with the Julia package `Unitful.jl`. The package emphasizes speed and ease of use, making it well-suited for high-performance simulations and physics-driven modeling.

INTRODUCTION

In accelerator and beam physics, the Bmad [1] ecosystem – developed at Cornell University's Laboratory for Elementary Particle Physics – has long served as a key framework for simulating charged-particle dynamics.

Currently, a new Julia-based simulation environment called SciBmad [2] is being developed to take advantage of Julia's strengths in performance, robust support for parallelism, and native multiple dispatch.

As part of the SciBmad ecosystem, the Julia package `AtomicAndPhysicalConstants.jl` was developed to fill the need for providing fast and stable access to physical constants. It provides 16 common constants, 14 subatomic particle, all NIST isotopes data, and supports 3 sets of unit systems and 3 data types to fit various needs.

Key Features

- **Configurable units and types** – Specify unit systems (`MKS`, `CGS`, or `ACCELERATOR`) and numeric representations (`Float64`, `Unitful` [3], or `DynamicQuantities` [4]) through a single macro call.
- **Data integration** – Pulls constants from CODATA (Committee on Data of the International Science Council) [5], NIST (National Institute of Standards and Technology) [6], and PDG (Particle Data Group) [7], with support for historical datasets.

Comparison with Existing Packages

Several Julia packages also provide access to physical constants. One of the most widely used package is `PhysicalConstants.jl` [8], which provides reliable static constants with uncertainty support. `AtomicAndPhysicalConstants.jl` offers a direct interface for constants access and an option to customize units and variable types. Another key differentiator is the `Species` type, which encapsulates particle attributes, such as mass and charge, for simpler access.

CUSTOMIZING UNITS AND TYPES

A key feature of `AtomicAndPhysicalConstants.jl` is the macro `@APCdef`, which enables users to configure global behaviors for the package, including the choice of unit system, variable type, and variable scoping. This allows users to work with constants within a consistent framework of units and variable types under a wide range of physical modeling needs, such as SciBmad and `DifferentialEquations.jl` [9].

However, custom unit and variable type usually lead to type instability and slow run-time performance. A typical approach is using functions to get constants and species properties and convert them to proper units and type at run time. This conversion not only introduces extra computation, but also makes the function type-unstable. If the return type is not decided at compile time, the compiler could not optimize the program, causing more memory and time usage.

We solve this by using a macro to generate code in the name space. Once the units and types are decided, constants and getter functions with a fixed unit and type will be written. Although this approach slows program at the macro call, it avoids converting units and specifies a certain type, making the program fast at run time.

`@APCdef` Usage

The `@APCdef` macro serves as the primary configuration interface and must be invoked once during package initialization (A second invocation raises an error). It assigns units and data types for all constants, including fundamental physical constants as well as `Species` mass and charge values (see Section 3). Upon invocation, the macro generates these constants and attaches appropriate getter functions—*e.g.* `massof()` and `chargeof()`—which return species properties.

The macro supports multiple unit systems through the `unitsystem` keyword, allowing users to choose from predefined conventions: `ACCELERATOR` (a unit system that contains commonly used units in accelerator physics), `MKS` (SI standard units), and `CGS` (centimeter-gram-seconds). Each system sets base units for mass, length, time, energy, and

* Work supported by Brookhaven Science Associates, LLC under Contract Nos. DE-SC0012704 and DE-SC0018008 with the U.S. Department of Energy.
† ll963@cornell.edu

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP010

charge accordingly. In addition, users may opt to provide a custom five-tuple of `Unitful`-typed units to define a domain-specific unit system.

The numerical type of constants can be selected via the `unittype` argument, supporting `Float64`, `Unitful.Quantity`, or `DynamicQuantity.Quantity`. This enables both unit-aware scientific calculations and dimension-free simulations.

By default, all constants are encapsulated in a named tuple (the user can set the name of this tuple by configuring the `name` keyword argument, otherwise it defaults to `APC`), enabling organized and type-stable access. However, if `tupleflag = false` is specified, constants are instead defined as global variables in the current namespace, allowing for direct usage without prefixing.

*Support for Unit-Aware Numerical Types with **Unitful.jl** and **DynamicQuantities.jl***

To support dimensionally consistent computations, `AtomicAndPhysicalConstants.jl` integrates with two major unit-aware-type packages in Julia: `Unitful.jl` and `DynamicQuantities.jl`

Unitful.jl `Unitful.jl` provides a rich system for representing quantities with units. This package is used internally to manage the unit of the physics quantities.

DynamicQuantities.jl `DynamicQuantities.jl` offers a dynamic, symbolic unit system with simpler types. Our package converts `Unitful` typed variables to `DynamicQuantities` typed variables.

SPECIES

To facilitate a convenient access to particle properties, `AtomicAndPhysicalConstants.jl` provides a `Species` type that encapsulates physical properties of particles and atomic isotopes. A `Species` object serves as a structured identifier for mass, charge, and other physical characteristics, allowing unit-aware access to species properties.

Each `Species` instance contains the following fields: `name`, `mass`, `charge`, `spin`, `moment`, `iso`. There are 5 kinds of species: `ATOM`, `HADRON`, `LEPTON`, `PHOTON`, and `NULL`.

Constructing Species

The `Species` constructor provides a standardized interface for constructing particles and atomic isotopes. Species are instantiated by passing an OpenPMD [10] standard particle name to the constructor `Species(::String)`.

Null Species A null species acts as an uninitialized value. It is used in cases where a species field must be present but is not yet assigned.

Subatomic Particles Elementary particles are constructed using their exact name. Antiparticles are designated using the "anti-" prefix. The constructor is case-sensitive and expects exact matches to supported names.The package supports 14 different subatomic particles.

Atomic Species Atomic species may be defined using a compact string that encodes the isotope and charge state. The constructor accepts:

- An atomic symbol (e.g., `"H"`, `"C"`, `"Al"`)
- An optional isotope mass number, either as a prefix (e.g., `"13C"`) or using Unicode superscript notation (e.g., `"`^{13}`C"`)
- An optional electric charge, specified using symbols such as `+`, `++`, `+n`, or their negative counterparts (Unicode superscripts are accepted).

If no isotope is specified, the mass is taken to be the mean weighted by isotope abundance (per NIST). Similarly, if no charge is provided, the species is assumed to be electrically neutral.

Accessing Species Properties

Once a `Species` instance is constructed, its physical characteristics — such as mass, charge, and quantum numbers — can be accessed in units and types defined by the `@APCdef` macro.

For any given species, fundamental properties such as rest mass and electric charge are available through standard getter functions. These include mass m, charge q, atomic number Z, and spin S.

In addition to these direct accessors, the package provides functions for computing derived physical quantities. supported physical quantities include: The gyromagnetic ratio g is defined as

$$g = \frac{2m\mu}{Sq} ,$$

where m is the mass, μ the magnetic moment, S the spin, and q the charge.

For leptonic species, the gyromagnetic anomaly,

$$g_a = \frac{g-2}{2} ,$$

where g is the gyromagnetic ratio of the species.

For baryonic species, a nucleon-normalized gyromagnetic factor is also provided,

$$g_n = \frac{gZm_p}{m} ,$$

where Z is the charge of the species, m_p the proton mass, and m the species mass.

These functions allow users to easily extract and manipulate fundamental and derived particle properties.

CONSTANTS SOURCES

It is essential to have accurate and traceable physical constants for quantitative physics research. `AtomicAndPhysicalConstants.jl` draws data from recognized sources and enables users to explicitly select the release year of reference data.

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP010

The primary source of fundamental physical constants is CODATA. By default, the package loads the most recent CODATA release (currently CODATA 2022), but earlier release such as CODATA 2018 remain accessible through versioned submodules that users can invoke when starting the package. This modular structure provides a straightforward way to select datasets.

Isotope data are sourced from NIST. NIST does not maintain history of their isotope datasets. Therefore, all isotope information reflects the most recent release.

Pion properties (for both π^0 and $\pi^{\pm}$) are obtained from PDG and downloaded directly from the PDG API. This ensures pion data reflect the latest experimental values.

QUALITY CONTROL

Testing

`AtomicAndPhysicalConstants.jl` is equipped with an extensive and modular test suite. This framework performs rigorous validation across `@APCdef` configurations, and `Species` constructor parsing rules. All tests are executed on GitHub Actions as part of a continuous integration (CI) workflow.

Benchmarking

`AtomicAndPhysicalConstants.jl` includes a benchmarking suite to maximize performance. The primary objective is to ensure type stability (the type of a variable is decided at compile time) and improve speed.

Two tools were used to analyze the performance of the package:

- **BenchmarkTools.jl** [11] was used to assess memory allocations and execution time of `@APCdef`, species construction, and getter functions (e.g., `massof()`, `chargeof()`).
- **JET.jl** [12] was used to detect and eliminate dynamic dispatches and type instabilities.

All exported functions are verified as type-stable using `JET.jl`, and benchmarks confirm minimal allocations during constant access. Core functions run time including `@APCdef` and `Species()` are reduced to less than 1 microsecond.

These characteristics make the package suitable for high-performance applications in particle tracking, where runtime speed are critical.

FUTURE WORK

- **Expanded data coverage.** Upcoming releases will include more physics constants.
- **Performance tuning.** We are improving runtime preformance for `@APCdef` and `Species` initialization.
- **Community-driven evolution.** We will update based on Github issues and bug reports. CODATA, NIST, and PDG updates will be integrated to keep the database current.

CONCLUSION

`AtomicAndPhysicalConstants.jl` delivers a type-stable, unit-aware interface to physical constants. By drawing from authoritative datasets and allowing users to pin historical versions, the package guarantees numerical traceability. The macro configuration supports multiple unit systems and variable type through integration with packages like `Unitful.jl` and `DynamicQuantities.jl`.The `Species` interface simplifies working with subatomic particles and isotopes. The library forms a foundation for SciBmad and any other physics application that demands accurate physical constants.

REFERENCES

[1] D. Sagan, *The Bmad Manual*, https://www.classe.cornell.edu/bmad/manual.html

[2] Bmad-Sim Developers, *SciBmad: Julia Interface to Bmad*, https://github.com/bmad-sim/

[3] PainterQubits, *Unitful.jl*, https://github.com/PainterQubits/Unitful.jl

[4] SymbolicML Contributors, *DynamicQuantities.jl*, https://github.com/SymbolicML/DynamicQuantities.jl

[5] E. Tiesinga, P. Mohr, D. Newell, and B. Taylor, *The 2022 CODATA Recommended Values of the Fundamental Physical Constants, ver. 9.0*, National Institute of Standards and Technology, Gaithersburg, MD. doi:10.48550/arXiv.2409.03787

[6] J. S. Coursey, D. J. Schwab, J. J. Tsai, and R. A. Dragoset, *Atomic Weights and Isotopic Compositions, ver. 4.1*. National Institute of Standards and Technology, Gaithersburg, MD. doi:10.1002/9781119894490.app2

[7] Particle Data Group. *PDG Live: Review of Particle Physics (2024) — 2025 Update*, Lawrence Berkeley National Laboratory, May 2025. https://pdg.lbl.gov

[8] JuliaPhysics, *PhysicalConstants.jl*, https://juliaphysics.github.io/PhysicalConstants.jl/stable/

[9] C. Rackauckas and Q. Nie, *DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia*. https://github.com/SciML/DifferentialEquations.jl

[10] A. Huebl *et al*. "openPMD 1.1.0: A Meta Data Standard for Particle and Mesh Based Data." Zenodo, 2015. doi:0.5281/zenodo.591699

[11] Julia Lab, *JET.jl*, https://github.com/aviatesk/JET.jl

[12] Julia Packages, *BenchmarkTools.jl*, https://juliapackages.com/p/benchmarktools