

## Julia in HEP

*Graeme Andrew Stewart*<sup>1,\*</sup>, *Alexander Moreno Briceño*<sup>2</sup>, *Philippe Gras*<sup>3</sup>, *Benedikt Hegner*<sup>1</sup>, *Uwe Hernandez Acosta*<sup>4,5</sup>, *Tamas Gal*<sup>6</sup>, *Jerry Ling*<sup>7</sup>, *Pere Mato*<sup>1</sup>, *Mikhail Mikhasenko*<sup>8</sup>, *Oliver Schulz*<sup>9</sup>, and *Sam Skipsey*<sup>10</sup>

<sup>1</sup>EP-SFT, CERN, Geneva, Switzerland

<sup>2</sup>Universidad Antonio Nariño, Ibagué, Colombia

<sup>3</sup>IRFU-CEA, Université Paris-Saclay, Gif-sur-Yvette, France

<sup>4</sup>Center for Advanced Systems Understanding, Görlitz, Germany

<sup>5</sup>Helmholtz-Zentrum Dresden-Rossendorf, Dresden, Germany

<sup>6</sup>Erlangen Centre for Astroparticle Physics, Friedrich-Alexander-Universität, Erlangen, Germany

<sup>7</sup>Laboratory for Particle Physics and Cosmology, Harvard University, Cambridge MA, USA

<sup>8</sup>Ruhr Universität Bochum, Bochum, Germany

<sup>9</sup>Max-Planck-Institut für Physik, Munich, Germany

<sup>10</sup>School of Physics & Astronomy, University of Glasgow, Glasgow, United Kingdom, G12 8QQ

**Abstract.** Julia is a mature general-purpose programming language, with a large ecosystem of libraries and more than 12000 third-party packages, specifically targeting scientific computing. Julia runs on x86, aarch64 and PowerPC architectures, and on all major GPU platforms. As a language, Julia is as dynamic, interactive, and accessible as Python with NumPy, but achieves run-time performance on par with C/C++. In this paper, we describe the state of adoption of Julia in HEP, where momentum has been gathering over a number of years. HEP-oriented Julia packages can already read HEP's major file formats, including TTree and RNTuple. Interfaces to some of HEP's major software packages, such as Geant4, are available too. Jet reconstruction algorithms in Julia show excellent performance. A number of full HEP analyses have been performed in Julia.

We show how, as the support for HEP has matured, developments have benefited from Julia's core design choices, which makes reuse from and integration with other packages easy. In particular, libraries developed outside HEP for plotting, statistics, fitting, and scientific machine learning are extremely useful.

We believe that the powerful combination of flexibility and speed, the wide selection of scientific programming tools, and support for all modern programming paradigms and tools, make Julia the ideal choice for a future language in HEP.

## 1 Programming Languages in High-Energy Physics

### 1.1 HEP Needs

High-energy physics (HEP) is a large field, consisting of tens of thousands of researchers, almost all of whom will need to interact with software and contribute to software projects

---

\*e-mail: [graeme.andrew.stewart@cern.ch](mailto:graeme.andrew.stewart@cern.ch)

during their careers [1]. It is also one of the biggest, if not the biggest, generators of scientific datasets today, with exabytes of storage used by the LHC experiments [2]. This data is processed by a huge corpus of software, estimated to be many tens of millions of lines in C++ and Python, as well as Fortran, CUDA and others [3].

This brings a challenge for HEP software. From the point of view of *code efficiency* we require fast execution, high throughput, and scalability at large heterogeneous computer centres and across distributed infrastructures. Considering *human efficiency* we would like a low barrier to entry for newcomers, the ability to prototype code rapidly, a broad ecosystem of well maintained packages, and excellent tooling for developers. These features are needed to make software able to deal efficiently with huge datasets, as well as accessible to a large group of developers. Stability and code preservation are also needed.

## 1.2 From Fortran to the C++/Python Era

In response to changing technology and needs the programming languages that are dominant in HEP have evolved over time. From [4] we can identify three major shifts.

**From assembler to Fortran** c. 1960 As computers developed from early specialised behemoths improved programming languages became available. For technical computing Fortran was the most effective language and HEP quickly adopted it as it brought a much improved syntax, as well as hardware portability.

**From Fortan to C++** c. 2000 Although Fortran had many advantages, HEP had to develop language extensions to introduce missing concepts, such as more advanced data structures [5]. A language which offered native object orientation was an attractive choice. In addition, the gap afforded by the end of the LEP accelerator and the construction of the LHC gave the field the time for a major language shift.

**The rise of Python** c. 2010 Python earned a well deserved reputation as an excellent language for programming efficiency, and also gained ground through being the de facto interface to many machine learning libraries. It has become widely used in HEP as a complement to C++.

The current situation for HEP is that C++ and Python are now both widely used, with each bringing specific advantages, as well as drawbacks. Good C++ excels at runtime efficiency, but is a difficult language to learn, far less to master, as well as suffering from memory safety issues and being difficult to compose. Python is expressive, much easier to work with, is safer with memory and composes better (via duck typing). However, it is very slow compared to C++, so not suitable for high throughput computing.

As discussed at length in [6], using two languages is not ideal: it requires additional expertise, necessitates reimplementing of code for performance, and reduces code reusability.

## 2 Julia

### 2.1 Julia's Motivations

The Julia programming language was announced in 2012 listing a series of ambitious goals for the language, and representing a view into the core developers' mindset, formalised in a later papers [7, 8].

Julia provides a syntax as productive as Python, especially for numerical work, whilst leveraging JAOT<sup>1</sup> compilation to provide speed similar to compiled, statically typed languages like C/C++ and Rust. It utilises type inference to allow coding in a “gradually typed” (§2.3.1), generic programming style, although the language will always track types for performance behind the scenes, and types can be specified explicitly if required. Like MatLab and Fortran, Julia’s native operations and type system support arrays as first-class entities, of any dimension, allowing array-oriented code to be productively generated and efficiently executed, with operations naturally “broadcast” element- or dimension-wise. This allows Julia to also be effective for writing, e.g., linear algebra heavy code. In addition, influence from the R community provides a wealth of statistical packages, and an R-flavoured approach to plots and visualisation. As with most languages of the 21st century, Julia is a fully open-source language, with its entire codebase freely available (and almost all of it written in Julia itself).

## 2.2 Julia in Practice

Julia’s syntax is familiar to programmers conversant with programming performant code in Python with NumPy, except that whitespace is not relevant (code blocks end with end).

For example, the listing in Figure 1 shows some toy code to generate a grayscale image of the famous Mandelbröt set.

```
1 using Images
2
3 function mandel(z; maxiter=80)
4     c = z
5     for n = 1:maxiter
6         if abs2(z) > 4
7             return n-1
8         end
9         z = z^2 + c
10    end
11    maxiter
12 end
13
14 set = [ mandel(complex(r,i)) for i=-1.:01:1., r=-2.0:01:0.5 ]
15 img = Gray.(set ./ 80)
```

Figure 1: Julia implementation of the Mandelbröt set

Here we demonstrate importing of packages (line 1); function declaration with implicit types, positional and named parameters (line 3); an explicit loop (line 5); branch constructs (line 6); the implicit Julia use of return values (line 11); implicit loops via “comprehensions” (line 14) to generate a value for every point in an implicitly defined array; and, finally, transparent broadcasting of operations over that array (line 15, where both the function call to Gray, and the division operation are distributed over the whole array). Unlike Python, explicit loops are optimised efficiently, and are no slower than comprehensions or “functional-style” iterators (which are also supported by Julia).

<sup>1</sup>Just-Ahead-Of-Time: meaning compilation to machine code happens, like C++, but just before a code block is executed, unlike C++. Julia can also cache these results between runs.

`juliaup` allows seamless management of multiple Julia releases on the same machine, including tracking particular patch releases, and choosing the system default.

The integrated package management in Julia, via the `Pkg` library, tracks and maintains the dependency graph of a Julia project. State is entirely stored within two human-readable files – `Project.toml` (the direct dependencies of the project) and `Manifest.toml` (the exact resulting environment, including secondary dependencies and exact versions). One can easily reproduce the exact environment used by a codebase as long as these files are provided.

The Julia Package “General Registry” indexes packages and their releases, and (like Rust and Javascript) relies on a public index hosted on GitHub. A help environment allows interactive help on any keyword or symbol known a the Read-Evaluate-Print-Loop (REPL) environment (which would be very familiar to Python users); it is trivial to add documentation to a function or type by simply prepending its definition with a triple-quoted docstring.

Leveraging the fact that Julia is JAOT-compiled, and that this allows its entire standard library to be written in idiomatic Julia, the REPL also provides a series of powerful macro utilities for inspecting the byte- and machine-code generated for a given expression (`@code_lowered`, `@code_native`) and for locating and displaying the source code for any function in the current namespace, including from the standard library (`@less`). Profiling of code in the REPL is similarly directly supported via macros (`@benchmark`), which provide detailed performance sampling. Extended introspection and profiling tools are also available in optional packages, such as `About.jl`, providing information on memory layout of datatypes and thread safety of functions.

A well-supported VSCode extension is available for the language, which also supports the standard Language Server Protocol allowing it to support development in other editors. This includes the usual benefits of code completion, including using Copilot and other AI extensions, Unicode completion, linting and highlighting. There is a built-in interactive debugger, as well as innovative tools to break into a REPL in the middle of code execution (`Infiltrator.jl`).

Julia is now popular enough to have many online training materials, an active support community on Slack and Discourse, and many postings on Stackoverflow.

Finally, Julia is one of the founding languages supported by Jupyter - being the “Ju” in the portmanteau – and also provides other native notebook implementations such as `Pluto.jl`.

## 2.3 Key Design Features for Performance

Whilst there are many features of the Julia language design which contribute to its performance, and productivity, we highlight two particular ones here.

### 2.3.1 Type System

Julia’s type system is an expressive, but simple, tree of sets, where only leaves of the tree can be instantiated as concrete types. The higher levels of abstract types provide a means for expressing categories of types which are all supported by an operation, all the way up to the `Any` type, which contains all other types, and is the default type for function parameters if none are specified. For example, using Julia’s notation for “is a subtype of”, `<:`, the concrete type `Float64` is in the following hierarchy:

**Float64** `<:` **AbstractFloat** `<:` **Real** `<:` **Number** `<:` **Any**

All higher levels are abstract, and thus cannot be used as the type for a binding. Other sibling subtypes of `Real` include `Float32`, `Int64` and `BigInt`.

Abstract types *can* be used as the type of an element of a container type, resulting in a container with boxed types, where each element can be any concrete subtype of the element; and similarly for function parameters, where they constrain that parameter to taking the relevant concrete subtypes as values.

Together, this allows complex type expressions such as

**AbstractArray{T,2}**

This represents a function argument that takes any Array-like type, of 2 dimensions, with an arbitrary element type T (which could be further constrained by annotation). As the subtypes of **AbstractArray** include performance specialised cases like diagonal and sparse arrays, as well as accelerated cases, like GPU arrays (computed on an accelerator, not the CPU), then all of these cases are naturally handled by the same function that supports this one type.

### 2.3.2 Multiple dispatch

The type system above allows for efficient specialisation of functions by the JAOT backend, based on the types they are called with. Unusually, Julia exposes this to the user, allowing specialisations of functions to be dispatched on the type of any (or all) parameters it is called with, not merely the first. This *multiple dispatch* allows effective provision of efficient special cases for functions to be provided, powered entirely by the type system.

Extending the nomenclature for the *single dispatch* used by class-based object orientation, Julia considers the variants of a function distinguished by their type signatures to be separate “methods”.

The runtime resolution of function dispatch also allows seamless composition of functionality between packages, without the need for special case code in any of the involved.

For example, in the code listing from Figure 2 below, the **Plots** and **DifferentialEquations** packages do not know about **Measurements** – and yet the types provided by **Measurements** can seamlessly extend their functionality (as well as providing their own extensions of mathematical operations), resulting in the plot combining all the features of the packages seamlessly.

```
using Measurements, Plots
using DifferentialEquations

g = 9.79 ± 0.02;
L = 1.00 ± 0.01;
u0 = [0.0 ± 0.0, π \ 60 ± 0.01]
tspan = (0.0 ± 0, 1.0 ± 0)
function p(du,u,p,t)
    du[1]=u[2]
    du[2]=-(g/L)*u[1]
end
prob = ODEProblem(p, u0, tspan)
sol = solve(prob, Tsit5());
plot(sol.t, getindex.(sol.u,2))
```

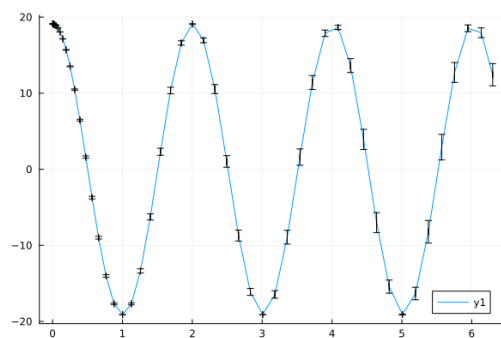


Figure 2: Plot with **Measurements** of numerical solution using **DifferentialEquations**, composed via multiple dispatch and the Julia type system.

## 3 Julia for Scientific Computing

### 3.1 GPU Programming

Julia's JAOT compilation model makes it ideal for running on GPUs. Julia supports GPU programming for specific backends such as `CUDA.jl` [9] for NVIDIA, `AMDGPU.jl` [10] for AMD, `Metal.jl` for M-series Mac devices, and `oneAPI.jl` for Intel devices. They require a minimal amount of development and support writing kernels with granular control.

Array based calculations are trivial to execute on the GPU. In the listing in Figure 3 an array is copied to the GPU and then executes a trivial operation, but mainly shows that there is basically no boilerplate required to access GPU computations.

```
using CUDA
```

```
a = CuArray([1,2,3,4])  
a * 2
```

Figure 3: Array based calculations for the `CUDA.jl` backend - the code for other backends is essentially the same.

It is not always possible to write an algorithm and run it using high-level array abstractions. Instead one has to write a GPU kernel – Kernel programming is close to the native toolkits. Writing and maintaining specific backend code is undesirable, from a portability point of view. Therefore Julia also supports a generic backend, `KernelAbstractions.jl` [11, 12]. This allows developers to separate the mathematical logic of their codes from the specific GPU backend on which it will run, resulting in high code portability. This feature is used heavily in Julia codes from `Flux.jl` [13], for machine learning; to `Oceananigans.jl` [14], a fluid dynamics code.

Memory management on the GPU is mostly automated, including garbage collection of unreachable objects, but more fine grained steering is available.

### 3.2 Julia HPC codes

Beyond the single-node practices exemplified above, large-scale applications often require high-performance computing (HPC) technologies. Here Julia has established itself as a strong contender alongside well-known HPC-capable programming languages such as C, C++, and Fortran. Benchmarks evaluating intra- and inter-node communication on CPUs have shown that Julia introduces negligible overhead compared to native C implementations [15].

A particularly noteworthy achievement is the near-zero-loss integration of MPI through `MPI.jl` [16], enabling efficient parallel communication. Furthermore, when it comes to computational performance, Julia has demonstrated its ability to keep up with traditional HPC technologies. For instance, in single-CPU-node scenarios, Julia's performance matches that of vendor-optimised libraries [17]. Similarly, performance portability studies in multi-node CPU/GPU benchmarks confirm Julia's competitiveness across architectures [18, 19].

Beyond controlled benchmarking environments, Julia has also proven its capabilities in real-world HPC applications. One prominent example is the `Celeste.jl` project [20, 21], where a peak performance of 1.54 PFLOPs was shown, demonstrating Julia's scalability, efficiency in large-scale computations and, consequently, membership of the petaflop club.

More recently, Julia has been advancing multi-GPU and multi-CPU applications in large-scale simulations. Notable examples include `FastIce.jl` for high-resolution flow simulations and various geocomputing applications leveraging HPC-related packages like `ParallelStencils.jl` and `ImplicitGlobalGrid.jl`. These packages enable high-performance stencil computations and efficient distributed memory parallelism, reinforcing Julia's role as a modern HPC language.

## 4 Julia in HEP

### 4.1 Challenges

High-energy physics faces significant challenges in software efficiency and scaling for the years to come, particularly driven by the physics programme of the high-luminosity LHC [3]. Data volumes will rise sharply and data complexity will also increase [22–24]. Therefore software efficiency and scaling are paramount. As we have seen in § 3.2, the Julia language can achieve very efficient execution on HPC clusters of CPUs and GPU accelerated nodes. It has also been shown that Julia artefacts can be effectively distributed on CVMFS, so that efficient running on, e.g., WLCG sites, would be possible.

HEP has millions of lines of legacy code, with which any new language must interoperate. In this respect Julia is in a very strong position. For calling into C or Fortran, Julia offers a simple `@ccall` macro, which is a direct, no overhead, no boilerplate interface to libraries compiled from these languages. There are many existing examples of calls to these foreign libraries being used in Julia, e.g., at a low level Julia itself uses the OpenBLAS library [25], written in C and modern Fortran.

C++ code is more difficult to interface to (a 'feature' of C++ itself), and the smoothest way to achieve it is to write a small wrapper in C++ to interface to Julia, making use of the Julia interface package `CxxWrap.jl`. This process can be automated with the helper program `WrapIt`, which can generate wrapper code automatically from C++ headers. With this, wrapping very large libraries becomes much easier, as shown below in § 4.4.

The other major desiderata for a programming language is that it is efficient for programmers – and for HEP this must cover the spectrum from novice coders to experienced software engineers. As illustrated in §2.2 Julia's development ecosystem is optimised for efficient code development, leading to widespread adoption in science [26].

Of course, a language having suitable *general* features, does not automatically mean that everything needed by researchers in a particular field have the tools and packages that they need. In the next sections we outline the growing ecosystem of packages specifically developed for HEP that make it possible to be productive with Julia from day 1 of coding.

### 4.2 HEP Data Formats

Most HEP experimental data is stored in ROOT [27] format, making it crucial for Julia to have the ability to read and write data in this format. Several options are available for achieving this functionality. One possibility is to create Julia bindings to the ROOT C++ package. However, this approach presents technical challenges, as it requires cross-compilation to generate binary artifacts, a process that is not yet fully supported by ROOT. Despite this, progress is being made with the `ROOT.jl` [28] package, which aims to address these challenges providing a direct interface to ROOT's C++ code.

An alternative is the `UnROOT.jl` [29] package, which offers a pure Julia implementation of the ROOT I/O format, independent of ROOT or Python, thus implying a lightweight solution, with far fewer dependencies and a light. This package supports transparent reading of

both TTree and the newer RNTuple storage formats. While writing functionality is still under development, `UnROOT.jl` provides fast, memory-efficient, and lazy data access, reading only the necessary parts of a file on demand.

The `UnROOT.jl` package can also be used to read EDM4hep [30] event data, in combination with the `EDM4hep.jl` package. The EDM4hep event data model aims to establish a standard for storing and exchanging event data in HEP experiments. The `EDM4hep.jl` package provides a pure Julia interface to this model, generating user-friendly Julia types that map to the EDM4hep data structure. These types are compatible with both arrays of structures and structures of arrays when reading data files typically produced by C++ programs. This structured approach makes data access more intuitive compared to reading flat n-tuples, facilitating the development of analysis code in a more natural and efficient way.

### 4.3 Event Generators

The numerical calculation of amplitudes for hard scattering processes and Monte Carlo event generation plays a fundamental role in high-energy physics analysis workflows, with a long history ([31]). Consequently, integrating various aspects of physics models with specialised numerical algorithms in a high-performance computing environment is recognised as one major challenge in HEP software development for future HEP experiments ([32–34]) and beyond.

With the open-source framework `QuantumElectrodynamics.jl` [35], the first steps in exploring how Julia can address these challenges are taken. Specifically, the framework facilitates the numerical calculation of scattering amplitudes and the implementation of Monte Carlo event generation within the domain of perturbative and strong-field quantum electrodynamics (cf. [36]). The overall structure of the framework is unified through interfaces defined in `QEDbase.jl`, which provides standardised representations for fundamental mathematical objects: four-momenta, bi-spinors, and phase space points. Additionally, these interfaces extend to the configuration entities: scattering processes, computational models, and phase space layouts. Further interfaces are provided for computable quantities, such as differential cross-sections, and various samplers for Monte Carlo event generation. Here, Julia’s multiple dispatch mechanism proves particularly powerful, allowing different implementations to be seamlessly integrated without explicitly specifying types within the interface definitions. Moreover, the domain-specific language facilitated by these interfaces – covering elements such as processes, models, and phase space layouts – combined with multiple dispatch enables the straightforward incorporation of analytical formulas whenever they are available. This typically reduces to implementing a single method for a specific function signature.

Beyond its interface definitions, `QuantumElectrodynamics.jl` provides concrete implementations for all major components: `QEDcore.jl` handles the fundamental mathematical structures, `QEDprocesses.jl` computes differential cross sections, and `QEDevents.jl` offers different samplers for event generation. The `QEDFeynmanDiagrams.jl` package, as part of `QEDprocesses.jl`, facilitates the calculation of scattering amplitudes for arbitrary QED processes by leveraging Julia’s metaprogramming and code generation capabilities. Built on top of `ComputableDAGs.jl`, the generated code can be directly analysed and manipulated within Julia itself, even without leaving the session. This enables meta-optimizations based on domain-specific knowledge, such as recognising patterns in Feynman diagrams and exploiting mathematical properties like distributivity.

Finally, by leveraging multiple dispatch and various array abstractions from the Julia ecosystem (§3.1) the generated code can be seamlessly compiled and executed on GPUs as well as CPUs, without requiring any modifications to its underlying structure. While some framework components are still under development and will be further extended to address

broader challenges in high-energy physics, the initial structures demonstrate great potential. The modular design, combined with Julia's capabilities, has the potential to contribute to future developments in numerical calculations and event generation within HEP.

#### 4.4 Simulation

Detector simulation is a crucial component of every HEP experiment, playing a key role both during the design and conception of the detector and later in data analysis. The most widely used toolkit for this purpose is Geant4 [37], a C++-based framework with over 2 million lines of code.

Given its complexity and extensive adoption, a complete rewrite of Geant4 in a new language is highly impractical. Instead, this presents an opportunity to explore Julia's interoperability with other languages. One particular challenge arises from Geant4's callback-based user interface, which relies on C++ virtual methods invoked at specific points during particle transport. Application developers must implement these callbacks to configure and control the simulation and extract relevant data. However, integrating this mechanism in Julia is more complex than in other languages, as Julia does not natively support virtual methods.

The `Geant4.jl` package has been developed to provide a Julia interface to Geant4 [38]. It leverages `CxxWrap.jl`, a package that enables calling C++ functions and types from Julia. Similar to Python's static bindings, invoking C++ code from Julia requires explicit wrapper definitions for each method exposed to Julia. However, given Geant4's large and complex codebase, manually writing and maintaining these wrappers is not a viable approach, especially for making it more sustainable with future toolkit updates. To address this, we use `WrapIt`, which automates wrapper generation by utilising the Clang library to parse C++ header files and extract class declarations. This automation significantly reduces development effort and ensures long-term maintainability of the interface.

Integrating Geant4 with Julia allows researchers to take advantage of Julia's high-level programming capabilities and performance (e.g., to plot results or code user actions), while retaining the full functionality and efficiency of the underlying Geant4 toolkit. This integration also provided an opportunity to rethink and streamline the interface, making it more intuitive and user-friendly. The package interface ensures that application developers can concentrate on the essential aspects of their simulations while minimising configuration overhead. Boilerplate code and C++ idiosyncrasies are hidden, allowing for a cleaner, more concise approach to defining simulations. The performance of the `Geant4.jl` package is comparable to that of the native C++ Geant4 toolkit, demonstrating the feasibility of using Julia for HEP detector simulation.

#### 4.5 Reconstruction

Reconstruction of HEP data is a core task for the field, and a huge amount of software, often rather detector specific, is dedicated to that task. Even for very detector specific code there are some early demonstrations that Julia would be suitable, and can be competitive with modern optimised C++ for a large experiment on multiple different GPU backends, e.g., for the CMS pixel detector [39]. For smaller experiments, there are already end-to-end solutions based on Julia, as shown below in §4.7.

For generic reconstruction tasks there is less code in general, however, for the task of sequential jet reconstruction, the `Fastjet` package [40] has seen very wide adoption across many experiments. There is now a native Julia reimplementations of many of the same algorithms as are found in `Fastjet`, in the `JetReconstruction.jl` package [41]. Code ergonomics have been found to be superior, taking advantage of many of Julia's features such as array

broadcasting and generic programming. Performance is, on average 16% better than Fastjet for popular  $pp$  reconstruction algorithms, and 33% better for the Durham algorithm for  $e^+e^-$  events in FCCee studies [42]. Using Julia's package extension mechanisms, the ubiquitous EDM4hep [30] event data model is supported directly. This adds to the user experience, where not only can the package offer simpler, more generic interfaces than Fastjet, but there is the ability to seamlessly hook into the rest of the Julia ecosystem, e.g., for jet visualisation with the native Julia `Makie.jl` package [43].

## 4.6 Analysis

In most HEP experiments, after the main production of reconstructed objects at a level suitable for analysis, the path to publication diverges into many individual and group analyses. Here, as there is a relative independence from code in other languages, Julia can be a highly efficient alternative to other analysis pipelines. An overview of the suitability of Julia for analysis is found in Stanitzki and Strube [44]. Statistical tools are well supported in Julia, e.g., the Julia Statistics community is a good entry point. HEP specific histogramming needs are supported by `FHist.jl`, which provides error propagation and high performance.

At this phase of the analysis, data, usually in ROOT format, is read (§4.2). Then there is a growing ecosystem of packages dedicated to HEP analysis. For example, for hadron physics one can use `ThreeBodyDecays.jl` to build hadronic decays with cascade reactions. Partial-wave analysis is then a common technique, which is supported by `PartialWaveFunctions.jl`, as well as fitting hadronic line shapes, via `HadronicLineshapes.jl`.

Such a corpus of packages have been used to support many final physics analyses, e.g., [45–47].

## 4.7 End-to-end Computing

The LEGEND [48] experiment demonstrates how Julia can be used as a basis for end-to-end analysis in a larger physics experiment with multiple subsystems. LEGEND uses Julia as its official secondary software stack, both to verify the results of the primary software stack (written in Python) and as a test-bed for future software technologies. The whole data analysis chain, encompassing raw waveform data signal processing, ML-based data quality cuts, data calibration, event building and high-level statistical analysis is implemented completely in Julia. LEGEND uses the Bayesian Analysis Toolkit in Julia (`BAT.jl`) [49] as its primary Bayesian framework for both background decomposition and final physics analysis, and the Julia package `SolidStateDetectors.jl` [50] for detector simulation and detector design. With the exception of a custom Geant4-based software, the LEGEND collaboration is now able to perform any simulation and analysis task in Julia.

## 5 Conclusions

As we have shown Julia is an ideal language for scientific computing. It combines runtime efficiency and scalability, which is as good as C++ and Fortran, with a modern high-productivity setup, suitable for rapid prototyping. AI and ML packages are supported, either as wrapped versions of Python/C++, or as native implementations. The higher level features of the language lend themselves to compact code that expresses mathematical operations in a natural way. Multiple dispatch, supported by Julia's type system, allows excellent code composition, with a very high degree of code reuse. Julia's package system allows for easy setup

of environments and excellent reproducibility. The language also interfaces with existing HEP code bases, allowing reuse of large codes, with improved ergonomics and integration with other Julia packages.

The Julia community in HEP is young, but there is a growing HEP specific set of packages that are making Julia productive and useful specifically for high-energy physics. The advantages of the language we find to be compelling and a path of gradual adoption, helped by groups like JuliaHEP is both possible and desirable.

## References

- [1] S. Malik, K. Lieret, P. Elmer, M. Hernandez Villanueva, S. Roiser, *Train to Sustain*, in *European Physical Journal Web of Conferences* (2024), Vol. 295, p. 05023
- [2] ATLAS Collaboration (2024), <https://cds.cern.ch/record/2904204>
- [3] HEP Software Foundation, *A Roadmap for HEP Software and Computing R&D for the 2020s*, *Computing and Software for Big Science* **3** (2019), <https://doi.org/10.1007/s41781-018-0018-8>
- [4] J. Pivarski, *History and Adoption of Programming Languages in NHEP* (2022), <https://indico.jlab.org/event/505/contributions/9207/>
- [5] J. Zoll, R. Brun, J. Shiers, S. Banerjee, O. Schaile, B.a. Holl, *ZEBRA : Overview of the ZEBRA System* (1995), <https://cds.cern.ch/record/2296399>
- [6] J. Eschle, et al., *Potential of the Julia programming language for high energy physics computing* (2023), 2306.03675, <https://doi.org/10.48550/arXiv.2306.03675>
- [7] J. Bezanson, A. Edelman, S. Karpinski, V.B. Shah, *Julia: A Fresh Approach to Numerical Computing*, *SIAM Review* **59**, 65 (2017), <https://doi.org/10.1137/141000671>
- [8] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V.B. Shah, J. Vitek, L. Zoubitzky, *Julia: Dynamism and Performance Reconciled by Design*, *Proc. ACM Program. Lang.* **2** (2018), <https://doi.org/10.1145/3276490>
- [9] T. Besard, C. Foket, B. De Sutter, *Effective Extensible Programming: Unleashing Julia on GPUs*, *IEEE Transactions on Parallel and Distributed Systems* (2018), 1712.03112, <https://doi.org/10.1109/TPDS.2018.2872064>
- [10] J. Samaroo, et al. (2025), <https://doi.org/10.5281/zenodo.1482675>
- [11] T. Besard, V. Churavy, A. Edelman, B.D. Sutter, *Rapid software prototyping for heterogeneous and distributed platforms*, *Advances in Engineering Software* **132**, 29 (2019), <https://doi.org/10.1016/j.advengsoft.2019.02.002>
- [12] T. Besard, C. Foket, B. De Sutter, *Effective Extensible Programming: Unleashing Julia on GPUs*, *IEEE Transactions on Parallel and Distributed Systems* **30**, 827 (2019)
- [13] M. Innes, *Flux: Elegant Machine Learning with Julia*, *Journal of Open Source Software* (2018), <https://doi.org/10.21105/joss.00602>
- [14] A. Ramadhan, G.L. Wagner, C. Hill, C. Jean-Michel, V. Churavy, A. Souza, A. Edelman, R. Ferrari, J. Marshall, *Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs*, *Journal of Open Source Software* **5**, 2018 (2020)
- [15] S. Hunold, S. Steiner, *Benchmarking Julia's communication performance: Is Julia HPC ready or Full HPC?*, in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)* (IEEE, 2020)
- [16] S. Byrne, L.C. Wilcox, V. Churavy, *MPI.jl: Julia bindings for the Message Passing Interface*, in *Proceedings of the JuliaCon Conferences* (2021), Vol. 1, p. 68

- [17] M. Giordano, M. Klöwer, V. Churavy, *Productivity meets performance: Julia on A64FX*, in *2022 IEEE Intl. Conf. on Cluster Comp. (CLUSTER)* (IEEE, 2022)
- [18] J.M. Teichgräber, *Julia: A competitive high-level choice for performance portability in HPC?*, in *Seminar (Performance) Portable Programming of HPC Applications* (2022)
- [19] W.F. Godoy et al., *Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes*, in *2023 IEEE Int. Parallel and Distributed Processing Sym. Workshops (IPDPSW)* (IEEE, 2023)
- [20] J. Regier et al., *Cataloging the Visible Universe Through Bayesian Inference at Petascale*, in *2018 IEEE Intl. Parallel and Distributed Processing Symposium* (2018)
- [21] J. Regier et al., *Approximate inference for constructing astronomical catalogs from images*, *The Annals of Applied Statistics* **13**, 1884 (2019), <https://doi.org/10.1214/19-AOAS1258>
- [22] Tech. rep., CERN, Geneva (2022), <https://cds.cern.ch/record/2802918>
- [23] CMS Offline Software and Computing, Tech. rep., CERN, Geneva (2022), <https://cds.cern.ch/record/2815292>
- [24] A. Valassi et al., *Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC*, *Comp. and Software for Big Science* **5**, 12 (2021), <https://doi.org/10.1007/s41781-021-00055-1>
- [25] Wang et al., *AUGEM: Automatically generate high performance Dense Linear Algebra kernels on x86 CPUs*, in *SC '13* (2013), pp. 1–12
- [26] J.M. Perkel, *Julia: come for the syntax, stay for the speed*, *Nature* **572**, 141 (2019), <https://doi.org/10.1038/d41586-019-02310-3>
- [27] I. Antcheva et al., *ROOT: A C++ framework for petabyte data storage, statistical analysis and visualization*, *Comput. Phys. Commun.* **182**, 1384 (2011)
- [28] J. Pata, O. Schulz, P. Gras, *Julia interface to the C++ ROOT* (2025), <https://github.com/JuliaHEP/ROOT.jl>
- [29] T. Gál, J.J. Ling, N. Amin, *UnROOT: an I/O library for the CERN ROOT file format written in Julia*, *J. Open Source Softw.* **7**, 4452 (2022)
- [30] F. Gaede, T. Madlener, P. Declara Fernandez, G. Ganis, B. Hegner, C. Helsens, A. Sailer, G. A. Stewart, V. Voelkl, *EDM4hep - a common event data model for HEP experiments*, *PoS ICHEP2022*, 1237 (2022)
- [31] J.M. Campbell, M. Diefenthaler, T.J. Hobbs, S. Höche, J. Isaacson, F. Kling, S. Mrenna, J. Reuter, S. Alioli, J.R. Andersen et al., *Event generators for high-energy physics experiments*, *SciPost physics* **16**, 130 (2024)
- [32] J. Albrecht et al. (HEP Software Foundation), *A Roadmap for HEP Software and Computing R&D for the 2020s*, *Comput. Softw. Big Sci.* **3**, 7 (2019), 1712.06982
- [33] S. Amoroso et al. (HSF Physics Event Generator WG), *Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC*, *Comput. Softw. Big Sci.* **5**, 12 (2021), 2004.13687
- [34] E. Yazgan et al. (HSF Physics Event Generator WG), *HL-LHC Computing Review Stage-2, Common Software Projects: Event Generators* (2021), 2109.14938
- [35] U. Hernandez Acosta, A. Reinhard, S. Ehrig, T. Jungnickel, K. Steiniger, *QuantumElectrodynamics.jl*, <https://github.com/QEDjl-project>
- [36] A. Fedotov, A. Ilderton, F. Karbstein, B. King, D. Seipt, H. Taya, G. Torgrimsson, *Advances in QED with intense background fields*, *Phys. Rept.* **1010**, 1 (2023), 2203.00019
- [37] S. Agostinelli et al. (GEANT4), *GEANT4 - A Simulation Toolkit*, *Nucl. Instrum. Meth. A* **506**, 250 (2003)

- [38] J. Allison and et al., *Recent developments in Geant4*, Nucl. Instruments and Methods in Phys. Research Section A **835**, 186 (2016), <https://doi.org/10.1016/j.nima.2016.06.125>
- [39] M. Ali, M.A. Charaf, M.K. Charaf, A. Bocci, P. Gras, R.E. Houssami, *Porting the CMS pixel reconstruction to Julia: preliminary results* (2024), <https://indico.cern.ch/event/1410341/contributions/6135572/>
- [40] M. Cacciari, G.P. Salam, G. Soyez, *FastJet User Manual*, Eur. Phys. J. C **72**, 1896 (2012), 1111.6097
- [41] G.A. Stewart, P. Gras, B. Hegner, A. Krasnopolski, *Polyglot Jet Finding*, EPJ Web of Conf. **295**, 05017 (2024), <https://doi.org/10.1051/epjconf/202429505017>
- [42] G.A. Stewart, S. Ganguly, A. Krasnopolski, P. Gras, S. Ghosh, *Fast Jet Finding in Julia*, EPJ Web of Conf. (CHEP2024) (in press)
- [43] S. Danisch, J. Krumbiegel, *Makie.jl: Flexible high-performance data visualization for Julia*, Journal of Open Source Software **6**, 3349 (2021), <https://doi.org/10.21105/joss.03349>
- [44] M. Stanitzki, J. Strube, *Performance of Julia for High Energy Physics Analyses*, Comput. Softw. Big Sci. **5**, 10 (2021), 2003.11952, <https://doi.org/10.1007/s41781-021-00053-3>
- [45] R. Aaij et al., *Study of the doubly charmed tetraquark  $T_{cc}^+$* , Nature Communications **13**, 3351 (2022), <https://doi.org/10.1038/s41467-022-30206-w>
- [46] R. Aaij et al. (LHCb Collaboration), *Observation of excited  $\Omega_c^0$  baryons in  $\Omega_b^- \rightarrow \Xi_c^+ K^- \pi^-$  decays*, Phys. Rev. D **104**, L091102 (2021), <https://link.aps.org/doi/10.1103/PhysRevD.104.L091102>
- [47] Ł. Bibrzycki, C. Fernández-Ramírez, V. Mathieu, M. Mikhasenko, M. Albaladejo, A.N. Hiller Blin, A. Pilloni, A.P. Szczepaniak,  *$\pi^- p \rightarrow \eta' \pi^- p$  in the double-Regge region*, The European Physical Journal C **81**, 647 (2021), <https://doi.org/10.1140/epjc/s10052-021-09420-1>
- [48] N. Abgrall et al., *The large enriched germanium experiment for neutrinoless double beta decay (LEGEND)* (AIP, 2017), Vol. 1894 of American Institute of Physics Conference Series, p. 020027, 1709.01980
- [49] O. Schulz, F. Beaujean, A. Caldwell, C. Grunwald, V. Hafych, K. Kröninger, S.L. Cagnina, L. Röhrig, L. Shtembari, *BAT.jl: A Julia-Based Tool for Bayesian Inference*, SN Computer Science **2**, 210 (2021), <https://doi.org/10.1007/s42979-021-00626-4>
- [50] I. Abt, F. Fischer, F. Hagemann, L. Hauertmann, O. Schulz, M. Schuster, A.J. Zsigmond, *Simulation of semiconductor detectors in 3D with SolidStateDetectors.jl*, Journal of Instrumentation **16**, P08007 (2021), <https://doi.org/10.1088/1748-0221/16/08/p08007>