Computing and Software for Big Science (2023) 7:10
https://doi.org/10.1007/s41781-023-00104-x

RESEARCH

Potential of the Julia Programming Language for High Energy Physics Computing

Jonas Eschle[1] · Tamás Gál[2] · Mosè Giordano[3] · Philippe Gras[4] · Benedikt Hegner[5] · Lukas Heinrich[6] · Uwe Hernandez Acosta[7,8] · Stefan Kluth[6] · Jerry Ling[9] · Pere Mato[5] · Mikhail Mikhasenko[10,11] · Alexander Moreno Briceño[12] · Jim Pivarski[13] · Konstantinos Samaras-Tsakiris[5] · Oliver Schulz[6] · Graeme Andrew Stewart[5] · Jan Strube[14,15] · Vassil Vassilev[13]

Received: 6 June 2023 / Accepted: 11 August 2023

Abstract
Research in high energy physics (HEP) requires huge amounts of computing and storage, putting strong constraints on the code speed and resource usage. To meet these requirements, a compiled high-performance language is typically used; while for physicists, who focus on the application when developing the code, better research productivity pleads for a high-level programming language. A popular approach consists of combining Python, used for the high-level interface, and C++, used for the computing intensive part of the code. A more convenient and efficient approach would be to use a language that provides both high-level programming and high-performance. The Julia programming language, developed at MIT especially to allow the use of a single language in research activities, has followed this path. In this paper the applicability of using the Julia language for HEP research is explored, covering the different aspects that are important for HEP code development: runtime performance, handling of large projects, interface with legacy code, distributed computing, training, and ease of programming. The study shows that the HEP community would benefit from a large scale adoption of this programming language. The HEP-specific foundation libraries that would need to be consolidated are identified.

Keywords Julia · HEP · Python · High energy and nuclear physics · Programming language · HPC

Introduction

High throughput computing plays a major role in high energy physics (HEP) research. The field requires the development of sophisticated computing codes, which are continuously evolving in the course of the research work. Computing grids, connecting computer centers all around the world, are required to process the experiments' data [1]. Computer algebra systems and high performance computers are used to build new models and to calculate particle production cross sections.

✉ Philippe Gras
philippe.gras@cern.ch

1 University of Zurich, Zürich, Switzerland

2 Erlangen Centre for Astroparticle Physics, Friedrich-Alexander-Universität, Erlangen-Nürnberg, Germany

3 University College London, London, UK

4 IRFU, CEA, Université Paris-Saclay, Gif-sur-Yvette, France

5 CERN, European Organization for Nuclear Research, Geneva, Switzerland

6 Max-Planck-Institut für Physik, Munich, Germany

7 Center for Advanced Systems Understanding, Görlitz, Germany

8 Helmholtz-Zentrum Dresden-Rossendorf, Dresden, Germany

9 Laboratory for Particle Physics and Cosmology, Harvard University, Cambridge, MA, USA

10 ORIGINS Excellence Cluster, Garching, Germany

11 Ludwig-Maximilians-Universität, Munich, Germany

12 Universidad Antonio Nariño, Ibagué, Colombia

13 Princeton University, Princeton, NJ, USA

14 Pacific Northwest National Laboratory, Richland, WA, USA

15 University of Oregon, Eugene, OR, USA

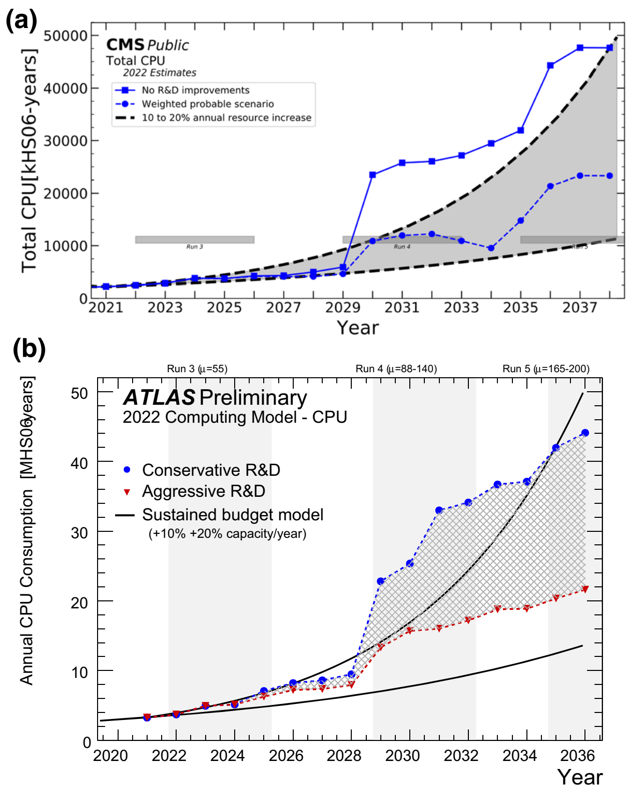

Fig. 1 Estimated CPU required by the CMS (top) and ATLAS (bottom) experiments for LHC and HL-LHC [6, 7]

Fig. 1 shows the expected needs for the ATLAS and CMS experiments [2, 3] at the Large Hadron Collider (LHC) [4] and its successor, the high-luminosity LHC (HL-LHC) [5], together with the estimated planned resources. A data processing improvement from R&D is required for HL-LHC to fit within the planned resources, which total $20 \cdot 10^9$ HS06[1] units of CPU resource.

The need to reconcile high performance with fast development has led to the development of a C++ interpreter [8] that provides the convenience of a read-eval-print-loop (REPL) interactive experience, also known as programming shell, that supports just-in-time compilation, and allows the use of the same programming language for compiled and interpreted code. The same analysis framework `ROOT` [9, 10] can then be used with compiled code and interactively. In addition to the REPL, ROOT supports Jupyter notebooks, which are another convenient method for interactive use. The shortcoming of this approach is that the use complex programming language is not optimal for easy and fast coding. For this reason, another approach that consists of using two languages, one optimal for fast development, typically Python, and one optimal for high performance, typically C++, is often adopted.

[1] https://www.spec.org/cpu2006/.

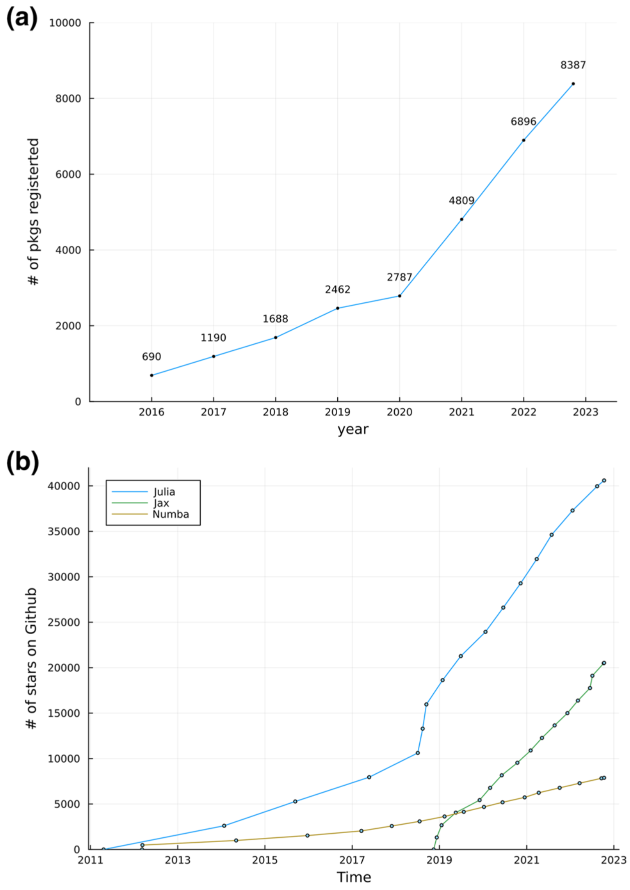

Fig. 2 Number of packages registered in the Julia general repository that can be installed by the integrated package manager (top) and of Julia language code GitHub stars (bottom) as function of time. The trend of the star counts is compared with `Numba` and `Jax`

Using two languages is not ideal: it expands the required area of expertise; it forces the reimplementation, in the high-performance language, of pieces of code originally written with the fast-development language when they do not meet the required performance; and it reduces the reusability of code.

In 2009, J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah imagined a new programming language to address this “two language problem” by providing high performance and ease of programming [11–13] simultaneously. It has been a successful approach. The new language, Julia, which has evolved year upon year, is now used by many users. Julia is a dynamic language, similar to Python, yet with a performance similar to C/C++. As of October 11, 2022, 8387 packages were registered in the Julia general registry [14], which are accessible to Julia’s integrated package manager. Fig. 2 shows the rapid growth in the number of packages.

As demonstrated by M. Stanitzki and J. Strube [15], the Julia language is a good alternative to the combination of C++ and Python for HEP data analysis and it fulfils its promise to be an easy, high performance language. This report extends that study. It explores the possible benefits of the adoption of Julia as the main programming language for HEP, in place of C++-Python, in a similar way as happened with the switch from Fortran to C++ in the late 90's.

The Programming Language Community

More important than a list of technical features, however, are the culture and interests of a programming language's community, because the language and its implementation will evolve to satisfy those interests. For instance, the Haskell community is focused on language theory, and is unlikely to put much effort into optimization for high performance computing, and the Go community is so focused on language simplicity that they have resisted try-catch logic [16]. The Julia community's interests are well aligned with HEP, and many Julia users are in the sciences. We see this in some design choices inherited from existing technical languages (Fortran, R, MATLAB, Wolfram), like 1-based indexing, column-major arrays, and built-in N-dimensional arrays, and also in the effort placed on interoperability with other languages: `ccall`, `PyCall.jl`, `RCall.jl`, `MathLink.jl`, and `JavaCall.jl`. Julia is supported by NumFOCUS (like many Python data science projects), and many of its most prominent applications are in numerical computing: NASA spacecraft modeling [17]; climate science [18]; and the Celeste project [19], which achieved 1.54 Petaflops on the Cori II supercomputer [20], a first for a dynamic language.

Key Features of Julia

To locate Julia in the space of programming languages, its primary features are characterized below.

- A single implementation, rather than an abstract language specification with multiple slightly incompatible implementations. The Julia computing platform is primarily implemented in Julia (most parts), C, and C++ (LLVM), and it has a built-in REPL.
- Every function, including those entered interactively, is compiled just-in-time (JIT) using LLVM as the back end. Julia has no virtual machine, and the JIT is eager (always compiles before execution), unlike the tracing/hot spot JIT seen in `LuaJIT` [21] or metatracing [22] like `PyPy` [23].
- Partly thanks to dynamic typing, and also being a JIT language, Julia fully supports: type reflection, source code as a built-in data type, which enables Lisp-like (hygienic) macros[2].
- Fast N-dimensional arrays that store elements in-place.
- Multiple dispatch (MD): a function-call invokes the most specific method that matches the type of *all* arguments. While many languages support (opt-in) multiple dispatch (C#, Common Lisp), Julia is the first language that uses MD as the central paradigm[3] while focusing on performance. MD allows for a surprising amount of code reuse and composition among packages that do not know about about each other (often a problem with OOP languages), this is further discussed in the "Polymorphism in C++, Python, and Julia" section.
- Apart from the lack of classes, Julia has a fairly standard mix of imperative and functional programming styles. Immutability is encouraged by default, but mutable structs and arrays are allowed and are frequently used.
- Built-in parallel processing support. Any piece of a program can be marked for execution in parallel. Threads are scheduled globally—allowing a multithreaded function to call other multithreaded function—on available resources without oversubscribing, saving the developer from the burden of taking care of the number of threads. Computing distributed on several computers is supported. Julia code can run natively on GPUs.
- Objects are not reference counted, but are garbage collected. The garbage collector is standard mark-and-sweep, generational (like Java), but non-compacting, so pointers to objects are valid as long as the objects remain in scope.

The manner in which polymorphism is supported is the most noticeable difference with CBOO programming languages, like C++ and Python, and it merits a dedicated discussion.

Polymorphism in C++, Python, and Julia

Polymorphism is the "ability to provide a single interface to entities of different types" [24, 25]: a polymorphic function will accept arguments of different types. Here we compare Julia with C++ and Python due to their prevalence in HEP.

We can distinguish two classes of polymorphism [26, 27]: ad-hoc polymorphism where a different implementation is provided for each set of types, and universal polymorphism, where a single generic implementation is provided for all the sets.

Function overloading is an example of ad-hoc polymorphism, while C++ templates are an example of universal polymorphism. Ad-hoc polymorphism can be combined with universal polymorphism using template specialization:

[2] https://en.wikipedia.org/wiki/Hygienic_macro.

[3] Not only Julia lacks OOP by design, it also has highest number of methods and degree of specialization among languages that support MD [11].

several implementations are provided, while each implementation can be generic, either partially or totally. A particular universal polymorphism is based on subtypes: the function scope is extended to all subtypes of its argument.

Polymorphism can be static, i.e., resolved at compile time, or dynamic, i.e., resolved at runtime. It can apply to a single entity e.g., one of the arguments of a function, or multiple entities e.g., all the arguments of a function. In a function call, the mechanism that selects the implementation to execute according the passed argument types is called dispatch.

In the following of this subsection, we will compare polymorphism provided by Julia, C++ and Python. Code examples illustrating our statements can be found in Appendix A.

Polymorphism is provided in C++ by two paradigms: one based on class inheritance, function overloading, and function overriding; the other based on templates. The first provides ad-hoc and subtype polymorphisms over functions, while the second provides universal and ad-hoc polymorphisms over both functions and types. By exploiting the `concepts` feature introduced by C++ 20, subtyping polymorphism support can be added to the templates. The functionalities of the two paradigms overlap.

In C++, class non-static member functions take a special argument, the class instance reference (`x`) or pointer (`ptr`), through a dedicated syntax, `x.f()` and `ptr->f()`. Both static and dynamic polymorphism are supported over this argument, while only static polymorphism is provided for the other arguments. Object copy with implicit type conversion can make difficult to follow the polymorphism flow of an object.

Static ad-hoc polymorphism is supported over the arguments of global functions and static member functions. C++ class templates provide static universal polymorphism. One notable usage is the containers of the standard template library.

The class inheritance is twofold, it provides inheritance of the interface through the subtype polymorphism previously described and inheritance of the data fields, a type is an aggregation of its fields and all the fields of its supertypes. The bounding of the two inheritances can result in breaking encapsulation [28] and it is often advocated to prefer composition to inheritance for the fields, as dictated by the “second principle of object-oriented design” of Ref. [29].

Python provides single dynamic dispatch for the class instance argument of member functions. It does not provide polymorphism for other arguments. Multiple dispatch emulation can be implemented in the function using conditions on the argument types or using a decorator [30].

Julia provides ad-hoc and universal polymorphism, including subtype polymorphism [31], within a consistent multiple-dispatch paradigm. Extending the dynamic dispatch of C++ to all arguments of a functions makes it extremely powerful, especially in terms of code re-usability. The Julia multiple dispatch exploits JIT compilation and the classification into static and dynamic dispatches is less relevant here: a specialized function is compiled only before its use, although the behavior is always consistent with dynamic dispatch; inlining and other compile-time optimizations can be performed despite the dynamic behavior. Nevetheless, this optimisation is subject to the ability of the compiler to infer the type of the passed arguments and requires some attention from the developer. In particular, when code performance is important, the developer must make sure that the return type of a function can be inferred from the types of the its arguments. Julia ad-hoc and universal polymorphism uses a simple syntax similar to function overloading, but with argument types specified only when required, either to extend a function or to enforce the types of arguments.

The implementation of a two-argument function to be called by default, in absence of a more specialized implementation fitting better with the types of the passed arguments, will be defined as `function(x, y)...end`. Its specialization for a first argument of type A or of a subtype of A will be defined by suffixing the first argument `x` with `::A`. It can be further specialized for a first argument of type A or a subtype of A and second argument of type B or of a subtype B, by annotating both arguments respectively with `::A` and `::B`, which will read as, `function f(x::A, y::B)`.

Universal polymorphism for type definition is supported by parametric types. In the following example the type `Point` has two fields of the same type, that must be a subtype of the `Number` type.

```
struct Point{T <:Number}
  x ::T
  y ::T
end
```

Contrary to C++, in Julia, subtyping does not involve field inheritance. Data aggregation must be done using composition, enforcing the “second principle of object-oriented design”. Subtypes are used to define a type hierarchy for the subtype polymorphism. The hierarchy tree is defined with abstract types that do not contain data, only the leaves of the tree can be a concrete type.

Because variable assignment and function argument passing do not trigger an object copy, Julia is not affected by the difficulty encountered in C++ mentioned before. This is demonstrated with the “King of Savannah” example discussed in the Appendix A.

HEP Computing Requirements

Because the program codes used in HEP research are very large, with high interdependence, a code typically uses many open source libraries developed by other authors; thus the effort to change programming language is consequential. The adoption of a new language can happen only if it brings a substantial advantage over the already used paradigm.

The key advantage of Julia that can make the language switch worthwhile is the simplification that will arise from using a single language in place of a combination of two, C++ and Python.

HEP computing is wide and includes many use cases: automation for the controls of the experiment, data acquisition, phenomenology and physics event generation, simulation of the physical experiment, reconstruction of physics events[4] from recorded data, analysis of the reconstructed events, and more.

We will review in this section the properties required for event analysis, event reconstruction, event simulation and event generation. We will start with general features, common to all the use cases.

General Features

An Easy Language

The easy language is one-side of the high-level and high-performance coin property that would motivate the adoption of Julia as a programming language. It is easy at least in two ways: easy, imperative syntax, and free of strong typing when writing code. The surface syntax of Julia largely resembles Python, MATLAB (control flow, literal array), while also getting inspiration, such as the do-block, from Lua and Ruby. It has all the high-levelness one would expect from a language: higher-order functions (functions can be returned and passed as variables), native N-dimensional arrays, nested irregular arrays (arrays of different-size arrays), and a syntax for broadcasting over arrays.

As a syntax comparison example, a for-loop will look like the following in Python, Julia, and C++.

[4] In HEP experiments, we observe collisions of subatomic particles or atoms. The result of a collision that produces new particles is called an event. Detectors, that can be complex apparatus as large as 46 m × 25 m producing tens of millions of MBytes of data per second, are used to capture the event.

```
# Python
a = 0.
for i in range(1,11):
  a += i
a /= 10.

# Julia
a = 0.
for i in 1:10
  a += i
end
a /= 10.

// C++
auto a = 0.;
for(auto i = 0;
    i < 11.; ++i){
  a += i;
}
a /= 10.;
```

We will note in this example that Julia is free of type declaration, just like Python. In this example, we use the C++ `auto` type declaration feature to achieve the same goal. It is worthy of mention that, as in C++, Julia code interpretation is not sensitive to changes in indentation: appending two spaces at the beginning of the last line, will change the behavior of the Python code only.

Julia supports list comprehension, like Python, as illustrated in this example that creates a vector with the series $1, 1/2, \ldots, 1/10$:

```
# Julia
v = [ 1/x for x in 1:10 ]

# Python
v = [ 1.0/x for x in range(1, 11) ]
```

`NumPy` [32] function vectorization is provided natively in Julia for all functions and operators, including those defined by the user, through the broadcast operator: a dot prefix is used to specify that the function must be applied to each element of a vector or of an array. The syntax is illustrated below.

```
# Julia
v = [1, 1, 1] ./ [1, 2, 3]

# Python
import numpy as np
v =  np.array([1, 1, 1]) \
   / np.array([1, 2, 3])
```

The following example illustrates the native support of linear algebra and multi-dimensional arrays and highlight the concise syntax it provides. It solves the simple equation,

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} x = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

```
# Julia
m = [ 1 1; 1 -1 ] \ [ 2, 0 ]

# Python
import numpy as np
m =  np.linalg.solve(
       np.array([[1, 1], [1, -1]]),
       np.array([2, 0]))
```

Because broadcasting has its own syntax, Julia is able to use mathematical operators “correctly” when they are not broadcast, instead of relying special names e.g., matrix multiplication (`np.matmul`) and exponentiation (`np.ling.expm`).

Two additional language design choices are worth noting for their contribution to make the language easy to use without sacrificing performance. The first is the evaluation of the function default parameter values, done at each call, instead of once for all in Python. Thanks to this choice, a function `f(x, v=[])` that appends the element x to vector v and returns the latter will always return a one-element when called as `f(x)` in Julia, while it will return a vector growing in size at each new call in Python. The second is the copy performed by the updating operators (+=, *=,..) instead of the in-place operation done by Python. In the Julia code, `A = [0, 1]; B = A; B += [1,1]`, the operator += will not modify the content of the vector `A`, it is syntactically strictly equivalent to `B = B + [1, 1]`), while it will in the Python code `A = np.array([0, 1]); B = A; B += np.array([1, 1])`. We judge these two Julia behaviors more natural and more likely to match to what a non-expert would expect when writing or reading the code.

As we illustrated with few examples, the Julia language is as easy as Python and sometimes easier thanks to a native support of features provided by external packages in Python that allows for a more concise and natural syntax.

Performance

The other major advantage to Julia is high performance. Julia provides performance similar to C++, and in some cases even surpassing C++, as can be seen in Fig 3. The shown comparison is obtained by repeating the micro-benchmark[5] for Julia version 1.9.0rc1 and Python 3.9.2. C and Julia implementations use OpenBLAS for the matrix operations, while Python uses `NumPy` (version 1.24.2) and OpenBLAS. This benchmark compares the run time of some short algorithms implemented in a similar way in the different languages. The results are divided by the time the C/C++ implementation takes. The GNU compiler `gcc` version 10.2.1 has been used. The test is performed on a laptop equipped with a 11th Gen Intel(R) Core(TM) i5-1135G7@2.40GHz CPU and 16 GB of random access memory (RAM) running the Linux operating system compile for x86 64-bit architecture. The 64-bit flavor OpenBLAS library version 0.3.21 is used. This setup is used for all performance tests described in this paper, if not specified otherwise. The score goes from 0.73 to 1.67 (smaller is better) for Julia and 1.12 to 107 for Python. C is doing the best with respect to the two other languages for the recursive Fibonacci algorithm, implemented in Julia as below.

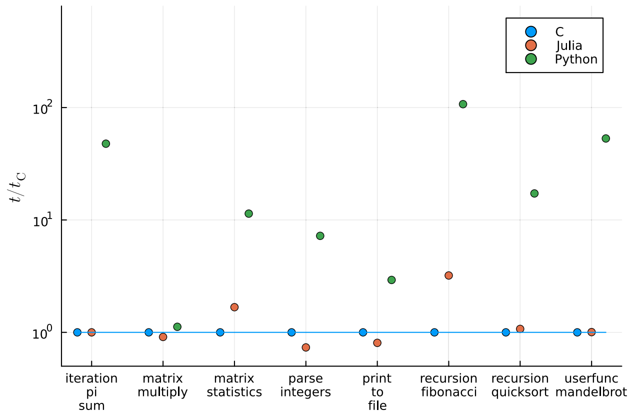

Fig. 3 Comparison of C/C++, Python and Julia language performance for a set of short algorithms. OpenBLAS, together with `NumPy` in the Python case are used for matrix operation. The score is defined as the time to run the algorithm divided by the time to run the C version of the same algorithm

```
fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)
```

This benchmark tests the performance for recursive calls. While expert developers typically avoid it for performance reasons, a recursive expression is the easiest and most natural way to implement a recursive algorithm. The mathematical series $u_{n+1} = f(u_n)$ maps directly to a recursive computing function call. We compute the 20th Fibonacci series elements, which results in 21,891 nested calls, a good example of recursive calls. The C/C++ implementation is doing better because of a tail recursion optimization performed

[5] https://julialang.org/benchmarks/.

by the compiler, that removes one of the two nested calls, disabling this optimization leads to performance a little worse than with Julia. The gain from this optimization is far from the one obtained by using a for-loop implementation instead of the recursion. Such implementation runs ≈1000 times faster. The tail recursion optimization does not work for the recursive quicksort, leading to similar performance from C/C++ and Julia (7% difference in favor of C/C++).

We can use LHC open data to test performance on HEP-oriented code. We make this test with a di-muon analysis on CMS data of LHC Run-1, from 2011 and 2012. The analysis consists of measuring the spectrum of the mass of the system made of a muon and an antimuon, produced in proton-proton collisions at the center-of-mass energy $\sqrt{s} = 7$ TeV. It uses data in which the muons and antimuons are already reconstructed and identified. It does not correct for instrumental efficiencies, contrary to the published physics results.

Different implementations have been compared: the for-loop based Julia implementation from Ref. [33], the equivalent for-loop based implementations in Python and C++, the `ROOT` data frame (RDataFrame) implementation from Ref. [34], its equivalent in C++ in two flavors, and a data frame based implementation done in Julia using the `DataFrames.jl` package [35]. In the data frame implementation, the table rows are first filtered to obtain a data frame with the di-muon events of interest, then a column with the dimuon mass is added to the data frame, and finally a histogram is filled. RDataFrames use lazy operations, and only the histogram is materialized, limiting the memory footprint. In the first flavor of the C++ implementation, the formula to compute the mass is provided as a character string and the code for this computation is compiled JIT. In the second flavor a user-defined C++ function is provided to compute this mass.

The input data are read from a file stored in the `ROOT` format with compression turned off. The `UnROOT.jl` package [36] (version 0.9.2) is used to read the file with the Julia code. This package is written in pure Julia. The native ROOT library (version 6.26/10) serves to read the files from C++ and Python. The GNU `gcc` compiler (version Debian-10.2.1-6) is used with a level-three optimization (option `-O3`). When JIT compilation is involved (the cases of Julia and JIT RDataFrame) the event analysis function is first run on a ten-event data file to trigger compilations before performing the timing on 1 billion events. For the Julia implementations, subsequent compilations occur during the timing loop; they represent only 1.1% of the time. In the case of JIT RDataframe an overhead (time independent of number of processed events) of 5.0 ± 0.2ms (C + + version) or 11.2 ± 2ms (Python version) is present in spite of the warm-up. The overhead is subtracted from the measurement. The obtained numbers are provided in Table 1. We observe that slight changes of source code can change the runtime of the C++ for-loop and native RDataFrame implementations beyond the statistical uncertainties. This effect is estimated by varying the code outside if the timed loop (addition of a print-out statement, change of code statement order) and included in the quoted uncertainties. For the other implementations, no significant change is observed and the quoted uncertainty include the statistical component only (at 68% confidence level).

Table 1 Comparison of the runtime of the di-muon spectrum analysis for implementations performed in different programming languages. The time corresponds to a run over 1 million events

Implementation	Time to process 10^6 events
Julia for-loop	0.147 ± 0.0014 s
Julia Dataframe	0.1839 ± 0.0019 s
Python for-loop	153.7 ± 5.7 s
Python RDataFrame	0.4083 ± 0.0083 s
C++ for-loop	0.1627 ± 0.0019 s
C++ RDataFrame	0.2338 ± 0.00031 s
C++ RDataFrame JIT	0.3051 ± 0.0023 s

In this example, the for-loop Julia implementation runs the fastest, the C++ for-loop implementation is slightly behind (11% slower). The Julia implementation using data frames takes 21% less time to run than with C++ RDataFrame. The Python for-loop implementation is 1000 times slower than with Julia. Delegating the loop to an underlying compiled library (in our case the `ROOT` library) is not sufficient to achieve good performance with Python: the RDataFrame python implementation is 2.2 (resp. 2.8) times slower than the Julia data frame (resp. for-loop) implementation. The C++ RDataFrame implementations are slower than the Julia and C++ for-loop implementations by a factor from 1.4 to 2.1 depending on the implementations we compare. The dimuon spectrum obtained with the Julia code is shown in Fig. 4.

The data frame benchmark includes the insertion of a column in the data frame with the dimuon mass. In the Julia case, the insertion is not needed for the analysis itself, but keeping it is interesting for benchmark purpose. The data frame returned by `UnROOT` does not allow direct insertion and the selected rows are copied to a DataFrames.jl data frame supporting such an insertion. That leaves room for improvements; we estimate that improved tools that would allow such insertion with no copy would reduce the runtime by 16%.

For Python, the pure python library `Uproot` [37] can be used instead of the native `ROOT` library to read the data. This library loads all the data of a file into the memory, similar to the Julia data frame implementation. The data can be provided as a set of Awkward Arrays [38], `NumPy` arrays, or as a Pandas data frame [39]. All these data structures support

Table 2 Runtime of the dimuon spectrum analysis for three Python implementations using the `Uproot` library to read the data

Implementation	Time to process 10^6 events
Vectorized with Awkward Arrays	0.2343 ± 0.0027 s
Vectorized with Panda dataframes	9.225 ± 0.081 s
For loop	177.2 ± 1.8 s

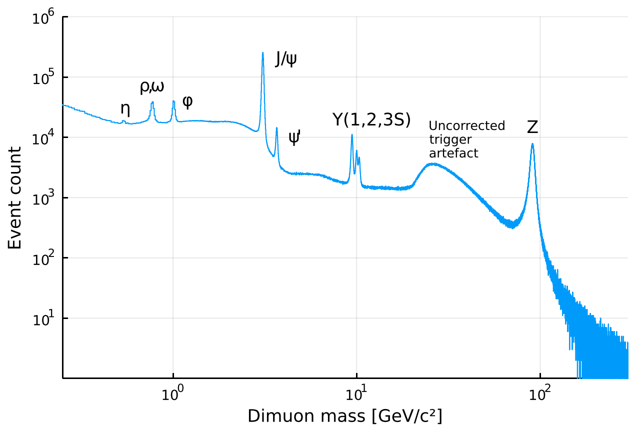

Fig. 4 Dimuon spectrum obtained from the CMS open data of Run 2012 with the Julia implementation of the analysis

vectorized operations permitting a delegation of the event loop to underlying compiled libraries improving the running time. The results are shown in Table 2. The measurement is done with `Uproot` version 4.3.4 (with `awkward` package version 1.10.3). The implementation using Awkward Arrays operating on a vector of all events runs faster than the Python RDataFrame implementation and is only 1.6 times slower than with the Julia for-loop. We note that the Python's performance is highly dependent on the algorithm implementation: the time ratio with respect to the Julia for-loop goes up to 63 for a vectorized implementation using Pandas data frames and to 1200 with the event loop.

Running on a 61.5 million event file shows that the for-loop and RDataFrame implementations scale well with larger input files with no penalty on the event throughput as we could have expected. The other implementations would require modifications in the code in order to process events in chunks and reduce the memory usage. The awkward array implementation requires 14.5 GiB, at the limit of 15 GiB available on the machine used for the measurement, while the Pandas and Julia data frame versions exceed this limit.

We see in this example that Julia is performing similar or better than C++ frameworks. For an event loop, Python is slower by three-orders-of-magnitude than Julia. Vectorization of event processing serves as a mitigation of Python's slowness by delegating the event loop to underlying compiled libraries and sacrifice flexibility, without achieving the performance of C++ and Julia[6].

Load the Julia magic extension

```
%load_ext julia.magic
```

```
Initializing Julia interpreter. This may take␣
↪some time...
```

Define a variable in a Python-code cell

```
x=10
```

Execute some code written in Julia. The variable `x` defined in the Python space is accessible from Julia.

```
%%julia
y = $x^2
println("x² = ", y)
```

```
x² = 100
```

Execute code in Python. The variable `y` defined in the Julia space is accessible from Python.

```
y = %julia y
print("x² =", y)
```

```
x² = 100
```

Fig. 5 Example of a Jupyter notebook mixing cells with Julia and Python code

Interoperability with Legacy Code

HEP computing is based on a heritage of program code written over decades. Interfacing to libraries developed in C++ and Fortran is unavoidable, apart from the last-step of analysis domain (and even here it would still be an attractive feature). Julia can natively call C and Fortran functions with no overhead compared to calling them from their native language. Examples of such calls are given in Listings 1 and 2. For convenience, a wrapper function written in Julia can be used to handle errors, as in the example in Listing 3.

Bindings to Python are supported thanks to the `PyCall` package. The interface is very convenient and transparent in both directions, Python from Julia and Julia from Python, as we can see in the examples provided in Listings 4 and 5. In a Jupyter notebook, in addition to calling a Julia function from a notebook running a Python kernel using these interfaces and vice-versa, it is possible to write Julia code in cells of a notebook using a Python Kernel, and mix cells written in Julia and in Python languages, as illustrated in Fig. 5.

[6] Recently, it became possible to use Numba + Awkward Array to enable fast loops, sacrificing some Python features due to the more strict compiling model

```julia
function compute_dot(DX::Vector{Float64},
                     DY::Vector{Float64})
  @assert length(DX) == length(DY)
  n = length(DX)
  incx = incy = 1
  product = ccall((:ddot_, "libLAPACK"),
            Float64,
            (Ref{Int32}, Ptr{Float64},
            Ref{Int32}, Ptr{Float64},
            Ref{Int32}),
            n, DX, incx, DY, incy)
  return product
end
```

Listing 1: Example of call from Julia of a function implemented in Fortran [40]

```julia
path = ccall(:getenv, Cstring, (Cstring,),
            "SHELL")
println(unsafe_string(path))
```

Listing 2: Example of call of a function of a C-library from Julia [40]

```julia
function getenv(var::AbstractString)
  val = ccall(:getenv, Cstring,
            (Cstring,), var)
  if val == C_NULL
    error("getenv: undefined variable: ",
          var)
  end
  return unsafe_string(val)
end
```

Listing 3: Example of a wrapper in Julia to handle errors from a c-library function [40]

```julia
# Enable Python call:
using PyCall

# Inport a python module:
math = pyimport("math")

# Use it as a Julia module:
math.sin(math.pi / 4)
```

Listing 4: Python function can be called transparently from Julia. Example of call of a function from the Python `math` package. [40]

```
$ python3
>>> import julia
>>> julia.install()
>>> from julia import Base
>>> Base.sind(90)
1.0
```

Listing 5: The Python package named `julia` allows use of Julia packages within Python codes.

Load the Julia magic extension

```
%load_ext julia.magic
```

```
Initializing Julia interpreter. This may take␣
↪some time...
```

Define a variable in a Python-code cell

```
x=10
```

Execute some code written in Julia. The variable `x` defined in the Python space is accessible from Julia.

```
%%julia
y = $x^2
println("x² = ", y)
```

```
x² = 100
```

Execute code in Python. The variable y defined in the Julia space is accessible from Python.

```
y = %julia y
print("x² =", y)
```

```
x² = 100
```

The `CxxWrap` package [41] can be used to add Julia bindings to C++ libraries. Once bound, the library is accessed transparently from Julia as if it was a native Julia package. The `object.method(args...)` and `object_ptr->method(args...)` C++ like method calls translate into the Julia-like call `method(object, args...)`. The package philosophy is similar to `Boost.Python` [42] and `Pybind11` [43]: the bindings are produced with few lines of C++ code, one line per class and one line per method, which must be compiled as a shared library. The package provides all the flexibility to expose a different Julia interface to the C++ one, for instance to adapt it to the Julia context and style of programming. `CxxWrap` internally uses the built-in Julia-C interface, used for the interface between shared libraries and Julia. The C++ standard

Table 3 Mean time to call the `Fill` method of a `ROOT` histogram from C++, Julia and Python

	Mean time [ns]
C++	5.74 ± 0.01
C API from C	6.48 ± 0.04
C API from Julia	5.97 ± 0.03
Julia - `CxxWrap`	8.21 ± 0.04
Julia - `Cxx`	5.97 ± (< 0.01)
Python	226 ± 5

The time corresponding to a single call is averaged on 10^6 `h->Fill(1.)` calls. Three cases are considered for Julia: use of the plain Julia C interface ("C API"), use of `CxxWrap`, and use of `Cxx`. For reference the time to call the same function from C/C++, within the same code ("C++ ") and through the shared library developed the Julia C interface, is also measured

template library `vectors` and `val_arrays` are mapped to Julia `Vectors` with zero copy.

The `WrapIt` project [44] has demonstrated that binding code can be generated automatically from a library's header files, which would make the process of adding Julia bindings to C++ libraries very easy. Automation of this Julia binding has been tested on the `ROOT` libraries, and we have been able to produce, draw, and fit histograms and graphs (`TGraph` class). The fit has been tested with both functions defined in `ROOT` and functions defined in Julia, demonstrating a perfect integration.

Unlike direct calls into C or Fortran libraries via the `ccall` function, calls between C++ and Julia have to go through the intermediate layer created by the wrapper code. We perform several measurements to estimate the overhead from the C++-Julia interface. The measurement is performed on a call to the `ROOT` `TH1D::Fill` method, that adds a value to a histogram: we time a loop of 1 million calls and average the result to get the time per call. First, we create a shared library, that exports C functions, we call from Julia with the `ccall` method. The pointer to the histogram object is passed to the C function as a `void*` type. When compared with a direct call to the `Fill` method within the same C++ code, the call from Julia shows an overhead of 0.23 ns. This overhead is unexpectedly smaller than when calling the wrapper function of the shared library from a program written in C: observed overhead of 0.74 ns in this case. In the end, the call from Julia takes only 4% more time than a direct call from C++. It is 38 times faster than a call from Python. Measurement is also done for a binding based on `CxxWrap`. All results are shown in Table 3.

We could imagine the Julia engine performing just-in-time compilation of C++ using the LLVM infrastructure it uses for the Julia code. The `Cxx` package [45] is providing this feature for Julia releases from 1.1.x to 1.3.x. With this package we can access to a C++ library without the need of a C++ wrapper. Nevertheless, a Julia wrapper is needed to provide the same transparency—calls to the C++ functions similar as calls to a Julia functions. Using this package the call to the `Fill` function in our example is found to be as fast as when using the C interface, as shown in Table 3. We used Julia version 1.3.1 to perform this measurement. The `Cxx` approach is a good alternative to `CxxWrap`.

In Ref. [15], `CxxWrap` was used to interface to the `LCIO` C++ library [46] to read ILC [47] simulated events and to `Fastjet` [48, 49] to cluster hadronic jets. The loss of event throughput compared to a code uniformly written in C++ was 37%.

Support of Standard HEP Data Formats

Different file formats are used to store HEP data and supporting them is crucial to a streamlined physics analysis experience.

The file formats currently used to store physics events are mainly `HepMC`, `LHE` [50], `LCIO`, and `ROOT`. The pachages `LHE.jl` [51] and `LCIO.jl` [52] provide supports for `LHE` and `LCIO`. Two packages are available to read `ROOT` files: `UpROOT.jl` [53] and `UnROOT.jl` [36].

The `UpROOT.jl` package uses the `Uproot` pure-Python library to provide read and write support. When using this package, a loop on events typically suffers of the same performance penalty as with Python. This has motivated the development of `UnROOT.jl`, a package written in pure Julia that provides a fast processing of events, as demonstrated in the performance measurements done in the previous section, which used this package. It leaves the flexibility to use an explicit event loop, with a small memory footprint, or works on vector of event quantities ("columnar analysis"). An event loop will look like the following code snippet, where `Muon_pt` is a vector (transverse momenta of the muons contained in the event).

```
for event in mytree
  # Access to a single-event quantity
  event.Muon_pt
end
```

A columnar analysis will look like the following.

```
# Access to a vector of event quantities
# (themselves vectors of numbers)
mytree.Muon_pt
```

UnROOT.jl uses thread-local storage to maximize performance and maintain thread-safety. An event loop can be parallelized in several threads with little effort using the standard Julia @threads macro:

```
julia> @threads for event in mytree
# ... Operate on event
end
```

The performance measurement presented in the "Performance" section are done in single-thread mode. There are limitations. First, this package does not support data write. Both UpRoot.jl and UnROOT.jl can access only to objects of a limited set of types, either stored as such in the file or in a TTree. The supported types covers already a large set of use cases, but not schemes where data is stored as object of serialized C++ classes. Using the genuine ROOT library via a Julia binding based on CxxWrap can be an alternative approach when required. We have successfully read and write histograms (TH1 objects) and graphs (TGraph objects) using this approach.

We expect the implementation of the support of RNTuple to be easier than TTree that it is expected to replace, thanks to its design. Data are stored in column of fundamental types (float, int,...) [54], similar to Apache Arrow [55], which should ease support from programming languages other than C++ like Julia.

In the neutrino physics community, the industry-standard HDF5 and Parquet have been used at scale, and these files can be readily read and write from Julia via their respective packages.

Parallel Computing

Apart from having memory shared multi-threading, Julia also ships with out-of-core distributed computing capability as a standard library (Distributed). In fact, it is as easy to command an array of heterogeneous nodes in real-time via packages such as ClusterManagers.jl [56], which can mimic Dask's experience [57] with a fraction of the code. For more advanced features, such as building out-of-core computation graphs, Dagger.jl [58] provides facilities.

While these libaries allow distribution of execution within the Julia code, parallelization can be also done, as with C++ and Python, by running parallel jobs of the same executable using commands of a batch processing system, like HTCondor [59, 60], typically used in computer cluster facilities.

Platform Supports and License

Julia is supported on all major platforms, a list of which can be found on the Julia website[7]. Three different support tiers are provided. The platforms with full-fledged support (classified as tier 1) are, as of Oct, 2022:

- macOS x86-64
- Windows x86-64 and CUDA
- Linux x86-64 and CUDA
- Linux i686

Worth noting that many platforms well on their way into tier 1, such as macOS with ARMv8 (M-series chips).

Julia is distributed under the MIT license along with vast majority of the ecosystem, which guarantees free use, modification, and re-distribution for any use case.

Reproducibility

Julia includes a package manager and a general registry used by the whole community in an organized manner. In particular, each package contains a Project.toml file, that records the dependency and compatibility with other packages in an uniform way.

Furthermore, any binary dependencies are also captured by the package system: they are distributed as "Artifact"—packages with names ending _jll—but still behave as normal packages when it comes to dependency and compatibility resolution. This eliminates a few problems, including running out of pip space just because you depend on a large library (e.g., CUDA). More details are giving in the "Packaging" section.

On the end-user side, one can easily capture the an environment by working with the Manifest.toml file. While Project.toml records compatibility and dependencies, Julia would try to use the latest compatible packages when instantiating the environment. Manifest.toml, on the other hand, captures the exact versions of every package used (recursively) such that exact reproducibility can be guaranteed.

Numerical Optimization

The statistical inference procedures relevant to HEP use numerical optimization heavily, from Maximal Likelihood Estimation (MLE) to scans over Parameters of Interest (POI) and obtaining the test statistics. Traditionally this is done by minuit2 [61, 62] in ROOT, which uses the finite difference method to provide gradient information for some of its optimization.

[7] https://julialang.org/downloads/#supported_platforms.

Julia has a solid ecosystem in numerical optimization (NLopt.jl [63], Optim.jl [64], and meta algorithm package such as Optimization.jl [65] that brings local and global optimization together). Julia's performance has lead to most libraries being written in pure Julia, which means that optimization tasks can often use better algorithms such as Broyden-Fletcher-Goldfarb-Shanno (BFGS) [66–69] that rely on gradient provided by automatic differentiation. Support for automatic differentiation is further described in the "Automatic differentiation" section.

Construction of a complex probability distribution function is a common problem in HEP. Description of continuous spectra often requires a multicomponent probability density function (PDF) e.g., a sum of a signal component and a background component. In addition, the convolution with the model PDF with the experimental resolution is an essential for the HEP applications. The RooFit framework is the standard tool for building complex high-dimensional parametric functions out of lower dimensional building blocks. As great convenience, the framework provides a homogeneous treatment of the PDF variables and parameters that can be fixed, restricted to a range, or constrained by a penalty to the likelihood functions. The framework is written in C++ and available in Python. A pure-python package zfit [70, 71] give an alternative solution to Python users that can better integrate with the scientific Python ecosystem.

Julia ecosystem offers a large variety of standard density functions in the Distributions.jl package [72, 73]. The package largely exploits the properties of the standard density functions, such as moments and quantiles, which are computed using analytic expressions for the unbound PDFs. Moreover, flexible construction functionality is greatly missing. The mixture models of the Distributions.jl are the holder for the multicomponent PDF, however, they cannot be used for fitting of the component fractions, the prior probabilities. Extension of the convolution functionality beyond a small set of low-level functions is required. The management of the distribution parameters is a key missing functionality in Julia modelling ecosystem.

Specific Needs for Analysis of Reconstructed Events

Tools to Produce Histograms and Publication-Quality Plot

The statistics community in Julia has support for N-dimensional histograms with arbitrary binning in StatsBase.jl [74], an extension to this basic histogram is implemented in FHist.jl [75], which added support for bin error and under/overflow and for filling the histograms in an event loop, as typically done in HEP analyses.

Many libraries of high quality are available for plotting from Julia. In the interests of standardization, the Plots.jl [76] package provides a front-end interface to many plotting packages, allowing easy switching from one to another. It supports the concept of recipes used by packages processing data to specify how to visualize them, without depending on the Plots package: the dependency is limited to the RecipeBase.jl [77] package which has less than 350 lines of code. The package supports, currently, 7 backends. It supports themes, which are sets of default attributes and provide a similar feature to the ROOTTStyle class. The back end selected by default in Plots is GR [78], a rich visualization package providing both 2D and 3D plotting and supporting LaTeX for text. The GR package, or its GRUtils.jl [79] extension, can be used directly when a shorter warm-up time is needed before obtaining the first plot of a running session (see the "Just-in-time compilation latency" section for a discussion on the warm-up time).

We should also mention the Makie.jl ecosystem [80], a rich plotting package targeting publication-quality plots, which is increasingly popular. This package supports the recipe and theme features, but is not itself supported by Plots.jl. For instance, the FHist.jl HEP-oriented histograming package mentioned before provides a recipe to plot the histograms. Makie.jl suffers from a longer time to obtain the first plot, even larger than with the Plots.jl package with its default backend OpenGL.

Use of LaTeX to generate high-quality plots has been popularized in HEP community with the plotting system of the Rivet Monte-Carlo event generator validation toolkit [81]. The PFGPlots [82] and PFGPlotsX [83] packages offer LaTeX-based plotting. They are both supported by the Plots.jl package. The Gaston.jl [84] package provides plotting using the popular Gnuplot.jl utility [85].

People used to the Python matplot.pyplot set of functions [86] can use the PyPlot.jl package that provides a Julia API to this package. Those who prefer plotly to matplot, can use the PlotlyJS.jl, a Julia interface to plotly. The high-level grammar of interactive graphics Vega-Lite [87] is also supported, thanks to the VegaLite.jl [88] package that supports exports to bitmap and vector image files, including the PDF format, which is convenient to include in papers written with LaTeX. Plotting can also be done on a text terminal, using the UnicodePlots.jl [89] package, supported by the Plot front end.

The visualization tool ecosystem for Julia is rich, with the added benefit of staying in the same environment as the analysis and enabling an interactive workflow.

Notebook Support

A computational notebook is an interface for literate programming that allows embedding calculations within text. Notebooks have been made popular by Mathematica [90], which has supported notebooks starting from its first version, 1.0, released in 1988. In HEP, notebooks are widely used by theoreticians for symbolic calculation e.g., with Mathematica, and by experimentalists, for data analysis, and plotting using Python or C++ as programming language.

The notebook system used with Python, Jupyter, fully supports Julia. The "ju" of **Ju**pyter stands for Julia, while "py" stands for Python and "er" for the R language. The `ROOT` analysis framework brings C++ support to Jupyter.

The notebook support for Julia is richer than for Python and C++. In addition to Jupyter, `Pluto.jl` [91] provides a new-generation notebook system for Julia. This system keeps track of the dependency of all calculations spread in the document and updates automatically any dependent results when a one of them is edited. Beyond being convenient, this automatic update provides *reproducibility*.

`Pluto.jl` is also a very easy solution for interactive notebooks, where buttons, drop-down menus and slides can be included. This is useful for students. It can also be used to build a tool for experiments running shifters to analyze the data quality in quasi-realtime.

With `Pluto.jl`, notebooks are normal executable Julia files. Notebook functionality is offered through special comments. This helps with version control.

Specific Needs for Physics Event Reconstruction, Simulation and Data Acquisition Trigger Software

Physics event reconstruction, simulation and trigger software are typically large codes developed by the experiment and project collaborations. The software stack of the LHC experiments is particularly large and complicated, due to the complexity of their detectors. The software is developed collaboratively by many developers, with different levels of software skills. Tools for both collaborative development and quality assurance are essential for all experiment software. Software distribution and release management are also important. The complexity of the C++ language, used in most of these frameworks, can limit the integration of contributions developed by students. This is more and more true given the growing use of high-level language (e.g., Python) as the teaching language for computing in universities, especially among natural science departments.

The Julia language and its ecosystem have been built using an open-source and community approach. Tools have been put in place and are widely adopted for efficient collaborative development. Julia comes with a standard and convenient package management system providing reproducibility, see the "Packaging" section Julia has built-in unit testing, coverage measurement, and officially maintained continuous integration recipes and documentation generator. These are used in almost all of the Julia packages registered publicly, thanks to the streamlined experience and low barrier to entry.

The simulation software of the experiments depends on external libraries to simulate the underlying physics, such as Monte Carlo event generators, and on some others, like Geant4 [92], to simulate the transport of the produced particles and their interaction with the detector. Interoperability with libraries written in C, C++, or Fortran, as discussed in the "Interoperability with legacy code" section, it is essential not to have to re-write all the external libraries in Julia.

Simulation and reconstruction is compute intensive and therefore good performance is essential: performance has a direct impact on the computing infrastructure cost. We have seen in the previous section that Julia meets the C/C++ performance and sometimes surpasses it. Code parallelization and efficient use of single instruction multiple data (SIMD) vectorization features of CPUs is essential at the LHC and for HL-LHC to efficiently use current hardware resources, with a high density of computing cores, including accelerators (e.g., GPU) that can count tens of thousands core [6]. The Julia language provides a very good support for multi-threading: a loop can be parallelized by a single macro (`@threads`), an operation can be made atomic by prefixing it with `@atomic`, a more general lock mechanism is provided, asynchronous tasks, with distribution of tasks to different threads, is natively supported. Julia supports distributed computing, using its own communication mechanism but also using MPI [93, 94]. It is possible to use Julia's compiler to vectorize loops by using the `@simd` macro or the more advanced `@turbo` from the `LoopVectorization.jl` package [95].

Due to its effective metaprogramming capabilities, Julia has great support for running code on heterogeneous architectures, Julia code can be compiled for Nvidia (CUDA), AMD (ROC) and Intel (oneAPI) GPUs via compiler written in Julia[8], without dependency on, for example, C++ CUDA or HIP library. Packages like `GPUArrays.jl` and `KernelAbstractions.jl` allow the use of exactly the same core algorithm written in Julia to be executed across different vendor platforms with minimal boilerplate code, which is a currently a unique feature among languages.

[8] https://github.com/JuliaGPU/GPUCompiler.jl.

On the more user-facing front, libraries such as `Tullio.jl` [96] combine metaprogramming and kernel programming ability to allow users to express tensor operation with Einstein notations regardless of whether the array lives on RAM or GPU VRAM. This is very relevant for data preservation and for unifying effort to write algorithms once and run them everywhere.

The ability to run native Julia code on both CPUs and GPUs, combined with the support for automatic differentiation in Julia, makes Julia an excellent platform for machine learning (ML) research. This is especially true for advanced scientific machine learning that goes beyond combining conventional matrix-crunching ML-primitives/layers and uses physical/semantic models or mixes them with generic ML constructs.

Specific Needs for Event Generation and for Phenomenology

Symbolic Calculations in Julia

Julia is a fast, solid and reliable programming language with a well developed Computer Algebra System (CAS) such as `Symbolics.jl` [97], a language for symbolic calculations such as `Symata.jl` [98], and an interface to `Mathematica` such as `MathLink.jl` [99], that could be widely used in HEP, considering the advantages Julia has.

`Symbolics.jl` [100] is a CAS written in pure Julia, which is developed by the SciML community [101] who also maintain the state-of-the-art differential equations ecosystem [102]. The package has scalable performance and integrates with the rest of Julia ecosystem thanks to its non-OOP design and multiple dispatch [97]. Some of the main features of `Symbolics.jl` include pattern matching, simplification, substitution, logical and boolean expressions, symbolic equation solving, support for non-standard algebras with non-commutative symbols, automatic conversion of Julia code to symbolic code and generation of high performance and parallel functions from symbolic expressions [103], which make it even more interesting for possible applications in HEP. At the heart of `Symbolics.jl`, we find `ModelingToolkit.jl`, a symbolic equation-based modeling system [104], and `SymbolicUtils.jl`, a rule-based rewrite system [105].

`Symata.jl` [98] is a language for symbolic computations in which some features, such as evaluation, pattern matching and flow control, are written in Julia, and symbolic calculations are developed by wrapping `Sympy`, a python library for symbolic mathematics.

`MathLink.jl` [99] is a Julia language interface for the Wolfram Symbolic Transfer Protocol (WSTP) (this requires the installation of Mathematica or the free Wolfram Engine to run properly). The interface is a `W""` string macro used to define Mathematica symbols. `MathLinkExtras.jl` [106] adds extra functionalities such as `W2Mstr`, which allows the conversion of Julia MathLink expressions into `Mathematica` expressions, and`W2Tex` which converts Julia `MathLink.jl` expressions into LaTeX format. And, finally, one can evaluate the expression in `Mathematica` using `weval`.

Event Generators

To be prepared for future needs for event generation [107], it is conceivable to rewrite parts of the existing event generators in Julia and making use of modern parallelisation technologies. One of the most demanding tasks in event generation is the evaluation of matrix elements and cross sections, where Julia provides several useful tools.

The package `Dagger.jl` is a framework for out-of-core and parallel computing written in pure Julia. It is similar to the python library `Dask` and provides a scheduler for parallelized execution of computing tasks represented as a directed acyclic graphs (DAGs). Such DAGs could be used to represent the evaluation of matrix elements in terms of elementary building blocks, similar to `HELAS`-like functions in `Madgraph4GPU` (see e.g., [108]). Furthermore, `Dagger.jl` supports the selection of different processors as well, making it possible to be use for distributed computing on GPU as well (see e.g., `DaggerGPU.jl` [109]).

Table 4 Summary of features needed for HEP applications and their availability in the Julia ecosystem

Requirement	Fulfilled by Julia
Easy to learn for HEP physicists	✓
Performance	✓
Interoperability with legacy code	✓
Support for HEP standard formats	Partial
Support for architectures and open license	✓
Cross-platform reproducibility	✓
Tools to perform optimization/minimization	✓
Histogramming	✓
Plotting with publication quality	✓
Notebook support	✓
Tooling for large project (unit tests, continuous integration, software distribution)	✓
SIMD and multi-threading	✓
Distributed parallel computing	✓
Native GPU support	✓
Machine learning libraries	✓
Computer Algebra System	✓

Feature Summary

The Table 4 summarizes the programming language and ecosystem features we have identified as required for HEP. It is surprising how Julia language manages to fulfill almost all of these features. We should note that the interoperability is less transparent for C++ than with the other languages as it requires to write a code wrapper. Many HEP file format are already supported, including for `ROOT` files, without covering the full versatility offered by this format.

The Bonuses

In addition to solving the two-language problem and the mandatory features listed in the previous section, the Julia ecosystem will provide other advantages over the C++ and Python languages currently used.

Packaging

Julia comes with a built-in package manager, `Pkg.jl`. It builds on the experience of package managers in other ecosystems, and it can be used to install packages and manage "environments", similar to the concept of virtual environments in Python. A Julia environment is defined by two files:

- `Project.toml`: this file records version and UUID of the current project, it also contains the list of direct dependencies of this project, as well as the compatibility bounds with these packages and Julia itself. Moreover, all Julia packages follow semantic versioning (semVer [110]): version numbers are composed of three parts, major, minor and patch numbers, and breaking changes can only be introduced in versions which change the left-most non-zero component (e.g., going from 0.0.2 to 0.1.0, or from 2.7.3 to 3.0.0 are considered breaking changes).
- `Manifest.toml`: this file is automatically generated by the package manager when instantiating an environment, if not already present, and it captures all packages in the environment with their versions, including all indirect dependencies. When used together with `Project.toml`, `Manifest.toml` file describes an exact environment that can be recreated on any other platforms, which is particularly important for reproducibility of applications (e.g., analysis pipelines).

Julia packages are organized in directories (usually also Git repositories) in which there is a `Project.toml` file to define its environment. Packages can be installed either via path (local path on a machine, or URL to a remote Git repository), or by name if they are present in a package registry. `Pkg.jl` is able to deal with multiple registries at the same time, which can be both public and private, in case there is a need to provide packages relevant only to a single group or collaboration.

Because there is only one package manager and only one way to record package dependency, the Julia package registry simply records the dependency and compatibility metadata separately from package's source code. It allows a local resolver to correctly resolve compatibility in a short amount of time.

Users can interact with the package manager either by using its programmatic API (useful for scripting) or an interactive REPL mode, which can be entered by typing the closing square bracket `]` in Julia's REPL. The package manager can also be used to run the tests for packages with a single command (`Pkg.test` using the API, or the command `test` in the REPL). Since `Pkg.jl` is a standard library and has many capabilities, all users are familiar with it and do not need to resort to third-party, mutually incompatible tools.

Multiple Dispatch and Fast Development

We group multiple dispatch and fast development (i.e., hot reloading) into the same section because of their direct impact on design of the packages and the quality of life of developers.

A side-by-side comparison between OOP and multiple dispatch has been shown in the "Polymorphism in C++, Python, and Julia" section. Here, we highlight that multiple dispatch is also a known solution to the *expression problem* [111, 112]. Essentially, in class-based OOP, one of the following is much less natural than the other:

- add new methods to existing data type (class/struct).
- add new data type (class/struct) for existing algorithm (method).

In OOP, the second one is easy, think inheritance; but the first requires access to source code. In Julia, the first one is trivial since methods do not bind to data type (class/struct) to begin with. But the second one can also be easily done by sub-typing the upstream abstract type.

Making it easy for developers to reuse existing packages is crucial in HEP also because libraries are sometimes under-maintained. If we can cleanly extend and reuse these libraries without making private forks, overall efficiency would be boosted.

As a dynamic language, hot reloading should not come as a surprise. It is, in fact, crucial for Julia, due to the inevitable latency introduced by the JIT compiler. The go-to package for this is `Revise.jl` [113] which automatically detects file modifications and re-compiles the relevant functions on the fly. It can also reload the source code of any Julia Base module, saving a lot of time if (re)compiling Julia.

Automatic Differentiation

The multiple dispatch system and the native speed of Julia eliminated the need for many specialized libraries to implement the same interface (e.g `NumPy`-interface in `JAX` [114], `TensorFlow` [115], `PyTorch` [116]). Instead, package maintainers only have to focus on providing rules for the built-in functions as they are fast already and downstream packages are mostly pure-Julia too, see `ChainRules.jl` [117]. A dedicated organization, JuliaDiff [118] collects all the packages and efforts regarding what each autodiff engine is good at.

Foundation HEP-Specific Libraries to be Developed or Consolidated

Integration in the ROOT Framework

Because of the ubiquity of `ROOT` in HEP, a Julia interface to this framework, similar to the existing Python one and that will allow people familiar to it to find their way easily, is essential for the development of Julia in the HEP community. In addition, this will provide access to a large set of software used in HEP (storage support, statistics tools, unfolding, etc.) before their counterpart are implemented in Julia.

HEP-Specific Data Storage Format

It will be important to consolidate the support for the `ROOT` data format. The `ROOT` data format is very versatile and allows the storage of instances of arbitrary C++ classes (this is true of the current `TTree` and the new `RNTuple` format). Current Julia packages for `ROOT` I/O do not cover this whole versatility and do not allow for reading and writing files with objects of sophisticated types.

Physics Object Types and Histogram

Packages to manipulate Lorentz vectors and to build histograms are already available [75, 119]. Leveraging multiple dispatch, these packages are relatively easy to implement, and compose well with rest of the Julia ecosystem (e.g., collection of 4-vectors can be stored and sorted efficiently without any special care). Defining a standard interface to Lorentz vectors and histogram data structures, with a similar approach as the Table interface [120] could be beneficial.

HEP Specific Statistical Tools

Over years, HEP community has developed its statistical standard to assert a level of confidence of the experimental results, for measurements, limits and observation of new phenomena. The Julia ecosystem contains several high-quality packages for Bayesian statistics and inference. Two examples are `BAT.jl` (Bayesian Analysis Toolkit in Julia) `BAT.jl` [121] and `Turing.jl` [122], which already have been used in several physics analyses. Both packages are being actively developed with good communication of the authors across the development teams. Common interfaces [123–125] have been established to increase interoperability.

More development is required for the frequentist CLs approach used at LHC [126–129] and based on profiled likelihood fits. The method is derived from the hybrid method of the same name developed at LEP [130, 131] and used later at Tevatron [132, 133]. The C++ tools typically used by LHC experiments are the `RooFit` (originally developed for the BaBar experiment [134, 135]) and `RooStats` libraries included in the ROOT framework. For multinodal distributions these libraries are used through the `HistFactory` [136] or `HiggsCombine` [127, 137, 138] tools. The `pyhf` package [139] provides a pure-Python implementation of `HistFactory` that offers different computational backends to perform the likelihood maximization and is gaining popularity. The `HistFactory`, `HiggsCombine`, and `pyhf` are standalone tools, for which inputs are provided in text files (XML or JSON). Thanks to the transparent Julia-Python interface, `pyhf` can also be used in a Julia session or code. For a perfect integration and to exploit the language performance, a Julia implementation is desirable. An effort to implement `pyhf` in Julia has already started [140] and would need to be consolidated.

Histogram unfolding [141] is another statistical tool widely used in HEP experiments. It is used to correct from the effect of the finite resolution of the particle detectors in differential cross section measurement. The `TUnfold` [142] and `RooUnfold` [143] are the most commonly used packages. The `RooFitUnfold` [144] package provides an extension of `RooUnfold`. New techniques to perform unbinned high-dimensional data unfolding has been recently developed [145]. Like for CLs, unfolding comes at the last step of a HEP data analysis, and a Julia implementation would be useful.

Limits of the Julia Programming Language

Language Popularity

Despite its smaller user base than C++ and Python, we have found that it is extremely easy to find information on the web, either from Stack Overflow or from dedicated channels, on Discourse, Slack, and Zulip. The community is very collaborative. An annual conference JuliaCon[9] is boosting this collaboration. In particular, it encourages exchanges between different fields, both from Academia and Industry. The popularity of Julia is growing and it has been adopted by large academic projects, like the Climate Modeling Alliance (CliMA); and companies like ASML, the largest supplier of photolithography systems; Pharmacology actors like Pfizer, Moderna, and AztraZeneca [146–148]; finance actors like Aviva, one of the largest insurers, and the Federal Reserve Bank of New York [149–151].

Just-In-Time Compilation Latency

While applications written in Julia run faster than with an interpreted language, the first execution requires additional time to perform the just-in-time (JIT) compilation. In order to limit this overhead, the intermediate results of the compilation, called precompiled code, is cached on disk. The precompilation of a package code is typically performed in parallel at installation time, and the cached content includes, but is not limited to: lowered code, type inference result, etc.; but at the time of writing, Julia does not yet cache compiled machine code[10]. The latency is often called "time-to-first-plot".

The JIT compilation latency has been improved from version-to-version, in particular with versions 1.5, 1.6, 1.8, and 1.9 by reducing the number of required recompilations. The various sources of latency have been studied extensively [152, 153] and the reduction of the time-to-first-plot is a high priority for the compiler team. Besides improvements coming from the compiler, following the general guidelines of Julia code style for performance [40], which ensure that the compiler can easily infer variable types, should reduce such latency [152]. At the same time, tools have been developed to both help "hunt" down unnecessary recompilation (`SnoopCompile.jl` [154]), as well as help precompile known common user routinges at installation time (`PrecompileTools.jl` [155]).

[9] https://juliacon.org/.

[10] Progress is being made, see https://github.com/JuliaLang/julia/pull/44527.

The latency can also be drastically reduced by preparing a custom system image: the system image contains cached machine code for a set of precompiled packages and past executions. It comes with the drawback that versions of the packages shipped in the system image take precedence over the ones installed via the package manager [156], which can be confusing and be a source of bugs. Updating these packages requires rebuilding the custom system image.

The time to produce a first plot, consisting of a 2-D plot of 100 points, was measured to be 2.09 ± 0.01s with Julia 1.9.0-rc1 and the `Plots.jl` [76] package. The `Makie.jl` package took 7.57 ± 0.02s using the Cairo backend. Time is similar with the GL backend. Subsequent plots take less than a millisecond. Building a custom system image brings down the latency to below 50 ms for both packages. While for `Plots.jl`, the latency using the standard system image is acceptable, building a custom system image would make use of `Makie.jl` for an interactive session or for a short batch script much more convenient. To measure the improvement brought by the efforts of Julia developers, the measurement is repeated with the older long-term-support release 1.6.7. With this older release the result is 29.0 ± 0.1s for `Makie.jl`, showing an improvement larger than a factor of 4.

The start-up time could be a concern for large HEP experiment simulation and reconstruction software. As an example of software size, the CMS experiment software, CMSSW [157], totals more than 2 million of lines of C++ code. The assessment was done with release 12.3.5 and the number of lines of code was defined as the number of semicolons contained in the code. In lieu of a similar sized HEP software package written in Julia, we have measured the start-up time on the relatively large package `OrdinaryDiffEq`, using its version 6.49.4. The package consists of about 125,000 lines of Julia code, excluding comments, and 390,000 when including the external packages. The lines of code have been counted with the `Tokei` software [158] version 12.1.2. and the extra time to run the Example 1 of the manual [159] the first time, compared to subsequent executions, was 5.91 ± 0.01 s. It goes down to 826 ± 2 ms when using a custom system image. We should also note that the precompilation happening on package installation for the package and its dependencies (120 packages) took 256 s only.

For a large experiment software framework, attention will need to be paid to limit code invalidation by respecting the guidelines to ease type inference. This will also help the compiler to optimize the code. While minimizing start-up time may require some effort for large HEP project, we do not expect it to be a show stopper. At worst, it will require to use custom images, with a conciliation on the package management. In addition, development to improve the start-up time is on-going and we should expect significant progress in the near future [160].

Application Programming Interface Specification

Julia lacks a single standard to define the application programming interface (API) of a package. The one with the best support is the use of the `export` directive to list the symbols exposed to the user. The directive is recognized by the language's introspection functions. The `names` function lists, by default, the exported symbols, with an option to list all symbols. The `methodswith` function, used to retrieve functions with an argument of a given type will list only functions from the export list.

Nevertheless, the `export` directive has the side-effect that all the public symbols end up in the user's namespace, if the package is imported with the `using` statement. To quote the Julia manual [40], "it is common to export names which form part of the API. [...] Also, some modules don't export names at all. This is usually done if they use common words, such as derivative, in their API, which could easily clash with the export lists of other modules.".

The Julia language itself uses the user manual to define the API, as explained in the "Frequently asked questions" of this document [40]. With such an approach, we lose the benefit of the introspection functions, themselves agnostic to the API information.

A built-in `unexport` directive, that would allow listing public symbols that the `using` statement must keep in the module namespace and which would be recognized by the introspection functions and also by the documentation generator [161], would be very beneficial.

Training and Language Transition Support

Julia has been successfully introduced into existing teams, gradually replacing their C++ with Julia packages over time, for example in the LEGEND and BAT groups at the Max-Planck-Institute for Physics. Julia is also the official secondary language (after Python) of the whole LEGEND [162] collaboration.

Observed experience is that students with a basic programming background (e.g., in Python or C++) do learn the language very quickly and become productive after just a few days. After exposure to the language for a few weeks, students are typically able to make contributions to larger software packages as well. No problems have been found using Julia for short-term thesis work (e.g., three-month bachelor theses) and even two-week internship, with students and interns who were new to the language. The reaction of these students has been uniformly positive.

Master and PhD theses that used Julia as the primary language have resulted in very positive experience for both students and supervisors. Students who use Julia in longer-term projects not only become very proficient in the language, but also gain a lower-level understanding of computing, data structures and performance implications of modern hardware in general, compared to students who work in Python. This is because Julia makes it very easy the move between higher-level and lower-level programming, in contrast to the Python-plus-C++ two-language approach.

More code reuse and transfer has been observed across student generations in Julia, compared to C++. This is due to the combination of an excellent package management with the use of multiple dispatch as a foundation. The first simplifies the maintenance of systems consisting of smaller and more modular packages, while the second solves the *expression problem*.

Conclusions

The Julia programming language has been presented and compared with C++ and Python. To study the potential of Julia for HEP, a list of requirements for offline and software-based-trigger HEP applications, covering both the language and its ecosystem, has been established. The compatibility of Julia with these requirements has been studied. Julia and its ecosystem are impressively fulfilling all these requirements. Moreover, Julia brings other features—integrated packaging system with reproducibility support, multiple dispatch and automatic differentiation—from which HEP applications would benefit.

The capacity to provide, at the same time, ease of programming and performance makes Julia the ideal programming language for HEP data analysis and is more generally an important asset for all the considered HEP applications. The dynamic multiple dispatch paradigm of Julia has proven to ease code reuse. This property will greatly benefit HEP community applications that involve code developed by many people from many different groups.

Using a single and easy programming language will facilitate training. Experience has shown students with either a C++ or Python background learn the language very quickly, being productive after a few days. Using Julia as mainstream language in a collaboration allows students on short-term projects to use the common programming language, while in case of C++, using a simpler language as Python is often needed. This eases the reuse of the code developed in such context.

We have measured the performance provided by the language in the context of HEP data analysis. The measurements show excellent runtime performance, competitive with C++: 11% faster for the simple LHC event analysis example used as benchmark. When compared to Python, in addition to being faster, it is much less sensitive to implementation choices. The Python implementation was shown to be three orders of magnitude slower than Julia when the event loop is performed in Python. Vectorization techniques can be used to move the event loop by using underlying compiled libraries and this reduces the gap in performance.

One difference with C++ and Python is that Julia is younger and has a smaller community. The Julia community is very collaborative and, despite its lower popularity, information for developing with this language is easy to find on the Internet. Julia's rapid growth in academia and industry gives us confidence on the long term continuity of the Julia language, which is essential for HEP projects, because of their large time span.

In view of this study, the HEP community will definitively benefit from a large scale adoption of the Julia programming language for its software development. Consolidation of HEP-specific foundation libraries will be essential to ease this adoption.

Appendix A

Polymorphism in C++ and Julia Illustrated in Code

Differences of polymorphism support in C++, Julia, and `Python` are discussed in the "Polymorphism in C++, Python, and Julia" section. This appendix provides code examples illustrating the discussion.

Static ad-hoc function polymorphism can be implemented in C++ using two different paradigms, function overriding and templates. We will illustrate this with an example. Let us consider two classes, `A` and a derived class `AChild`, and a global function `f()`. Using the function overriding paradigm, the ad-hoc polymorphism on the function argument can be implemented as,

```cpp
#include <iostream>

struct A{};
struct AChild: public A{};
struct B{};

void f(A a){
  std::cout << "A\n";
}

void f(B b){
  std::cout << "B\n";
}
```

The same ad-hoc polymorphism can be implemented in the template paradigm as follows.

```cpp
#include <iostream>
#include <concepts>

template<typename C, typename P>
concept Derived =
     std::is_base_of<P, C>::value;

struct A{};
struct AChild: public A{};
struct B{};

template<typename T> void f(T x){}

template<Derived<A> T>
void f(T a){
  std::cout << "A\n";
}

template<>
void f(B b){
  std::cout << "B\n";
}
```

Both implementations can be tested with the following code, which will result in the same output.

```cpp
A a;
AChild aChild;
B b;

int main(){
  f(a); //prints A
  f(b); //prints B
  f(aChild); // prints A
  return 0;
}
```

Output:

```
A
B
A
```

While in C++, an ad-hoc function polymorphism can be implemented using two different paradigms, the multiple dispatch feature of the Julia language provides a single and consistent way to implement polymorphism. The Julia implementation looks like the following.

```
abstract type AbstractA end
abstract type AbstractAChild <: AbstractA end

struct A <: AbstractA end
struct AChild <: AbstractAChild end
struct B end

function f(a::AbstractA)
  println("A")
end

function f(b::B)
  println("B")
end

a = A()
aChild = AChild()
b = B()

f(a) #prints A
f(b) #prints B
f(aChild) #prints A
```

In C++, the dispatch on argument type can be static or dynamic for the class instance argument (`this`) and is always static for the other arguments. This situation can be intricate as in the example below, where the selection of a static or dynamic dispatch over the `this` argument depends on the type of the other argument.

```
#include <iostream>
struct A{

  virtual void f(int x) const {
    std::cout << "A::f(int)\n";
  }

  void f(const char* x) const {
    std::cout << "A::f(const char*)\n";
  }
};

struct B: public A{
  void f(int x) const {
    std::cout << "B::f(int)\n";
  }

  void f(const char* x) const {
    std::cout << "B::f(const char*)\n";
  }
};

int main(){
  B b;

  A& a = b;

  a.f(1);  //prints B::f(int)
  a.f(""); //prints A::f(const char*)
}
```

The C++ language includes an implicit type conversion when copying an object. This feature can lead to confusion, as illustrated in the code below.

```
#include <iostream>

struct Animal{
  virtual void f() const {
    std::cout << "Animal\n";
  }
};

struct Lion: public Animal {
  void f() const {
    std::cout << "King of Savannah\n";
  }
};


void g(const Animal& a){
  a.f();
}

void h(Animal a){
  a.f();
}


int main(){
  Lion leo;

  g(leo); //prints King of Savannah
  h(leo); //prints Animal

  return 0;
}
```

This pitfall does not exist in Julia. There is a single way to pass arguments, "call by sharing" [163], which does not copy the arguments.

```
abstract type Animal end

struct Lion <: Animal
end

function f(x::Animal)
  println("Animal")
end

function f(x::Lion)
  println("King of Savannah")
end

function g(x)
  f(x)
end

leo =  Lion()
g(leo) # prints "King of Savannah"
```

The lion is grateful to Julia for honoring his title.

Acknowledgements The authors would like to thank Enrico Guiraud (Princeton University and CERN) for the review of the C++ code used in the dimuon analysis benchmark and fruitful discussions.

Endorsement Johannes Blaschke (NERSC, Lawrence Berkeley National Laboratory); Ankur Dhar (SLAC National Accelerator Laboratory); Matthew Feickert (University of Wisconsin–Madison, Madison; Sam Foreman (Fakultät Physik, Technische Universität Dortmund); Cornelius Grunwald (Massachusetts Institute of Technology); Alexander Held (University of Wisconsin–Madison, Madison), Philip Ilten (University of Cincinnati), Adam L. Lyon (Fermi National Accelerator Laboratory), Mark Neubauer (University of Illinois Urbana-Champaign); Ianna Osborne (Princeton University), Johannes Schumann (Erlangen Centre for Astroparticle Physics, Friedrich-Alexander-Universit¨at Erlangen-N¨urnberg); Daniel Spitzbart (Boston University, Boston); James Simone (Fermi National Accelerator Laboratory), Rongkun Wang (Harvard University, Cambridge); Michael Wilkinson (University of Cincinnati); and Efe Yazgan (National Taiwan University) from the High Energy Physics and Astrophysics communities endorse this work. The authors thank them for their support.

Author Contributions All the authors contributed to the investigations of the features of the Julia programming language and its adequacy with the HEP application requirements. Manuscript writing has been shared between the authors and coordinated by P. Gras. Performance measurements were performed by J. Ling and P. Gras. All the authors have reviewed the manuscript.

Funding Open access funding provided by CERN (European Organization for Nuclear Research)

Declarations

Conflict of Interest The authors declare no competing interests.

References

1. Bird I et al (2014) Update of the computing models of the WLCG and the LHC experiments. Technical Report CERN-LHCC-2014-014, CERN. https://cds.cern.ch/record/1695401
2. Collaboration A (2022) ATLAS Software and Computing HL-LHC Roadmap. Technical Report CERN-LHCC-2022-005, LHCC-G-182, CERN, Geneva. http://cds.cern.ch/record/2802918
3. Software CO (2022) Computing. CMS phase-2 computing model: Update document. Technical Report CERN-CMS-NOTE-2022-008, CERN, Geneva. http://cds.cern.ch/record/2815292
4. Evans L, Bryant P (2008) LHC machine. JINST 3:S08001. https://doi.org/10.1088/1748-0221/3/08/S08001
5. Apollinari G et al (2017) High-luminosity large hadron collider (HL-LHC): technical design report V. 0.1. Technical Report CERN-2017-007-M, CERN. https://doi.org/10.23731/CYRM-2017-004
6. Albrecht J et al (2019) A roadmap for HEP software and computing R &D for the 2020s. Comput Softw Big Sci 3:7. https://doi.org/10.1007/s41781-018-0018-8
7. Sexton-Kennedy E (2018) HEP software development in the next decade; the views of the HSF community. J Phys Conf Ser 1085:022006. https://doi.org/10.1088/1742-6596/1085/2/022006
8. Vassilev V, Canal P, Naumann A, Moneta L, Russo P (2012) Cling–the new interactive interpreter for ROOT 6. J Phys Conf Ser 396:052071. https://doi.org/10.1088/1742-6596/396/5/052071
9. Brun R, Rademakers F (1997) ROOT: an object oriented data analysis framework. Nucl Instrum Meth A. 389:81–86. https://doi.org/10.1016/S0168-9002(97)00048-X
10. Antcheva I et al (2009) ROOT–A C++ framework for petabyte data storage, statistical analysis and visualization. 40 YEARS OF CPC: a celebratory issue focused on quality software for high performance, grid and novel computing architectures. Comp Phys Commun 180:2499–2512. https://doi.org/10.1016/j.cpc.2009.08.005
11. Bezanson J, Edelman A, Karpinski S, Shah VB (2017) Julia: a fresh approach to numerical computing. SIAM Rev 59:65–98. https://doi.org/10.1137/141000671
12. Bezanson J et al (2018) Julia: dynamism and performance reconciled by design. Proc ACM Program Lang. https://doi.org/10.1145/3276490
13. Bezanson J, Karpinski S, Shah VB, Edelman A (2012) Why we created Julia. https://julialang.org/blog/2012/02/why-we-created-julia/
14. Julia Computing (2022) Newsletter january 2022–julia growth statistics. https://juliacomputing.com/blog/2022/01/newsletter-january/
15. Stanitzki M, Strube J (2021) Performance of Julia for high energy physics analyses. Comput Softw Big Sci 5:10. https://doi.org/10.1007/s41781-021-00053-3
16. Go 2 error handling feedback. https://github.com/golang/go/wiki/Go2ErrorHandlingFeedback. Accessed 12 Oct 2021
17. Rackauckas C (2021) ModelingToolkit, modelica, and modia: the composable modeling future in Julia. Winnower. https://doi.org/10.15200/winn.162133.39054
18. Julia Computing NVIDIA Julia computing brings support for NVIDIA GPU computing on Arm powered servers. https://juliacomputing.com/blog/2019/12/nvidia-ngc-arm. Accessed 12 Oct 2021
19. Regier J et al (2019) Cataloging the visible universe through Bayesian inference in Julia at Petascale. J Parallel Distrib Comput 127:89–104. https://doi.org/10.1016/j.jpdc.2018.12.008
20. Claster A, Julia Joins Petaflop Club. https://juliacomputing.com/media/2017/09/julia-joins-petaflop-club. (accessed October 12, 2021)
21. The LuaJIT project website. https://luajit.org. Accessed 7 Apr 2022
22. Bolz CF, Tratt L (2015) The impact of meta-tracing on VM design and implementation. Sci Comp Program 98:408–421. https://doi.org/10.1016/j.scico.2013.02.001
23. Pypy project website. https://www.pypy.org/. Accessed 7 Apr 2022
24. Stroustrup B (2018) A Tour of C++ C++ in-depth series. Pearson Education

25. Stroustrup B (1994) The design and evolution of C++. Pearson Education
26. Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism. ACM Comput Surv 17:471–523. https://doi.org/10.1145/6041.6042
27. Strachey C (2000) Fundamental concepts in programming languages. High Order Symb Comput 13:11–49. https://doi.org/10.1023/A:1010000313106
28. Snyder A (1986) Encapsulation and inheritance in object-oriented programming languages. SIGPLAN Not 21:38–45. https://doi.org/10.1145/960112.28702
29. Gamma E, Helm R, Johnson RE, Vlissides J (1995) Design Patterns: Elements of Reusable Object-Oriented Software Addison-Wesley Professional Computing Series, Reading: Addison-Wesley. https://www.safaribooksonline.com/library/view/design-patterns-elements/0201633612/
30. Python multid-dispatch module (2022) https://multiple-dispatch.readthedocs.io. Accessed 25 Mar 2022
31. Zappa Nardelli F et al (2018) Julia subtyping: a rational reconstruction. Proc ACM Program Lang. https://doi.org/10.1145/3276483
32. Harris CR et al (2020) Array programming with Numpy. Nature 585:357–362. https://doi.org/10.1038/s41586-020-2649-2
33. Gras P (2012). Analysis of the di-muon spectrum using data from the CMS detector taken in 2012. https://doi.org/10.7483/OPENDATA.KS4A.BD5W
34. Wunsch S (2019). Analysis of the di-muon spectrum using data from the CMS detector taken in 2012. https://doi.org/10.7483/OPENDATA.CMS.AAR1.4NZQ
35. Dataframes.jl package documentation (2022) https://dataframes.juliadata.org. Accessed 1 Aug 2022
36. Gál T, Ling JJ, Amin N (2022) UnROOT: an I/O library for the CERN ROOT file format written in Julia. J Open Source Softw 7, 4452. https://doi.org/10.21105/joss.04452
37. Pivarski J et al (2017) Uproot. https://doi.org/10.5281/zenodo.4340632
38. Pivarski J et al (2018) Awkward array Zenodo. https://doi.org/10.5281/zenodo.7079705
39. Reback J et al (2022) pandas-dev/pandas: Pandas 1.4.4. Zenodo. https://doi.org/10.5281/zenodo.7037953
40. The Julia language manual (2021) https://docs.julialang.org/en/v1/. Accessed 29 Sep 2021
41. Janssens B CxxWrap code repository. https://github.com/JuliaInterop/CxxWrap.jl. Accessed 17 Mar 2022
42. Abrahams D, Grosse-Kunstleve RW (2003) Building hybrid systems with Boost.Python. https://www.boost.org/doc/libs/1_80_0/libs/python/doc/html/article.html. Accessed 1 Aug 2022
43. Pybind11 code repository. https://github.com/pybind/pybind11. Accessed 1 Aug 2022
44. Gras P Automatic generation of c++–julia bindings. https://github.com/grasph/wrapit. Accessed 17 Mar 2022
45. Fischer K et al Cxx.jl code repository. https://github.com/JuliaInterop/Cxx.jl. Accessed 17 Mar 2022
46. The linear collider I/O framework code repository (2022) https://github.com/iLCSoft/LCIO. Accessed 1 Aug 2022
47. Behnke T et al (2013) The international linear collider technical design report–volume 1: executive summary. Technical Report, The International Linear Collider. http://arxiv.org/abs/1306.6327
48. Cacciari M, Salam GP, Soyez G (2012) Fastjet user manual. Eur Phys J C 72:1896. https://doi.org/10.1140/epjc/s10052-012-1896-2
49. Dispelling the n^3 myth for the k_t jet-finde. Phys Lett B 641:57–61. https://doi.org/10.1016/j.physletb.2006.08.037
50. Alwall J et al (2007) A standard format for les Houches event files. Comp Phys Commun 176:300–304
51. LHEF.jl (2021) https://github.com/JuliaHEP/LHEF.jl. Accessed 29 Sep 2021
52. Strube J, Saba E, TagBot J (2021) jstrube/lcio.jl: v1.9.2. Zenodo. https://doi.org/10.5281/zenodo.4560484
53. UpROOT.jl library code repository (2021) https://github.com/JuliaHEP/UpROOT.jl. Accessed 29 Sep 2021
54. Blomer J, Canal P, Naumann A, Piparo D (2020) Evolution of the ROOT Tree I/O. EPJ Web Conf 245:02030. https://doi.org/10.1051/epjconf/202024502030
55. Foundation, T. A. S. Apache arrow. https://arrow.apache.org/. Accessed 1 Aug 2022
56. ClusterManagers package code repository. https://github.com/JuliaParallel/ClusterManagers.jl. Accessed 29 Sep 2021
57. Dask library web site (2021) https://docs.dask.org. Accessed 29 Sep 2021
58. Dagger.jl package code repository. https://github.com/JuliaParallel/Dagger.jl. Accessed 29 Sep 2021
59. Litzkow M, Livny M, Mutka M (1988) Condor—a hunter of idle workstations, IEEE, pp 104–111. https://doi.org/10.1109/DCS.1988.12507
60. HTCondor software website (2021). https://htcondor.org/. Accessed 29 Sep 2021
61. James F, Roos M (1975) Minuit: a system for function minimization and analysis of the parameter errors and correlations. Comput Phys Commun 10:343–367. https://doi.org/10.1016/0010-4655(75)90039-9
62. James F, Roos M Minuit2 user guide. https://root.cern/doc/master/md_math_minuit2_doc_Minuit2.html. Accessed 1 Aug 2022
63. The NLopt module for julia, code repository (2021) https://github.com/JuliaOpt/NLopt.jl. Accessed 29 Sep 2021
64. Optim.jl package code repository (2021) https://github.com/JuliaNLSolvers/Optim.jl. Accessed 29 Sep 2021
65. Optimization.jl package code repository (2021) https://github.com/SciML/Optimization.jl. Accessed 29 Sep 2021
66. Broyden CG (1970) The convergence of a class of double-rank minimization algorithms 1. General considerations. IMA J Appl Maths 6:76–90. https://doi.org/10.1093/imamat/6.1.76
67. Fletcher R (1970) A new approach to variable metric algorithms. Comput J 13:317–322. https://doi.org/10.1093/comjnl/13.3.317
68. Goldfarb D (1970) A family of variable-metric methods derived by variational means. Maths Comput. https://doi.org/10.2307/2004873
69. Shanno DF (1970) Conditioning of quasi-newton methods for function minimization. Maths Comput. https://doi.org/10.2307/2004840
70. Eschle J, Puig Navarro A, Silva Coutinho R, Serra N (2019) ZFIT: scalable pythonic fitting. SoftwareX. https://doi.org/10.1016/j.softx.2020.100508
71. Eschle J, Puig AN, Silva Coutinho R, Serra N (2020) ZFIT: scalable pythonic fitting. EPJ Web Conf 245:06025. https://doi.org/10.1051/epjconf/202024506025
72. Besançon M et al (2011) Distributions.jl: Definition and modeling of probability distributions in the Juliastats ecosystem. J Stat Softw 98:1–30
73. Lin D et al (2019) JuliaStats/Distributions.jl: a Julia package for probability distributions and associated functions. Zenodo. https://doi.org/10.5281/zenodo.2647458
74. StatsBase.jl package code repository (2021) https://github.com/JuliaStats/StatsBase.jl. Accessed 29 Sep 2021
75. Ling J, Amin N, Jacobsen R, Gal, T (2022) A pure julia 1D/2D histogram package that focus on speed and thread-safe. https://doi.org/10.5281/zenodo.7191111
76. Breloff T Plots–powerful convenience for vizualisation in Julia. https://docs.juliaplots.org/v1.30/. Accessed 15 Jun 2022

77. RecipeBase.jl (2022) https://github.com/JuliaPlots/RecipesBase.jl. Accessed 15 Jun 2022
78. Heinen J et al (1985–2022) GR framework. https://gr-framework.org/. Accessed 15 Jun 2022
79. Rosario HD, Heinen J (2019–2022) GRUtils. https://heliosdrm.github.io/GRUtils.j. Accessed 15 Jun 2022
80. Danisch S, Krumbiegel J (2021) Makie.jl: flexible high-performance data visualization for Julia. J Open Source Softw 6:3349. https://doi.org/10.21105/joss.03349
81. Bierlich C et al (2020) Robust independent validation of experiment and theory: rivet version 3. SciPost Phys 8:026. https://doi.org/10.21468/SciPostPhys.8.2.026
82. PFGPlots code repository (2022) https://github.com/JuliaTeX/PGFPlots.jl. Accessed 15 Jun 2022
83. PFGPlotsX code repository (2022) https://github.com/KristofferC/PGFPlotsX.jl. Accessed 15 Jun 2022
84. Gaston code repository (2022) https://github.com/mbaz/Gaston.jl. Accessed 15 Jun 2022
85. Gnuplot (2022) http://www.gnuplot.info/. Accessed 15 Jun 2022
86. Matplotlib (2022). https://matplotlib.org/. Accessed 15 Jun 2022
87. Vega-Lite (2022) https://vega.github.io/vega-lite/. Accessed 15 Jun 2022
88. VegaLite.jl code repository (2022) https://github.com/queryverse/VegaLite.jl. Accessed 15 Jun 2022
89. UnicodePlots code repository (2022) https://github.com/JuliaPlots/UnicodePlots.jl. Accessed 15 Jun 2022
90. Inc. W R (2022) Mathematica, Version 13.1. https://www.wolfram.com/mathematica. Champaign
91. van der Plas F et al (2022) fonsp/pluto.jl: v0.19.11. Zenodo. https://doi.org/10.5281/zenodo.6916713
92. Agostinelli S et al (2003) Geant4-a simulation toolkit. NIM-A 506:250–303
93. Byrne S, Wilcox LC, Churavy V (2021) Mpi.jl: Julia bindings for the message passing interface. Proc JuliaCon Conf 1:68. https://doi.org/10.21105/jcon.00068
94. Poulson J et al Elemental.jl code repository (2017-2022). https://github.com/JuliaParallel/Elemental.jl. Accessed 17 Mar 2022
95. Elrod C et al Loopvectorization code repository. https://github.com/JuliaSIMD/LoopVectorization.jl. Accessed 17 Mar 2022
96. Abbott M et al (2022) mcabbott/tullio.jl: v0.3.5. Zenodo. https://doi.org/10.5281/zenodo.7106192
97. Gowda S et al (2022) High-performance symbolic-numerics via multiple dispatch. ACM Commun Comput Algebra 55:92–96. https://doi.org/10.1145/3511528.3511535
98. Symata.jl (2022) https://github.com/jlapeyre/Symata.jl. Accessed 17 Mar 2022
99. MathLink.jl (2022) https://github.com/JuliaInterop/MathLink.jl. Accessed 15 Mar 2022
100. Gowda S et al (2021) High-performance symbolic-numerics via multiple dispatch. ACM Commun Comput Algebra 55:92–96. https://doi.org/10.1145/3511528.3511535
101. (2022) https://sciml.ai/. Accessed 1 Aug 2022
102. Rackauckas C, Nie Q (2017) Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. J Open Res Softw 5:15. https://doi.org/10.5334/jors.151
103. JuliaSymbolics roadmap: a modern computer algebra system for a modern language (2022) https://juliasymbolics.org/roadmap/. Accessed 16 Mar 2022
104. Ma Y et al (2021) Modelingtoolkit: a composable graph transformation system for equation-based modeling. http://arxiv.org/abs/2103.05244
105. SymbolicUtils.jl (2022) https://github.com/JuliaSymbolics/SymbolicUtils.jl. Accessed 15 Mar 2022
106. MathLinkExtras.jl (2022) https://github.com/fremling/MathLinkExtras.jl. Accessed 17 Mar 2022
107. Amoroso S et al (2021) Challenges in monte Carlo event generator software for high-luminosity LHC. Comput Softw Big Sci. 5:12. https://doi.org/10.1007/s41781-021-00055-1
108. Valassi A, Roiser S, Mattelaer O, Hageboeck S (2021) Design and engineering of a simplified workflow execution for the MG5aMC event generator on GPUs and vector CPUs. EPJ Web Conf. 251:03045. https://doi.org/10.1051/epjconf/202125103045
109. DaggerGPU package code repository (2021) https://github.com/JuliaGPU/DaggerGPU.jl. Accessed 29 Sep 2021
110. Preston-Werner T (2013) Semantic versioning 2.0.0. http://semver.org
111. Expression problem (2022) https://en.wikipedia.org/wiki/Expression_problem. Champaign
112. Reynolds JC (1978) User-Defined types and procedural data structures as complementary approaches to data abstraction, New York: Springer, pp 309–317 . https://doi.org/10.1007/978-1-4612-6315-9_22
113. Revise package code repository (2021) https://github.com/timholy/Revise.jl. Accessed 29 Sep 2021
114. JAX library code repository (2021) https://github.com/google/jax. Accessed 29 Sep 2021
115. TensorFlow web site (2021) https://www.tensorflow.org. Accessed 29 Sep 2021
116. PyTorch web site (2021) https://pytorch.org/. Accessed 29 Sep 2021
117. White FC et al (2022) Juliadiff/chainrules.jl: v1.44.7. Zenodo. https://doi.org/10.5281/zenodo.7182461
118. Juliadiff organisation website. https://juliadiff.org/. Accessed 7 Apr 2022
119. Lorentzvectorhep. https://github.com/JuliaHEP/LorentzVector-HEP.jl. Accessed 17 Mar 2022
120. Table.jl. https://tables.juliadata.org/stable/. Accessed 17 Mar 2022
121. Schulz O et al (2021) Bat.jl: a Julia-based tool for Bayesian inference. SN Comp Sci 2:210. https://doi.org/10.1007/s42979-021-00626-4
122. Ge H, Xu K, Ghahramani Z (2018) Turing: a language for flexible probabilistic inference, 1682–1690 MLR Press. In: proceedings of the Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain. http://proceedings.mlr.press/v84/ge18b.html
123. Gabler P, Schulz O, Widmann D et al Densityinterface.jl code repository (2021-2022). https://github.com/JuliaMath/DensityInterface.jl. Accessed 23 Nov 2022
124. Plavin A, Scherrer C, Schulz O, Widmann D et al Inversefunctions.jl code repository (2021-2022). https://github.com/JuliaMath/InverseFunctions.jl. Accessed 23 Nov 2022
125. Schulz O, Widmann D et al Changesofvariables.jl code repository (2021-2022). https://github.com/JuliaMath/ChangesOfVariables.jl. Accessed 23 Nov 2022
126. Collaboration TA, Collaboration TC, Group TLHC (2011) Procedure for the LHC higgs boson search combination in summer 2011. Technical Report CMS-NOTE-2011-005, ATL-PHYS-PUB-2011-011, ATL-PHYS-PUB-2011-11, CERN, Geneva. https://cds.cern.ch/record/1379837
127. Chatrchyan S et al (2012) Combined results of searches for the standard model Higgs boson in pp collisions at $\sqrt{s} = 7$TeV. Phys Lett B 710:26–48. https://doi.org/10.1016/j.physletb.2012.02.064
128. Collaboration TA (2012) Observation of an excess of events in the search for the standard model Higgs boson with the ATLAS detector at the LHC. https://cds.cern.ch/record/1460439. ATLAS-CONF-2012-093

129. Cowan G, Cranmer K, Gross E, Vitells O (2011) Asymptotic formulae for likelihood-based tests of new physics. Eur Phys J. https://doi.org/10.1140/epjc/s10052-011-1554-0. [Erratum: Eur. Phys. J. C 73, 2501 (2013)]
130. Junk T (1999) Confidence level computation for combining searches with small statistics. Nucl Instrum Meth A 434:435–443. https://doi.org/10.1016/S0168-9002(99)00498-2
131. Read AL (2002) Presentation of search results: the CL(s) technique. J Phys G 28:2693–2704. https://doi.org/10.1088/0954-3899/28/10/313
132. Fisher W (2006) Systematics and limit calculations. FERMILAB-TM-2386-E, D0-NOTE-5309. https://doi.org/10.2172/923070
133. Junk T (2006) Sensitivity, exclusion and discovery with small signals, large backgrounds, and large systematic uncertainties. https://inspirehep.net/literature/1361506. CDF-8128, CDF-Note-8128
134. Verkerke W, Kirkby D (2006) The RooFit toolkit for data modeling. World Sci. https://doi.org/10.1142/9781860948985_0039
135. Boutigny D et al (1995) BaBar technical design report. SLAC-R-457 http://www.slac.stanford.edu/cgi-wrap/getdoc/slac-r-457.pdf
136. Cranmer K, Lewis G, Moneta L, Shibata A, Verkerke W (2012) HistFactory: A tool for creating statistical models for use with RooFit and RooStats. CERN-OPEN-2012-016. https://cds.cern.ch/record/1456844
137. ATLAS, CMS, LHC Higgs Combination Group (2011) Procedure for the LHC Higgs boson search combination in Summer 2011. CMS-NOTE-2011-005, ATL-PHYS-PUB-2011-11. https://cds.cern.ch/record/1379837
138. HiggsCombine code repository. https://github.com/cms-analysis/HiggsAnalysis-CombinedLimit. Accessed 1 Aug 2022
139. Heinrich L, Feickert M, Stark G, Cranmer K (2021) PYHF: pure-Python implementation of HistFactory statistical models. J Open Source Softw 6:2823. https://doi.org/10.21105/joss.02823
140. Ling J (2022) LiteHF.jl: Julia implementation of HistFactory-style likelihood ratio methods and test statistics. Zenodo. https://doi.org/10.5281/zenodo.7435541
141. Blobel V (2013) Unfolding, Ch. 6, 187–225 John Wiley and Sons, Ltd, https://doi.org/10.1002/9783527653416.ch6
142. Schmitt S (2012) TUnfold: an algorithm for correcting migration effects in high energy physics. JINST 7:T10003. https://doi.org/10.1088/1748-0221/7/10/T10003
143. Adye T (2011) in Proceeedings of the PHYSTAT 2011 workshop on statistical issues related to discovery claims in search experiments and unfolding Prosper H, Lyons L. (eds) Unfolding algorithms and tests using RooUnfold 313–318 (CERN, Geneva, 2011). https://doi.org/10.5170/CERN-2011-006.313. arXiv:1105.1160
144. Brenner L et al (2020) Comparison of unfolding methods using RooFitUnfold. Int J Mod Phys A 35:2050145. https://doi.org/10.1142/S0217751X20501456
145. Andreassen A, Komiske PT, Metodiev EM, Nachman B, Thaler J (2020) OmniFold: a method to simultaneously unfold all observables. Phys Rev Lett 124:182001. https://doi.org/10.1103/PhysRevLett.124.182001
146. Pharmaceutical development, pfizer uses julia to accelerate simulations of new therapies for metabolic diseases up to 175x. https://juliacomputing.com/case-studies/pfizer/. Accessed 1 Aug 2022
147. (Moderna), H A Modeling and simulation to guide dose selection for mRNA therapeutics and vaccines. Presented at the JuliaCon 2022 conference. https://live.juliacon.org/talk/9N9HZ3.
148. Predicting toxicity. https://juliacomputing.com/case-studies/astra-zeneca/. Accessed 1 Aug 2022
149. https://juliacomputing.com/industries/banking-and-finance/. Accessed 1 Aug 2022
150. Solvency II compliance, one of Europe's largest insurers is using Julia for solvency II compliance. https://juliacomputing.com/case-studies/aviva/. Accessed 1 Aug 2022
151. Macroeconomic modeling, the Federal reserve bank of New York publishes its trademark dynamic stochastic general equilibrium models in Julia. https://juliacomputing.com/case-studies/ny-fed/. Accessed 1 Aug 2022
152. Holy T, Bezanson J, Nash J Analyzing sources of compiler latency in Julia: method invalidations. https://julialang.org/blog/2020/08/invalidations/. Accessed 7 Apr 2022
153. Holy T Tutorial on precompilation. https://julialang.org/blog/2021/01/precompile_tutorial/. Accessed 14 Oct 2022
154. SnoopCompile package code repository (2021) https://github.com/timholy/SnoopCompile.jl. Accessed 29 Sep 2021
155. PrecompileTools package code repository (2023) https://github.com/JuliaLang/PrecompileTools.jl. Accessed 5 Jun 2023
156. Packagecompiler manual: Sysimages. https://julialang.github.io/PackageCompiler.jl/v2.0/sysimages.html. Accessed 15 Jun 2022
157. Collaboration C. CMS offline software repository. https://github.com/cms-sw/cmssw. Accessed 1 Aug 2022
158. Tokei computer program code repository (2021) https://github.com/XAMPPRocky/tokei. Accessed 29 Sep 2021
159. Differentialequations.jl: Scientific machine learning (SciML) enabled simulation and estimation. https://diffeq.sciml.ai/v7.3.0/. Accessed 15 Jun 2022
160. Holy T, Churavy V (2022) Improvements in package precompilation. Talk given at JuliaCon 2022. https://live.juliacon.org/talk/DUQQLN.
161. Documenter.jl, a documentation generator for Julia. https://juliadocs.github.io/Documenter.jl/v0.27/. Accessed 15 Jun 2022
162. Abgrall N et al (2021) The Large enriched germanium experiment for neutrinoless $\beta\beta$ decay: LEGEND-1000 preconceptual design report. http://arxiv.org/abs/2107.11462
163. Evaluation strategy. https://en.wikipedia.org/wiki/Evaluation_strateg. Accessed 1 Aug 2022