Awkward Packaging: building Scikit-HEP

Henry Schreiner[‡*], Jim Pivarski[‡], Eduardo Rodrigues[§]

Abstract—Scikit-HEP has grown rapidly over the last few years, not just to serve the needs of the High Energy Physics (HEP) community, but in many ways, the Python ecosystem at large. AwkwardArray, boost-histogram/hist, and iminuit are examples of libraries that are used beyond the original HEP focus. In this paper we will look at key packages in the ecosystem, and how the collection of 30+ packages was developed and maintained. Also we will look at some of the software ecosystem contributions made to packages like cibuildwheel, pybind11, nox, scikit-build, build, and pipx that support this effort. We will also discuss the Scikit-HEP developer pages and initial WebAssembly support.

Index Terms—packaging, ecosystem, high energy physics, community project

Introduction

High Energy Physics (HEP) has always had intense computing needs due to the size and scale of the data collected. The World Wide Web was invented at the CERN Physics laboratory in Switzerland in 1989 when scientists in the EU were trying to communicate results and datasets with scientist in the US, and vice-versa [LCC+09]. Today, HEP has the largest scientific machine in the world, at CERN: the Large Hadron Collider (LHC), 27 km in circumference [EB08], with multiple experiments with thousands of collaborators processing over a petabyte of raw data every day, with 100 petabytes being stored per year at CERN. This is one of the largest scientific datasets in the world of exabyte scale [PJ11], which is roughly comparable in order of magnitude to all of astronomy or YouTube [SLF+15].

In the mid nineties, HEP users were beginning to look for a new language to replace Fortran. A few HEP scientists started investigating the use of Python around the release of 1.0.0 in 1994 [Tem22]. A year later, the ROOT project for an analysis toolkit (and framework) was released, quickly making C++ the main language for HEP. The ROOT project also needed an interpreted language to driving analysis code. Python was rejected for this role due to being "exotic" at the time, and because it was considered too much to ask physicists to code in two languages. Instead, ROOT provided a C++ interpreter, called CINT, which later was replaced with Cling, which is the basis for the clang-repl project in LLVM today [IVL22].

Python would start showing up in the late 90's in experiment frameworks as a configuration language. These frameworks were primarily written in C++, but were made of many configurable parts [Lam98]. The glueing together of the system was done in Python, a model still popular today, though some experiments are now using Python + Numba as an alternative model, such as for example the Xenon1T experiment [RTA+17], [RS21].

In the early 2000s, the use of Python HEP exploded, heavily driven by experiments like LHCb developing frameworks and user tools for scripting. ROOT started providing Python bindings in 2004 [LGMM05] that were not considered Pythonic [GTW20], and still required a complex multi-hour build of ROOT to use[1]. Analyses still consisted largely of ROOT, with Python sometimes showing up.

By the mid 2010's, a marked change had occurred, driven by the success of Python in Data Science, especially in education. Many new students were coming into HEP with little or no C++ experience, but with existing knowledge of Python and the growing Python data science ecosystem, like NumPy and Pandas. Several HEP experiment analyses were performed in, or driven by, Python, with ROOT only being used for things that were not available in the Python ecosystem. Some of these were HEP specific: ROOT is also a data format, so users needed to be able to read data from ROOT files. Others were less specific: HEP users have intense histogram requirements due to the data sizes, large portions of HEP data are "jagged" rather than rectangular; vector manipulation was important (especially Lorenz Vectors, a four dimensional relativistic vector with a non-Euclidean metric); and data fitting was important, especially with complex models and accurate error estimation.

Beginnings of a scikit

In 2016, the ecosystem for Python in HEP was rather fragmented. Physicists were developing tools in isolation, without knowing out the overlaps with other tools, and without making them interoperable. There were a handful of popular packages that were useful in HEP spread around among different authors. The ROOTPy project had several packages that made the ROOT-Python bridge a little easier than the built-in PyROOT, such as the root-numpy and related root-pandas packages. The C++ MINUIT fitting library was integrated into ROOT, but the iminuit package [Dea20] provided an easy to install standalone Python package with an extracted copy of MINUIT. Several other specialized standalone C++ packages had bindings as well. Many of the initial authors were transitioning to a less-code centric role or leaving for industry, leaving projects like ROOTPy and iminuit without maintainers.

1. Almost 20 years later ROOT's Python bindings have been rewritten for easier Pythonizations, and installing ROOT in Conda is now much easier, thanks in large part to efforts from Scikit-HEP developers.

* Corresponding author: henryfs@princeton.edu
‡ Princeton University
§ University of Liverpool

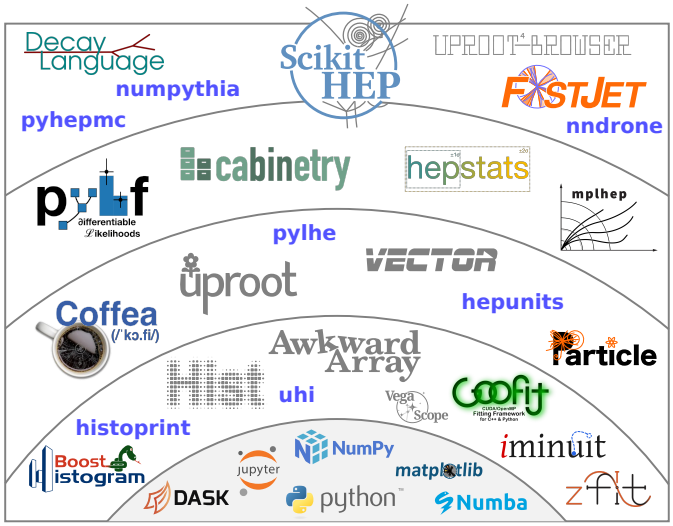

Fig. 1: *The Scikit-HEP ecosystem and affiliated packages.*

Eduardo Rodrigues, a scientist working on the LHCb experiment for the University of Cincinnati, started working on a package called scikit-hep that would provide a set to tools useful for physicists working on HEP analysis. The initial version of the scikit-hep package had a simple vector library, HEP related units and conversions, several useful statistical tools, and provenance recording functionality,

He also placed the scikit-hep GitHub repository in a Scikit-HEP GitHub organization, and asked several of the other HEP related packages to join. The ROOTPy project was ending, with the primary author moving on, and so several of the then-popular packages[2] that were included in the ROOTPy organization were happily transferred to Scikit-HEP. Several other existing HEP libraries, primarily interfacing to existing C++ simulation and tracking frameworks, also joined, like PyJet and NumPythia. Some of these libraries have been retired or replaced today, but were an important part of Scikit-HEP's initial growth.

First initial success

In 2016, the largest barrier to using Python in HEP in a Pythonic way was ROOT. It was challenging to compile, had many non-Python dependencies, was huge compared to most Python libraries, and didn't play well with Python packaging. It was not Pythonic, meaning it had very little support for Python protocols like iteration, buffers, keyword arguments, tab completion and inspect in, dunder methods, didn't follow conventions for useful reprs, and Python naming conventions; it was simply a direct on-demand C++ binding, including pointers. Many Python analyses started with a "convert data" step using PyROOT to read ROOT files and convert them to a Python friendly format like HDF5. Then the bulk of the analysis would use reproducible Python virtual environments or Conda environments.

This changed when Jim Pivarski introduced the Uproot package, a pure-Python implementation of a ROOT file reader (and later writer) that could remove the initial conversion environment by simply pip installing a package. It also had a simple, Pythonic interface and produced outputs Python users could immediately use, like NumPy arrays, instead of PyROOT's wrapped C++ pointers.

Uproot needed to do more than just be file format reader/writer; it needed to provide a way to represent the special structure and common objects that ROOT files could contain. This lead to the development of two related packages that would support uproot. One, uproot-methods, included Pythonic access to functionality provided by ROOT for its core classes, like spatial and Lorentz vectors. The other was AwkwardArray, which would grow to become one of the most important and most general packages in Scikit-HEP. This package allows NumPy-like idioms for array-at-a-time manipulation on jagged data structures. A jagged array is a (possibly structured) array with a variable length dimension. These are very common and relevant in HEP; events have a variable number of tracks, tracks have a variable number of hits in the detector, etc. Many other fields also have jagged data structures. While there are formats to store such structures, computations on jagged structures have usually been closer to SQL queries on multiple tables than direct object manipulation. Pandas handles this through multiple indexing and a lot of duplication.

Uproot was a huge hit with incoming HEP students (see Fig 2); suddenly they could access HEP data using a library installed with pip or conda and no external compiler or library requirements, and could easily use tools they already knew that were compatible with the Python buffer protocol, like NumPy, Pandas and the rapidly growing machine learning frameworks. There were still some gaps and pain points in the ecosystem, but an analysis without writing C++ (interpreted or compiled) and compiling ROOT manually was finally possible. Scikit-HEP did not and does not intend to replace ROOT, but it provides alternative solutions that work natively in the Python "Big Data" ecosystem.

Several other useful HEP libraries were also written. Particle was written for accessing the Particle Data Group (PDG) particle data in a simple and Pythonic way. DecayLanguage originally provided tooling for decay definitions, but was quickly expanded to include tools to read and validate "DEC" decay files, an existing text format used to configure simulations in HEP.

Building compiled packages

In 2018, HEP physicist and programmer Hans Dembinski proposed a histogram library to the Boost libraries, the most influential C++ library collection; many additions to the standard library are based on Boost. Boost.Histogram provided a histogram-as-an-object concept from HEP, but was designed around C++14 templating, using composable axes and storage types. It originally had an initial Python binding, written in Boost::Python. Henry Schreiner proposed the creation of a standalone binding to be written with pybind11 in Scikit-HEP. The original bindings were removed, Boost::Histogram was accepted into the Boost libraries, and work began on boost-histogram. IRIS-HEP, a multi-institution project for sustainable HEP software, had just started, which was providing funding for several developers to work on Scikit-HEP project packages such as this one. This project would pioneer standalone C++ library development and deployment for Scikit-HEP.

There were already a variety of attempts at histogram libraries, but none of them filled the requirements of HEP physicists:

2. The primary package of the ROOTPy project, also called ROOTPy, was not transferred, but instead had a final release and then died. It was an inspiration for the new PyROOT bindings, and influenced later Scikit-HEP packages like mplhep. The transferred libraries have since been replaced by integrated ROOT functionality. All these packages required ROOT, which is not on PyPI, so were not suited for a Python-centric ecosystem.

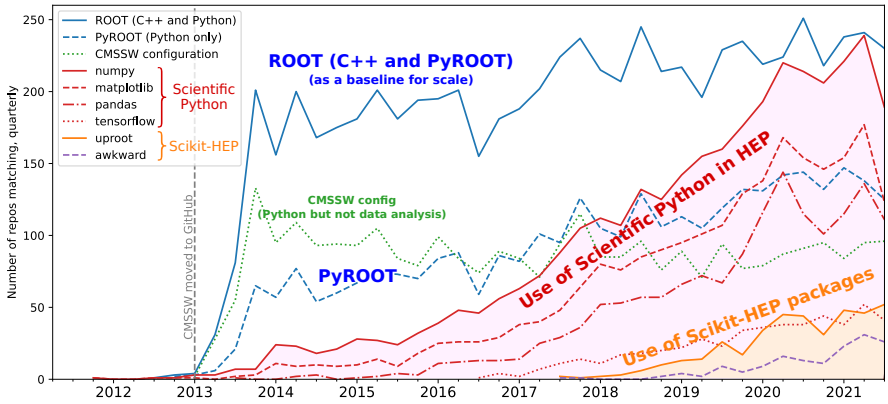

Fig. 2: *Adoption of scientific Python libraries and Scikit-HEP among members of the CMS experiment (one of the four major LHC experiments). CMS requires users to fork* `github:cms-sw/cmssw`*, which can be used to identify 3484 physicist users, who created 16656 non-fork repos. This plot quantifies adoption by counting* `"#include X"`*,* `"import X"`*, and* `"from X import"` *strings in the users' code to measure adoption of various libraries (most popular by category are shown).*

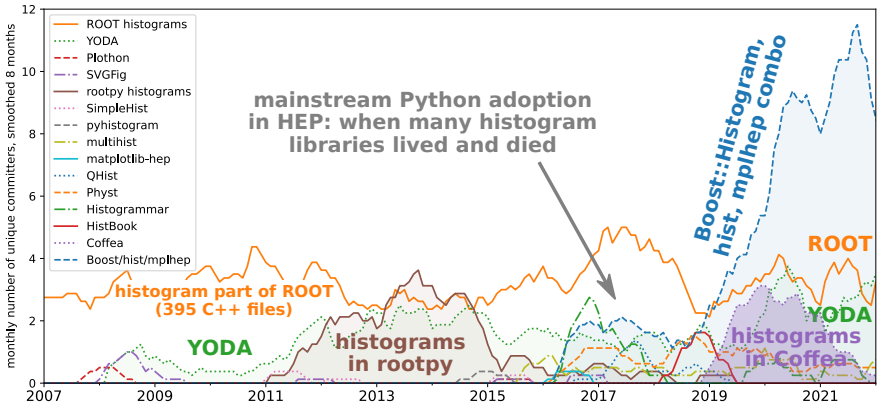

Fig. 3: *Developer activity on histogram libraries in HEP: number of unique committers to each library per month, smoothed (derived from git logs). Illustrates the convergence of a fractured community (around 2017) into a unified one (now).*

fills on pre-existing histograms, simple manipulation of multi-dimensional histograms, competitive performance, and easy to install in clusters or for students. Any new attempt here would have to be clearly better than the existing collection of diverse attempts (see Fig 3). The development of a library with compiled components intended to be usable everywhere required good support for building libraries that was lacking both in Scikit-HEP and to an extent the broader Python ecosystem. Previous advancements in the packaging ecosystem, such as the wheel format for distributing binary platform dependent Python packages and the manylinux specification and docker image that allowed a single compiled wheel to target many distributions of Linux, but there still were many challenges to making a library redistributable on all platforms.

The boost-histogram library only depended on header-only components of the Boost libraries, and the header-only pybind11 package, so it was able to avoid a separate compile step or linking to external dependencies, which simplified the initial build process. All needed files were collected from git submodules and packed into a source distribution (SDist), and everything was built using only setuptools, making build-from-source simple on any system supporting C++14. This did not include RHEL 7, a popular platform in HEP at the time, and on any platform building could take several minutes and required several gigabytes of memory to resolve the heavy C++ templating in the Boost libraries and pybind11.

The first stand-alone development was azure-wheel-helpers, a set of files that helped produce wheels on the new Azure Pipelines platform. Building redistributable wheels requires a variety of techniques, even without shared libraries, that vary dramatically between platforms and were/are poorly documented. On Linux, everything needs to be built inside a controlled manylinux image, and post-processed by the auditwheel tool. On macOS, this includes downloading an official CPython binary for Python to allow older versions of macOS to be targeted (10.9+), several special environment variables, especially when cross compiling to Apple Silicon, and post processing with the delocate tool. Windows is the simplest, as most versions of CPython work identically there. azure-wheel-helpers worked well, and was quickly adapted for the other packages in Scikit-HEP that included non-ROOT binary components. Work here would eventually be merged into the existing and general cibuildwheel package, which would become the build tool for all non-ROOT binary packages in Scikit-HEP, as well as over 600 other packages like matplotlib and numpy, and was accepted into the PyPA (Python Packaging Authority).

The second major development was the upstreaming of CI and build system developments to pybind11. Pybind11 is a C++ API for Python designed for writing a binding to C++, and provided significant benefits to our packages over (mis)-using Cython for bindings; Cython was designed to transpile a Python-

like language to C (or C++), and just happened to support bindings since you can call C and C++ from it, but it was not what it was designed for. Benefits of pybind11 included reduced code complexity and duplication, no pre-process step (cythonize), no need to pin NumPy when building, and a cross-package API. The iMinuit package was later moved from Cython to pybind11 as well, and pybind11 became the Scikit-HEP recommended binding tool. We contributed a variety of fixes and features to pybind11, including positional-only and keyword-only arguments, the option to prepend to the overload chain, and an API for type access and manipulation. We also completely redesigned CMake integration, added a new pure-Setuptools helpers file, and completely redesigned the CI using GitHub Actions, running over 70 jobs on a variety of systems and compilers. We also helped modernize and improve all the example projects with simpler builds, new CI, and cibuildwheel support.

This example of a project with binary components being usable everywhere then encouraged the development of Awkward 1.0, a rewrite of AwkwardArray replacing the Python-only code with compiled code using pybind11, fixing some long-standing limitations, like an inability to slice past two dimensions or select "n choose k" for $k > 5$; these simply could not be expressed using Awkward 0's NumPy expressions, but can be solved with custom compiled kernels. This also enabled further developments in backends [PEL20].

Broader ecosystem

Scikit-HEP had become a "toolset" for HEP analysis in Python, a collection of packages that worked together, instead of a "toolkit" like ROOT, which is one monopackage that tries to provide everything [R+20]. A toolset is more natural in the Python ecosystem, where we have good packaging tools and many existing libraries. Scikit-HEP only needed to fill existing gaps, instead of covering every possible aspect of an analysis like ROOT did. The original scikit-hep package had its functionality pulled out into existing or new separate packages such as HEPUnits and Vector, and the core scikit-hep package instead became a metapackage with no unique functionality on its own. Instead, it installs a useful subset of our libraries for a physicist wanting to quickly get started on a new analysis.

Scikit-HEP was quickly becoming the center of HEP specific Python software (see Fig. 1). Several other projects or packages joined Scikit-HEP iMinuit, a popular HEP and astrophysics fitting library, was probably the most widely used single package to have joined. PyHF and cabinetry also joined; these were larger frameworks that could drive a significant part of an analysis internally using other Scikit-HEP tools.

Other packages, like GooFit, Coffea, and zFit, were not added, but were built on Scikit-HEP packages and had developers working closely with Scikit-HEP maintainers. Scikit-HEP introduced an "affiliated" classification for these packages, which allowed an external package to be listed on the Scikit-HEP website and encouraged collaboration. Coffea had a strong influence on histogram design, and zFit has contributed code to Scikit-HEP. Currently all affiliated packages have at least one Scikit-HEP developer as a maintainer, though that is currently not a requirement. An affiliated package fills a particular need for the community. Scikit-HEP doesn't have to, or need to, attempt to develop a package that others are providing, but rather tries to ensure that the externally provided package works well with the

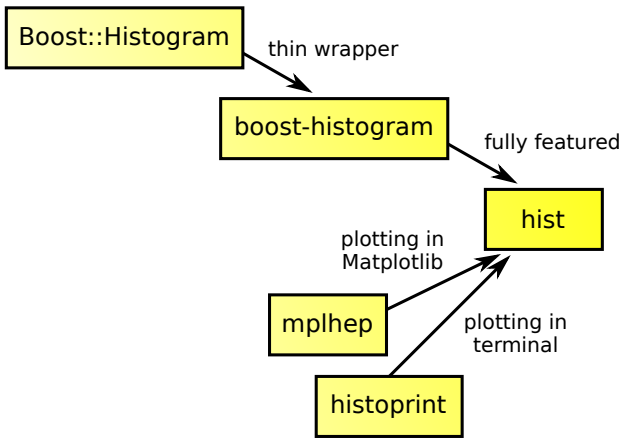

Fig. 4: *The collection of histogram packages and related packages in Scikit-HEP.*

broader HEP ecosystem. The affiliated classification is also used on broader ecosystem packages like pybind11 and cibuildwheel that we recommend and share maintainers with.

Histogramming was designed to be a collection of specialized packages (see Fig. 4) with carefully defined interoperability; boost-histogram for manipulation and filling, Hist for a user-friendly interface and simple plotting tools, histoprint for displaying histograms, and the existing mplhep and uproot packages also needed to be able to work with histograms. This ecosystem was built and is held together with UHI, which is a formal specification agreed upon by several developers of different libraries, backed by a statically typed Protocol, for a PlottableHistogram object. Producers of histograms, like boost-histogram/hist and uproot provide objects that follow this specification, and users of histograms, such as mplhep and histoprint take any object that follows this specification. The UHI library is not required at runtime, though it does also provide a few simple utilities to help a library also accept ROOT histograms, which do not (currently) follow the Protocol, so several libraries have decided to include it at runtime too. By using a static type checker like MyPy to statically enforce a Protocol, libraries that can communicate without depending on each other or on a shared runtime dependency and class inheritance. This has been a great success story for Scikit-HEP, and We expect Protocols to continue to be used in more places in the ecosystem.

The design for Scikit-HEP as a toolset is of many parts that all work well together. One example of a package pulling together many components is uproot-browser, a tool that combines uproot, Hist, and Python libraries like textual and plotext to provide a terminal browser for ROOT files.

Scikit-HEP's external contributions continued to grow. One of the most notable ones was our work on cibuildwheel. This was a Python package that supported building redistributable wheels on multiple CI systems. Unlike our own azure-wheel-helpers or the competing multibuild package, it was written in Python, so good practices in Python package design could apply, like unit and integration tests, static checks, and it was easy to remain independent of the underlying CI system. Building wheels on Linux requires a docker image, macOS requires the python.org Python, and Windows can use any copy of Python - cibuildwheel uses this to supply Python in all cases, which keeps it from

depending on the CI's support for a particular Python version. We merged our improvements to cibuildwheel, like better Windows support, VCS versioning support, and better PEP 518 support. We dropped azure-wheel-helpers, and eventually a scikit-build maintainer joined the cibuildwheel project. cibuildwheel would go on to join the PyPA, and is now in use in over 600 packages, including numpy, matplotlib, mypy, and scikit-learn.

Our continued contributions to cibuildwheel included a TOML-based configuration system for cibuildwheel 2.0, an override system to make supporting multiple manylinux and musllinux targets easier, a way to build directly from SDists, an option to use build instead of pip, the automatic detection of Python version requirements, and better globbing support for build specifiers. We also helped improve the code quality in various ways, including fully statically typing the codebase, applying various checks and style controls, automating CI processes, and improving support for special platforms like CPython 3.8 on macOS Apple Silicon.

We also have helped with build, nox, pyodide, and many other packages, improving the tooling we depend on to develop scikit-build and giving back to the community.

The Scikit-HEP Developer Pages

A variety of packaging best practices were coming out of the boost-histogram work, supporting both ease of installation for users as well as various static checks and styling to keep the package easy to maintain and reduce bugs. These techniques would also be useful apply to Scikit-HEP's nearly thirty other packages, but applying them one-by-one was not scalable. The development and adoption of azure-wheel-helpers included a series of blog posts that covered the Azure Pipelines platform and wheel building details. This ended up serving as the inspiration for a new set of pages on the Scikit-HEP website for developers interested in making Python packages. Unlike blog posts, these would be continuously maintained and extended over the years, serving as a template and guide for updating and adding packages to Scikit-HEP, and educating new developers.

These pages grew to describe the best practices for developing and maintaining a package, covering recommended configuration, style checking, testing, continuous integration setup, task runners, and more. Shortly after the introduction of the developer pages, Scikit-HEP developers started asking for a template to quickly produce new packages following the guidelines. This was eventually produced; the "cookiecutter" based template is kept in sync with the developer pages; any new addition to one is also added to the other. The developer pages are also kept up to date using a CI job that bumps any GitHub Actions or pre-commit versions to the most recent versions weekly. Some portions of the developer pages have been contributed to packaging.python.org, as well.

The cookie cutter was developed to be able to support multiple build backends; the original design was to target both pure Python and Pybind11 based binary builds. This has expanded to include 11 different backends by mid 2022, including Rust extensions, many PEP 621 based backends, and a Scikit-Build based backend for pybind11 in addition to the classic Setuptools one. This has helped work out bugs and influence the design of several PEP 621 packages, including helping with the addition of PEP 621 to Setuptools.

The most recent addition to the pages was based on a new repo-review package which evaluates and existing repository to see what parts of the guidelines are being followed. This was helpful for monitoring adoption of the developer pages, especially newer additions, across the Scikit-HEP packages. This package was then implemented directly into the Scikit-HEP pages, using Pyodide to run Python in WebAssembly directly inside a user's browser. Now anyone visiting the page can enter their repository and branch, and see the adoption report in a couple of seconds.

Working toward the future

Scikit-HEP is looking toward the future in several different areas. We have been working with the Pyodide developers to support WebAssembly; boost-histogram is compiled into Pyodide 0.20, and Pyodide's support for pybind11 packages is significantly better due to that work, including adding support for C++ exception handling. PyHF's documentation includes a live Pyodide kernel, and a try-pyhf site (based on the repo-review tool) lets users run a model without installing anything - it can even be saved as a webapp on mobile devices.

We have also been working with Scikit-Build to try to provide a modern build experience in Python using CMake. This project is just starting, but we expect over the next year or two that the usage of CMake as a first class build tool for binaries in Python will be possible using modern developments and avoiding distutils/setuptools hacks.

Summary

The Scikit-HEP project started in Autumn 2016 and has grown to be a core component in many HEP analyses. It has also provided packages that are growing in usage outside of HEP, like AwkwardArray, boost-histogram/Hist, and iMinuit. The tooling developed and improved by Scikit-HEP has helped Scikit-HEP developers as well as the broader Python community.

REFERENCES

[Dea20] Hans Dembinski and Piti Ongmongkolkul et al. scikit-hep/iminuit. Dec 2020. URL: https://doi.org/10.5281/zenodo.3949207, doi:10.5281/zenodo.3949207.

[EB08] Lyndon Evans and Philip Bryant. Lhc machine. *Journal of instrumentation*, 3(08):S08001, 2008.

[GTW20] Galli, Massimiliano, Tejedor, Enric, and Wunsch, Stefan. "a new pyroot: Modern, interoperable and more pythonic". *EPJ Web Conf.*, 245:06004, 2020. URL: https://doi.org/10.1051/epjconf/202024506004, doi:10.1051/epjconf/202024506004.

[IVL22] Ioana Ifrim, Vassil Vassilev, and David J Lange. GPU Accelerated Automatic Differentiation With Clad. *arXiv preprint arXiv:2203.06139*, 2022.

[Lam98] Stephan Lammel. Computing models of cdf and dØ in run ii. *Computer Physics Communications*, 110(1):32–37, 1998. URL: https://www.sciencedirect.com/science/article/pii/S0010465597001501, doi:10.1016/s0010-4655(97)00150-1.

[LCC+09] Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Larry G Roberts, and Stephen Wolff. A brief history of the internet. *ACM SIGCOMM Computer Communication Review*, 39(5):22–31, 2009.

[LGMM05] W Lavrijsen, J Generowicz, M Marino, and P Mato. Reflection-Based Python-C++ Bindings. 2005. URL: https://cds.cern.ch/record/865620, doi:10.5170/CERN-2005-002.441.

[PEL20] Jim Pivarski, Peter Elmer, and David Lange. Awkward arrays in python, c++, and numba. In *EPJ Web of Conferences*, volume 245, page 05023. EDP Sciences, 2020. doi:10.1051/epjconf/202024505023.

[PJ11] Andreas J Peters and Lukasz Janyst. Exabyte scale storage at CERN. In *Journal of Physics: Conference Series*, volume 331, page 052015. IOP Publishing, 2011. doi:10.1088/1742-6596/331/5/052015.

[R+20] Eduardo Rodrigues et al. The Scikit HEP Project – overview and prospects. *EPJ Web of Conferences*, 245:06028, 2020. arXiv:2007.03577, doi:10.1051/epjconf/202024506028.

[RS21] Olivier Rousselle and Tom Sykora. Fast simulation of Time-of-Flight detectors at the LHC. In *EPJ Web of Conferences*, volume 251, page 03027. EDP Sciences, 2021. doi:10.1051/epjconf/202125103027.

[RTA+17] D Remenska, C Tunnell, J Aalbers, S Verhoeven, J Maassen, and J Templon. Giving pandas ROOT to chew on: experiences with the XENON1T Dark Matter experiment. In *Journal of Physics: Conference Series*, volume 898, page 042003. IOP Publishing, 2017.

[SLF+15] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS biology*, 13(7):e1002195, 2015.

[Tem22] Jeffrey Templon. Reflections on the uptake of the Python programming language in Nuclear and High-Energy Physics, March 2022. None. URL: https://doi.org/10.5281/zenodo.6353621, doi:10.5281/zenodo.6353621.