EPJ Web of Conferences **245**, 05014 (2020) https://doi.org/10.1051/epjconf/202024505014
CHEP 2019

Recent developments in histogram libraries

Hans Peter Dembinski[1], *Jim* Pivarski[2], and *Henry* Schreiner[2,*]

[1]Max Planck Institute for Nuclear Physics, Heidelberg, Germany
[2]Princeton University, Princeton, USA

Abstract. Boost.Histogram, a header-only C++14 library that provides multi-dimensional histograms and profiles, became available in Boost 1.70. It is extensible, fast, and uses modern C++ features. Using template metaprogramming, the most efficient code path for any given configuration is automatically selected. The library includes key features designed for the particle physics community, such as optional under- and overflow bins, weighted increments, reductions, growing axes, thread-safe filling, and memory-efficient counters with high-dynamic range.

Python bindings for Boost.Histogram are being developed in the Scikit-HEP project to provide a fast, easy-to-install package as a backend for other Python libraries and for advanced users to manipulate histograms. Versatile and efficient histogram filling, effective manipulation, multithreading support, and other features make this a powerful tool. This library has also driven package distribution efforts in Scikit-HEP, allowing binary packages hosted on PyPI to be available for a very wide variety of platforms.

Two other libraries fill out the remainder of the Scikit-HEP Python histogramming effort. Aghast is a library designed to provide conversions between different forms of histograms, enabling interaction between histogram libraries, often without an extra copy in memory. This enables a user to make a histogram in one library and then save it in another form, such as saving a Boost.Histogram in ROOT. And Hist is a library providing friendly, analyst-targeted syntax and shortcuts for quick manipulations and fast plotting using these two libraries.

1 Introduction

There is no shortage of histogramming libraries for Python (see Table 1). However, many/most of these are abandoned, have a narrow focus, and most importantly, have little or no interaction with other histogramming libraries. For the Scikit-HEP family of Python libraries [1], histogramming was identified as a weak point in the scientific Python stack that could be addressed directly in Scikit-HEP. At this time, a new histogramming library for C++ was being developed and was about to be proposed for inclusion in the Boost Libraries. Working together with the author of what would become Boost.Histogram via a thorough review and unanimous approval, a plan was devised to build a Python system for histogramming on top of Boost.Histogram. Together, these two closely related projects have been providing some of the most exciting developments in histogramming for HEP in recent years.

*e-mail: hschrein@cern.ch

Table 1. Histogram libraries for Python. For the PyPI column, "Pure" means the library is pure Python, "Wheels" means it is compiled but binary wheels are available, "Source" means the code is there, but must be compiled and may require other dependencies, and "No" means it is not hosted on PyPI. Old projects that predate universal (pure python) wheels may be incorrectly listed as "Source". Ordered by last update time, compiled near the end of 2019.

Library	Updated	PyPI	Notes
NumPy	2019	Wheels	Very simple histogramming functions
coffea	2019	Pure	Family of tools for HEP Columnar analysis
Histogrammar	2019	Pure	Multilanguage, limited support
pygram11	2019	Wheels	Unix only, Python 3 only
PyROOT	2019	No	CERN's ROOT, UNIX binaries on conda-forge
YODA	2019	No	HEP tool for MCnet
physt	2019	Pure	Non-HEP specific tool
fast-histogram	2019	Yes	Fast but limited
Vaex	2019	Source	Large system for data analysis
hdrhistogram	2019	Source	Multilanguage, large range
multihist	2019	Pure	NumPy wrapper for syntax
HistBook	2018	Pure	Archived, Replaced by boost-histogram / hist
qhist	2018	Source	ROOT required, Python 2.7 only
theodoregoetz	2018	No	Tried to combine many of the below packages
rootplotlib	2016	No	ROOT backend
matplotlib-hep	2016	Source	Focused on plotting
SVGFig	2016	No	Plotting framework
Plothon	2015	No	Predecessor to SVGFig
pyhistogram	2014	Pure	Inspired by rootpy
pypeaks	2014	Pure	Peak detection
Cassius	2013	No	Statistical Modeling Package
histogramy	2013	Pure	1D with some fitting tools
histogram	2011	Source	For Distributed Data Analysis for Neutron Scattering
SimpleHist	2011	Pure	NumPy based
paida	2007	Source	Analysis and plotting

2 Boost.Histogram for C++

Boost.Histogram is a header-only C++14 library which implements feature-rich multi-dimensional generalized histograms. It is part of the Boost C++ Libraries and only depends on a few other header-only libraries from the Boost project and the C++ standard library. The feature set of Boost.Histogram was designed with the needs of the (astro)particle physics community and the wider data analysis community in mind. In particular, feature parity with the histogram implementations from the ROOT framework [2] and the GNU Scientific Library [3] was a central goal.

The library was designed to be easy to use for the casual user, while offering a high amount of flexibility and extensibility for the power user. In a classic C++ design based on run-time polymorphism, flexibility and extensibility comes at the cost of performance. Boost.Histogram avoids the run-time trade-off and achieves higher performance than other libraries with static polymorphism based on templates and modern template meta-programming.

Compatibility with other Boost libraries and the C++ standard library was another central design goal. By adhering to standard interfaces, the library avoids duplicating functionality provided by other libraries, especially the standard library. For example, these are one-liners:

- Sum all counts with `std::accumulate`.

- Find the cell with the highest count with `std::max_element`.
- Compute the cumulative distribution with `std::partial_sum`.

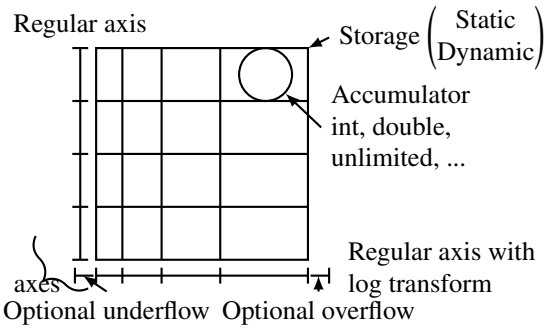

Figure 1. Components of a histogram. Zero or more axes (two shown) are combined with a storage. Each item stores an accumulator.

2.1 Generalized histograms

A histogram in the common sense (see Figure 1) is a collection of counts n_i, where each is associated to an interval out of a sequence of non-overlapping consecutive intervals, called *bins*. When a value x is passed to a histogram, the bin i is found which contains x and the associated count n_i is incremented. A multi-dimensional histogram accepts value tuples $(x, y, z, \dots)$. For each value in the tuple an independent sequence of bins exists, which we call an *axis*. The count is now looked up based on a multi-dimensional index $(i, j, k, \dots)$ obtained from the mappings $(x \to i, y \to j, z \to k, \dots)$.

Boost.Histogram generalizes the classic histogram concept in three ways.

Custom axis objects. The mapping from input value to index provided by an axis $x \to i$ can be completely user-defined. Values are not restricted to numbers; arbitrary C++ objects can be used as values when an appropriate mapping is provided, for example, strings can be used. Bins do not have to be consecutive, they can have gaps and arbitrary hyperdimensional shapes. Hexagonal binning is possible, or bins that are HEALPix[1] pixels [4] (although neither are currently implemented in the core library).

Custom accumulators. The counts in a generalized histogram can be replaced by user-defined accumulators. Accumulators can take any number of *samples* which, like values, can be arbitrary C++ objects. Values and samples are passed to the generalized histogram together; values are converted to indices which address the accumulator, and samples are passed to the accumulator. The library provides a few standard accumulators, such as accumulators to compute mean and variance of samples in each bin. This is called a *profile* in the ROOT framework and is a separate class, while it is just a choice of accumulator in a normal histogram in Boost.Histogram.

[1] Hierarchical Equal Area isoLatitude Pixelation of a sphere.

Custom storages. It is possible to completely customize how counts (or accumulators) are allocated and addressed in memory. The sub-component responsible for this is called a *storage*. The histogram class converts the multi-dimensional index $(i, j, k, \dots)$ into a single linear index, which is passed to the storage. The standard storage allocates memory upfront for all cells and places the counts sequentially in memory, using the linear index as a memory offset. This gives the best look-up performance and is space-efficient for densely filled histograms, since it is not necessary to store the linear index explicitly for each cell. The downside of dense storage is that all cells take up space, even if they contain zero counts. The library also supports sparse storage based on an STL-compliant hash map. Only cells with non-zero counts use space in such a sparse storage, but each cell has to store its linear index in addition to the payload and incurs some overhead for hash-based addressing.

The memory allocation strategy of a storage can be customized as well. The standard storage allocates memory dynamically from the heap, but the library has builtin support for a storage based on a fixed-size stack-based memory buffer. The latter allows one to efficiently create and destroy many small histograms, for example. The builtin `unlimited_storage` dynamically allocates memory to grow the counter capacity as needed, starting with a single byte per cell up to arbitrarily many bytes (limited by available memory only). This storage offers a unique no-overflow-guarantee and is memory-efficient in high dimensions, where the number of cells is large and the small memory footprint per cell pays off.

The three sub-components of a generalized histogram, axis types, storages, and accumulators, are orthogonal. This means that any sub-component can be replaced or modified independently of the others. Orthogonal design is very powerful since it offers a huge customization potential from all possible combinations.

2.2 Notable features

We briefly list some of the other notable features here.

Arithmetic operators. Histograms support the standard math operators `+`, `-`, `*`, and `/`.

Growing axes. A standard axis has a fixed value range and number of bins, defined at the time of construction. The library also supports growing axes; such an axis has an initial range and bin number but grows with the input. If a value is encountered that would fall outside of the axis range, the axis range is extended to contain the value and the number of bins is increased.

Optional underflow and overflow bins. Each axis of the histogram can have optional underflow and overflow bins. These are extra bins beyond the defined range of the axis that count all values which fall below the smallest value on the axis or above the largest value, respectively. The existence or absence of these bins is mostly transparent for the user. They are very useful to detect outliers and to offer lossless reductions (explained below).

Reductions. The library offers tools to reduce the memory footprint of a histogram by reducing the number of bins of an axis or by completely removing an axis. The number of bins in an axis can be reduced by shrinking its value range and/or by merging any number of adjacent bins into one larger bin. Likewise, an axis can be removed completely by summing over its bins (a so called *projection*). In both cases, the presence of underflow and overflow bins guarantees that the reduced histogram is identical to one obtained by filling with the original values. This is not generally possible when underflow and overflow bins are missing.

Thread-safe filling of histograms. Filling histograms with values is not thread-safe in general, but the library offers a builtin thread-safe counter based on a `std::atomic` integer type and a thread-safe locking infrastructure for storages; the latter is needed when growing axes are present, which can trigger a resizing of the storage.

Static and dynamic axis configuration. The library supports both histograms with static and dynamic axis containers. A histogram with a static axis container is fixed at compile-time in the number and types of axes. The number and axis types can vary at run-time for a dynamic axis holder. The static axis container produces histograms which are more performant, as the compiler can find more opportunities to optimize the code. When Boost.Histogram is used as a backend in a run-time environment like Python, however, axes must be configurable at run-time. The performance difference vanishes when the histogram is filled with chunks of values at once, see next item.

Filling individual values and chunks of values. Histograms can be filled with one value at once or by passing a contiguous chunk of values at once. Both cases are handled by separate code segments, which were highly optimized for performance. Passing contiguous chunks of values is up to five times faster for chunks of moderate size (32768 values) than filling one value at a time, and therefore preferred when chunks are available. Using the chunk code for single values would be slow, however, and therefore the other optimized code path exists.

3 boost-histogram for Python

Boost.Histogram was developed with Python in mind. Original prototype bindings using Boost.Python were included in the draft first submitted to Boost; however, to keep the library focused they were removed before the library was accepted. New bindings based on PyBind11 were developed as part of the Scikit-HEP family of Python packages.

The new bindings were designed around four key ideas based on a study of the libraries in Table 1: *Design*, *Flexibility*, *Speed*, and *Distribution*. No single existing library provided a strong entry in all four of these areas.

3.1 Design

The design of boost-histogram follows Boost.Histogram closely, with appropriate changes to adapt to Python and interactive usage. The description of a Histogram matches that described in Section 2.1.

The features listed in Section 2.2 are mostly available to Python users. Python histograms support **arithmetic operators**. Most axis types support **growing axes** and **optional underflow and overflow bins**. The special storage that enables **thread-safe filling of histograms** is provided. And, since Python is a dynamic language, the bindings take advantage of **dynamic axis configuration** to set up all histograms. The other features adapted or specific to Python are listed below.

Filling with chunks of values. The Python library supports filling contiguous chunks of values, based on the Boost.Histogram C++ feature. Since this looks like a NumPy array operation [5], the differences should be noted. The data to be filled must be in the correct format (doubles or ints), and must be continuous (slices and other operations in NumPy can create arrays with strides, and Boost.Histogram's fill feature does not operate on strides). If these conditions are not met, a copy is made, affecting the performance and memory usage slightly.

UHI (reductions). Unified Histogram Indexing (UHI) was developed to provide a way for histograms to be manipulated and reduced in Python in a natural, concise way, and to decouple the "tags" describing actions and the histogram library performing the actions. Custom actions can be developed by users. See Section 3.1.1 for more details.

Python interaction. Histograms (and the other objects provided) support the standard Python copy, deepcopy, and pickle protocols, and have docstrings, signatures (enhanced in Python 3), IPython keyboard completions, and natural textual representation, providing a native experience for Python.

NumPy interaction. Special care went into making the library interact gracefully with NumPy. The underlying bin data for even accumulator based storages is available via mutable, no copy access. The Histogram object conforms to the Python buffer protocol. A special method is provided to produce NumPy style output tuples. For input, boost-histogram provides a NumPy module that provides functions identical to the three NumPy histogram functions, but powered by boost-histogram (and as such, up to 10 times faster than native NumPy histograms), and with extra keyword only arguments to provide a way to return boost-histogram objects instead of tuples.

3.1.1 UHI

Unified Histogram Indexing (UHI) was inspired by an early design for Aghast (see Section 4.1). The key advancement in UHI was the design of a general API for providing "tag" types that can be used to perform operations in indexing. These tags can be provided by one library and used by another, and new tags can be written by users. And indexing is carefully designed to either behave like a NumPy array index if the syntax is allowed on both arrays and histograms, or to give an error on one or the other. The goal was to minimize situations where indexing will silently fail if you interchange histograms and arrays.

UHI allows single bin and array setting and accessing, and even provides optional access to the flow bins when setting. The third element of the Python slice syntax is used to perform actions on axes, such as rebinning or summation. While not currently implemented in version 0.6.2, the UHI specification includes arbitrary actions on axes. UHI was originally designed to provide an action per axis, though later based on user requests, it was expanded to allow a mapping to be provided, allowing a small number of axes to be operated on without having to explicitly list every axis.

3.1.2 Axes

Axes are available in a special augmented tuple. This tuple can accept any method a single axis can, and performs the property access or the method call on each axis, returning a tuple or ndarray as appropriate. This reduces many traditionally complex procedures into a simple, concise one or two lines.

3.2 Flexibility

Great care was taken to keep the incredible flexibility of Boost.Histogram intact in a dynamic, pre-compiled environment. A collection of over 20 axis types is provided, including **Regular** binning, **Integer** binning, and **Variable** binning, each with variants for underflow/overflow and growth. Both integer and string category axes are provided, also with optional growth. And regular binning can have a **transform** applied; the transform can be created by the user

with a compiled callback using a language like Numba [6] with a maximum of about 7% penalty in speed over a precompiled function in C++.

Seven storages are provided as well, including three "complex" storages made from accumulators. The **Weight** storage provides high precision weighted sums that track the count as well. The **Mean** and **WeightedMean** storages can be used to produce "profile" histograms that track the average of a weight in each bin, rather than the total number of entries observed. In some situations, a N-dimensional profile can replace a N+1-dimensional histogram.

3.3 Speed

Performance of the library was a key design consideration. For a significantly large 1D dataset, boost-histogram was measured to be 2.4 times faster than NumPy for regular bin spacing, which NumPy is also optimized for in 1D. For a 2D dataset, NumPy does not have special optimizations for regular binning, and in that case boost-histogram is 13 times faster. Both tests were performed with a single thread; with multithreaded filling boost-histogram was observed to gain another factor of 2-4, depending on the number of physical threads available on the hardware and the scale of the problem.

3.4 Distribution

To be used in the modern scientific Python ecosystem, a library must fulfill a variety of packaging criteria. All modern packages should be hosted on PyPI.org and should provide **wheels**. Pure Python packages only need to provide a "universal wheel", but compiled packages like boost-histogram need to provide a collection of wheels, one for each Python version and each supported platform. A system was developed for boost-histogram to build these wheels on the Azure DevOps cloud platform Continuous Integration (CI) system; these are automatically built whenever a new release is made via a manual request in the web API. The system developed for boost-histogram is now directly in use by at least three other Scikit-HEP packages.

One of the challenges unique to boost-histogram was the need for C++14 support by the compiler; the classic "ManyLinux1" wheels were not sufficient for compiling boost-histogram. The new ManyLinux2010 format was being finalized while this was being developed, but the older ManyLinux1 specification was still in heavy use (primary due to older versions of pip, the package installer for Python), so a special docker image was created that has a newer compiler but mostly conforms to ManyLinux1. This allows boost-histogram to support both specifications.

The wheel building system (and therefore boost-histogram) supports Python 2.7 and Python 3.6 through 3.8 on 64-bit architectures. On Linux systems, it also supports Python 3.5, partially because the infrastructure is already there in the official tooling and Linux tends to have older Python versions available in official channels. For ManyLinux1 and Windows, 32-bit architectures are also supported. As an organization, Scikit-HEP has agreed on a slightly modified version of NumPy Enhancement Proposal (NEP) 29, which outlines a community Python version support policy. Future versions of boost-histogram retain the right to remove versions that are no longer supported under that plan; the wheel building tools provide the ability to select versions and platforms for each project.

If a user is on an unsupported platform, such as specialized Linux distributions like Clear-Linux or Alpine, the only requirements to build boost-histogram are a C++14 compatible compiler. Every dependency is header-only and supplied in the source tarball available on PyPI or via git submodules when building from the source repository.

Another way to distribute packages that has strong community adoption in the sciences and is rapidly gaining support in HEP is Conda. Conda is an alternative package manager designed to supply binary packages to a wide variety of systems. In contrast to pip, Conda never builds from source on the target machine, is less Python-centric, and provides a complete compiler stack with minimal dependencies on the underlying system. The most popular community source of packages and build infrastructure, Conda-forge has successfully been used to provide a complete ROOT build in for HEP [7], and is being used to supply a growing number of HEP packages, both with and without Python. Conda-forge packages of boost-histogram cover all supported targets of Conda-forge except for the rapidly disappearing Python 2.7 on Windows target; supported platforms include ARM and PowerPC.

4 The Scikit-HEP family

The boost-histogram project is part of a larger plan for the Scikit-HEP family of tools (see Figure 2).

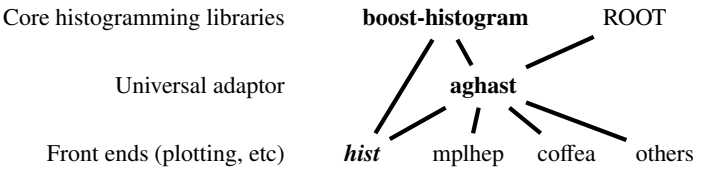

Figure 2. Plan for Scikit-HEP projects and relationship between them.

4.1 Aghast

Aghast is a histogramming library that does not fill histograms and does not plot them. It is a conversion library, designed to make all the other histogramming libraries talk together. It defines an in-memory format for histograms, using flatbuffers. It understands a superset of all the functionality and binning methods that the other libraries support, and therefore can convert between them. It can convert to and from boost-histogram, ROOT (without a ROOT dependency by using uproot [8]), NumPy, and more.

4.2 Hist

Hist is a project that will provide an analyst-centric interface for histograms. It will use boost-histogram as the computational backend, but will connect to and depend on packages that are not allowed by the core boost-histogram package. It will use Aghast to facilitate opening and saving histograms in different formats. It will also provide ways to directly plot histograms and will interact nicely with the mplhep package that is being developed. It will also initially provide extra shortcuts designed for interactive use that might eventually be accepted into the underlying boost-histogram package as well.

5 Summary

Histograms in Python has been a weak point in the adoption of Python in HEP. The future for Histograms in Python is now an exciting one, as high performance histograms represented as objects are now available through boost-histogram. And future packages such as Hist will provide a simple interface for analysts.

Acknowledgments

This work was supported by the National Science Foundation under Cooperative Agreements OAC-1836650 and OAC-1450377.

References

[1] E. Rodrigues et al. (2019), CHEP 2019, https://indico.cern.ch/event/773049/contributions/3476182/

[2] R. Brun, F. Rademakers, Nucl. Instrum. Meth. **A389**, 81 (1997)

[3] B. Gough, *GNU scientific library reference manual* (Network Theory Ltd., 2009)

[4] K.M. Gorski, E. Hivon, A.J. Banday, B.D. Wandelt, F.K. Hansen, M. Reinecke, M. Bartelmann, The Astrophysical Journal **622**, 759 (2005)

[5] S.v.d. Walt, S.C. Colbert, G. Varoquaux, Computing in Science & Engineering **13**, 22 (2011)

[6] S.K. Lam, A. Pitrou, S. Seibert, *Numba: A LLVM-based Python JIT compiler*, in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Association for Computing Machinery, New York, NY, USA, 2015), LLVM '15, pp. 1–6, ISBN 9781450340052, https://doi.org/10.1145/2833157.2833162

[7] C. Burr, H.F. Schreiner, E. Guiraud, J.C. Villanueva, B. Couturier (2019), CHEP 2019, https://indico.cern.ch/event/773049/contributions/3473243/

[8] J. Pivarski, P. Das, C. Burr, D. Smirnov, M. Feickert, T. Gal, N. Smith, O. Shadura, N. Biederbeck, M. Proffitt et al., *scikit-hep/uproot: 3.11.2* (2020), https://doi.org/10.5281/zenodo.3631827