Boost-histogram: High-Performance Histograms as Objects

Henry Schreiner[‡*], Hans Dembinski[§], Shuo Liu[¶], Jim Pivarski[‡]

https://youtu.be/ERraTfHkPd0

Abstract—Unlike arrays and tables, histograms in Python have usually been denied their own object, and have been represented as a single operation producing several arrays. Boost-histogram is a new Python library that provides histograms that can be filled, manipulated, sliced, and projected as objects. Building on top of the Boost libraries' Histogram in C++14 provided interesting distribution and design challenges with useful solutions. This is meant to be a foundation that others can build on; in the Scikit-HEP project[1], a physicist friendly front-end "Hist" and a conversion package "Aghast" are already being designed around boost-histogram.

Index Terms—Histogram, Analysis, Data processing, Data reduction, NumPy, Aggregation

Motivation

As an example of a problem that becomes much easier with histograms as objects, let's look at the Python 3 adoption of several libraries using PyPI download statistics. There are three columns of interest: The package name, the date of the download, and the Python version used when downloading the package. In order to look at trends, you will want to answer questions about the download behavior over time ranges, such as what is the fraction of Python 2 downloads out of all downloads for each month. Let's look at what a solution to this would entail using traditional histogramming methods [NumPy]:

- *Date*: You could make a histogram over datetime objects, but then you will be responsible for finding the bin range (dates are just large numbers), probably using `np.searchsorted` on the edges array, and then making slices in the binned array yourself.
- *Python version*: You would have to force some sort of artificial binning scheme, such as one with edges `[2, 3.0, 3.59, 3.69, 3.79, 4]`, in order to collect information for each Python version of interest. You would have to use a 2D array, and keep the selections/edges straight yourself; in practice, you would probably just create a Python dict of 1D histograms for each major version.
- *Package names*: This would require making a dict and storing each 2D (or set of 1D) histograms manually. NumPy does not support category axes or strings.

If your data doesn't fit into memory, you will have to build in the batching and combining yourself. For each piece.

Now look at this with an object-based Histogram library, such as boost-histogram:

- *Package names*: This can be string categories.
- *Python version*: You could simply multiply by 10 and make these int categories, or just use string categories.
- *Date*: Use a regular spaced binning from start to stop in the resolution you are interested, such as months. Use the loc indexer to convert when slicing. No manual tracking or searching. Use rebinning to convert months into years in one step.

In the object-based version, you fill once. If your data doesn't fit into memory, just fill in batches. The API for ND histograms is identical to 1D histograms, so you don't have to use different functions or change significant portions of code even if you add a new axes later.

Now let's look at using the object to make a series of plots, with one shown in Figure 1[2]. The code required to make the plot is shown below, with minor formatting details removed.

```python
for name in hist.axes[0]:
    fig, ax = plt.subplots()
    ax.set_title(name)
    for vers in hist.axes[1]:
        dhist = hist[bh.loc(name), bh.loc(vers), :]
        (dt,) = d.axes.centers
        xs = mpl.dates.date2num(pd.to_datetime(dt))
        ax.plot_date(xs, dhist, label=f"{vers/10}")
```

Note how all the computation, and the version information is stored in a single histogram object. The datetime centers are accessible after the package and version number are selected. Looping over the categories is trivial. Since the histogram is already filled, there are no other loops over the data to slow down manipulation. We could rebin or set limints or sum over axes cleanly as well.

* Corresponding author: henryfs@princeton.edu
‡ Princeton University
§ TU Dortmund
¶ Sun Yat-sen University

1. https://scikit-hep.org

2. Code available at https://github.com/scikit-hep/scikit-hep-orgstats

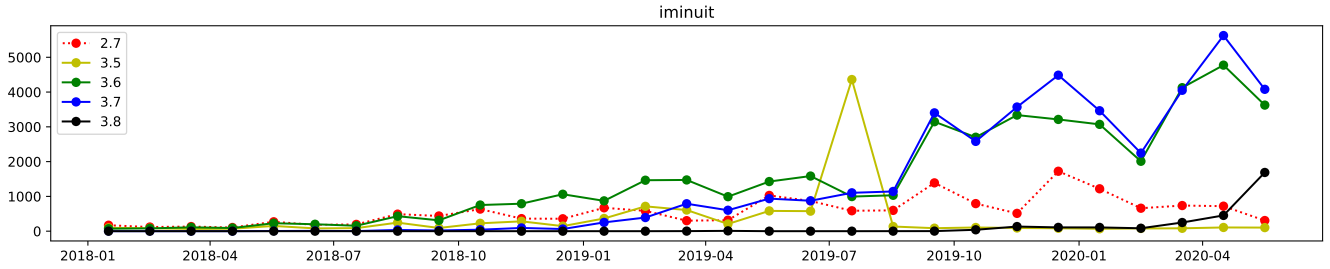

***Fig. 1:** A downloads vs. time histogram plot for iMinuit [iMinuit] by Python version, made with Matplotlib [Matplotlib].*

Introduction

In the High Energy Physics (HEP) community, histogramming is vital to most of our analysis. As part of building tools in Python to provide a friendly and powerful alternative to the ROOT C++ analysis stack [ROOT], histogramming was targeted as an area in the Python ecosystem that needed significant improvement. The "histograms are objects" mindset is a general, powerful way of interacting with histograms that can be utilized across disciplines. We have built boost-histogram in cooperation with the Boost C++ community [Boost] for general use, and also have separate more specialized tools built on top of boost-histogram that customize it for HEP analysis (which will be discussed briefly at the end of this paper).

At the start of the project, there were many existing histogram libraries for Python (at least 24 were identified by the authors), but none of them fulfilled the requirements and expectations of users coming from custom C++ analysis tools. Four key areas were identified as key to a good library for creating histograms: Design, Flexibility, Performance, and Distribution.

Before we continue, a brief description of a histogram should suffice to set the stage until we describe boost-histogram's approach in more detail. A histogram reduces an arbitrarily large dataset into a finite set of bins. A histogram consists of one or more *axes* (sometimes called "binnings") that describe a conversion from *data coordinates* to *bin coordinates*. The data coordinates may be continuous or discrete (often called categories); the bin coordinates are always discrete. In NumPy [NumPy], this conversion is internally derived from a combination of the `bin` and `range` arguments. Each *bin* in the histogram stores some sort of aggregate information for each value that falls into it via the axes conversion. This is a simple sum in NumPy. When something besides a sum is used, this is a "generalized histogram", which is called a `binned_statistic` in `scipy.stats` [SciPy]; for our purposes, we will avoid this distinction for the sake of brevity, but our histogram definition does include generalized histograms. Histograms often have an extra "weight" value that is available to this aggregate (a weighted sum in NumPy).

Almost as important as defining what a histogram is limiting what a histogram is not. Notice the missing item above: a histogram, in this definition, is not a plot or a visual aid. It is not a plot any more than a NumPy array is a plot. You can plot a Histogram, certainly, and customisations for plotting are useful (much as Pandas has custom plotting for Series [Pandas]), but that should not part of a core histogram library, and is not part of boost-histogram (though most tutorials include how to plot using Matplotlib [Matplotlib]).

The first area identified was **Design**; here many popular libraries fell short. Histograms need to be represented as an object, rather than a collection of NumPy arrays, in order to naturally manipulate histograms after filling. You should be able to continue to fill a histogram after creating it as well; filling in one pass is not always possible due to memory limits or live data taking conditions. Once a histogram is filled, it should be possible to perform common operations on it, such as rebinning to a courser binning scheme, projecting on a subset of axes, selecting a subset of bins then working with or summing over just that piece, and more. You should be able easily sum histograms, such as from different threads. You also should be able to easily access the transform between data coordinates and bin coordinates for each axes. Axis should be able to store extra information, such as a title or label of some sort, to assist the user and external plotting tools.

The second area identified was **Flexibility**; there are a wide range of things a histogram should be able to do; these traditionally are split into different functions and objects, but as we show, a clear, consistent design makes it possible to unify around a single object. Axes should support several forms of binning: variable width binnings, regularly spaced binnings (a performance-optimized subset of variable binning), and categorical binning. Out-of-range bins (called flow bins, discussed later) are also key for enabling lossless sums over a partial collection of axes. Axes should also be able to optionally grow when a fill is out of range instead. The bins themselves should support simple sums, like NumPy, but should also support means (sometimes called profile histograms). High-precision weighted summing is also useful. Finally, if you add a sample parameter to the fill, you can also keep track of the variance for each bin.

The third area identified was **Performance**; when dealing with very large datasets that will not fit in memory, the filling performance becomes critical. High performance filling is also useful in real-time applications. A highly performance histogram library should support fast filling with a compiled loop, it should avoid reverting to a slower $\mathcal{O}(n)$ lookup when filling a regularly spaced axes, and it should be able to take advantage of multiple cores when filling from a large dataset. NumPy, for example, does do well for a single regularly spaced axes, but it still does not optimize for two regularly spaced axes (an image is an example of a common regularly spaced 2D histogram).

The fourth and final area identified was **Distribution**. A great library is not useful if no one can install it; it is especially important that students and inexperienced users be able to install the histogramming package. This is one of Python's strengths compared to something like C++, but the above requirements necessitate compiled components, so this is important to get right. It also needed to work flawlessly in virtual environments and in the Conda package manager. It also needed to be available on as many platforms and for as many Python versions as possible to

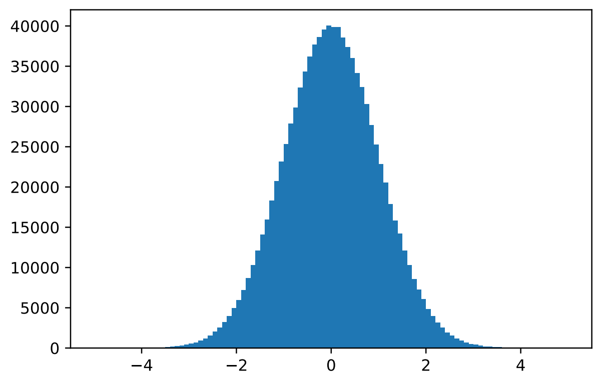

Fig. 2: An example of a 1D-histogram.

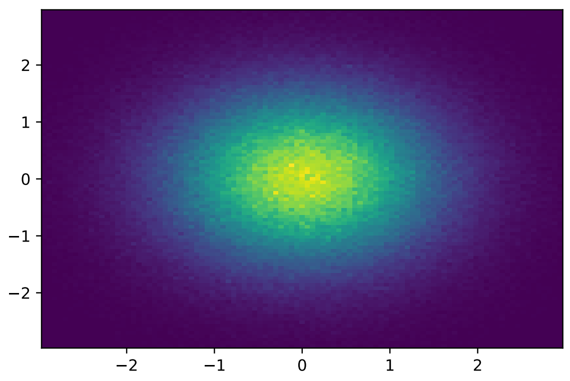

Fig. 3: An example of a 2D-histogram.

support both old and new data acquisition and analysis systems.

About a year ago, a new C++14 library was proposed to the Boost C++ libraries called Boost.Histogram; it was unanimously accepted and released as part of the Boost C++ libraries version 1.70 after the review process. It was a well designed header-only package that fulfilled exactly what we wanted, but in C++14 rather than Python. A proposal was made to get a full-featured Python binding developed as part of an institute for sustainable software for HEP [IRIS-HEP], as one of the foundations for a Python based software stack being designed to be part of the Scikit-HEP community [SkHEP]. We built boost-histogram for Python in close collaboration with the original Histogram for Boost author, Hans Dembinski, who had always intended Boost.Histogram to be accessible from Python. Due to this close collaboration, concepts and design closely mimic the spirit of the Boost counterpart.

An example of the boost-histogram library approach, creating a 1D-histogram and adding values, is shown below, with results plotted in Figure 2:

```
import boost_histogram as bh
import numpy as np
import matplotlib.pyplot as plt

ax = bh.axes.Regular(100, start=-5, stop=5)
hist = bh.Histogram(ax)

hist.fill(np.random.randn(1_000_000))

plt.bar(hist.axes[0].centers,
        hist.view(),
        width=hist.axes[0].widths)
```

For future code snippets, the imports used above will be assumed. Using `.view()` is optional, but is included to make these explicit. You can access `ax` as `hist.axes[0]`. Note that boost-histogram is not plotting; this is simply accessing histogram properties and leveraging existing Matplotlib functionality. A similar example, but this time in 2D, is shown in Figure 3, illustrating the identical API regardless of the number of dimensions:

```
hist_2d = bh.Histogram(bh.axis.Regular(100, -3, 3),
                       bh.axis.Regular(100, -3, 3))

hist_2d.fill(np.random.randn(1_000_000),
             np.random.randn(1_000_000))

X, Y = hist_2d.axes.centers
plt.pcolormesh(X.T, Y.T, hist_2d.view().T)
```

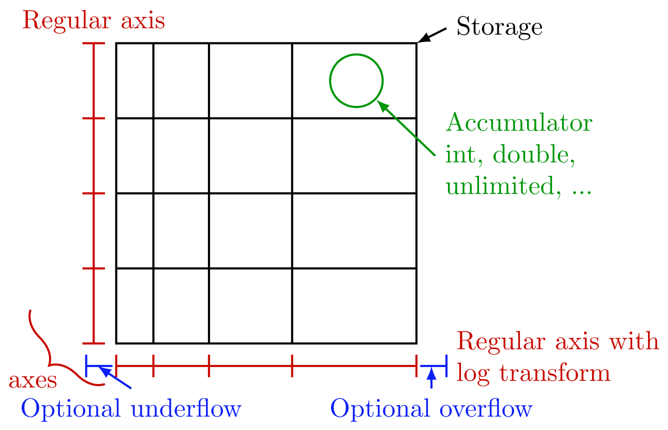

Fig. 4: The components of a histogram, shown for a 2D histogram.

Boost-histogram is available on PyPI and conda-forge, and the source is BSD licensed and available on GitHub[3]. Extensive documentation is available on ReadTheDocs[4].

The Design of a Histogram

Let's revisit our description of a histogram, this time mapping boost-histogram components to each piece. See Figure 4 for an example of how these visually fit together to create an 2D histogram.

The components in a bin are the smallest atomic piece of boost-histogram, and are called **Accumulators**. Four such accumulators are available. `Sum` just provides a high-accuracy floating point sum using the Neumaier algorithm [Neu74], and is automatically used for floating point histograms. `WeightedSum` provides an extra term to allow sample sizes to be given. `Mean` stores a mean instead of a sum, created what is sometimes called a "profile histogram". And `WeightedMean` adds an extra term allowing the user to provide samples. Accumulators are like a 0D or scalar histogram, much like dtypes are like 0D scalar arrays in NumPy.

The above accumulators are then provided in a container called a **Storage**, of which boost-histogram provides several. The available storages include choices for the four accumulators listed

3. https://github.com/scikit-hep/boost-histogram
4. https://boost-histogram.readthedocs.io

above (the storage using `Sum` is just called `Double()`, and is the default; unlike the other accumulator-based storages it provides a simple NumPy array rather than a specialized record array when viewed). Other storages include `Int64()`, which stores integers directly, `AtomicInt64`, which stores atomic integers, so can be filled from different threads concurrently, and `Unlimited()`. which is a special growing storage that offers a no-overflow guarantee and automatically uses the least possible amount of memory for a dense uniform array of counters, which is very helpful for high-dimensional histograms. It also automatically converts to doubles if filled with a weighted fill or scaled by a float.

The next piece of a histogram is an **Axis**. A `Regular` axis describes an evenly spaced binning with start and end points, and takes advantage of the simplicity of the transform to provide $\mathcal{O}(1)$ computational complexity. You can also provide a **Transform** for a `Regular` axes; this is a pair of C function pointers (possibly generated by a JIT compiler [Numba]) that can apply a function to the transform, allowing for things like log-scale axes to be supported at the same sort of complexity as a `Regular` axis. Several common transforms are supplied, including log and power spacings. You can also supply a list of bin edges with a `Variable` axis. If you want discrete axes, `Integer` provides a slightly simpler version of a `Regular` axes, and `IntCategory`/`StrCategory` provide true non-continuous categorical axes for arbitrary integers or strings, respectively. Most axes have configurable end behaviors for when a value is encountered by a fill that is outside the range described by the axis, allowing underflow/overflow bins to be turned off, or replaced with growing bins. All axes also have a metadata slot that can store arbitrary Python objects for each axis; no special meaning is applied by boost-histogram, but these can be used for titles, units, or other information.

An example of a custom transform applied to a `Regular` axis is shown below using Numba to create C pointers; any ctypes pointer is accepted.

```python
import numba

@numba.cfunc(numba.float64(numba.float64))
def exp(x):
    return math.exp(x)

@numba.cfunc(numba.float64(numba.float64))
def log(x):
    return math.log(x)

transform_log = bh.axis.transform.Function(log, exp)

bh.axis.Regular(10, 1, 4, transform=transform_log)
```

You need to provide both directions in the transform, so that boost-histogram can add values to bins and find bin edges. Note: don't actually use exactly this code; there is a `bh.axis.transform.log` already compiled in the library.

A **Histogram** is the combination of a storage and one or more axes. Histograms always manage their own memory, though they provide a view of that storage to Python via the buffer protocol and NumPy. Histograms have the same API regardless of whether they have one axes or thirty-two, and they have a rich set of interactions defined, which will be the topic of the next section. This is an incredibly flexible design; you can orthogonally combine any mixture of axes and storages with associated accumulators, and in the future, new axes types or accumulators and storages can be added.

Interactions with a Histogram

A Histogram supports a variety of operations, many of which use Python's syntax to be expressed naturally and succinctly. Histograms can be added, copied, pickled (special attention was paid to ensure even accumulator storages are pickled quickly and efficiently), and used most places a NumPy array is accepted. Scaling a histogram can be done simply by using Python's multiplication and division operators.

Conversion to a NumPy array was carefully designed to provide a comfortable interface for Python users. The "flow" bins, which are the bins that are used when an event is encountered outside the range of the current axis, are an essential feature for partial summations. These extra bins are not as common in NumPy based analyses (though you can create flow bins manually in NumPy by using $\pm\infty$), so these generally are not needed or expected when converting to an array. The array interface and all external methods do not include flow bins by default, but they can be activated by passing `flow=True` to any of the methods that could be affected by flow bins. You can directly access a view of the data without flow bins with `.view()`, and you can include flow bins with `.view(flow=True)`. The stride system is descriptive enough to avoid needing to copy memory in either case. Views of accumulator storages are NumPy record arrays, enhanced with property-based access for the fields as well as common computed properties, like the variance. Finally, there is an explicit `.to_numpy()` method that returns the same tuple you would get if you used one of the `np.histogram` functions.

Axes are presented as a property returning an enhanced tuple. You can use access any method or property on all axes at once directly from the `AxesTuple`. Array properties (like edges) are returned in a shape that is ready for broadcasting, allowing natural manipulations directly on the returned values. For example, the following snippet computes the density of a histogram, regardless of the number of dimensions:

```python
# Compute the "volume" of each bin (useful for 2D+)
volumes = np.prod(hist.axes.widths, axis=0)

# Compute the density of each bin
density = hist.view() / hist.sum() / volumes
```

Unified Histogram Indexing

Indexing in boost-histogram, based on a proposal called Unified Histogram Indexing (UHI)[5], allows NumPy-like slicing and is based on tags that can be used cross-library. They can be used to select items from axes, sum over axes, and slice as well, in either data or bin coordinates. One of the benefits of the axes based design is that selections that traditionally would have required multiple histograms now can simply be represented as an axes in a single histogram and then UHI is used to select the subset of interest.

The key design is that any indexing expression valid in both NumPy and boost-histogram should return the same thing regardless of whether you have converted the histogram into an array via `.view()` or `np.asarray` or not. Freedom to access the unique parts of boost-histogram are only granted through syntax that is not valid on a NumPy array. This is done through special tags that are not valid in NumPy indexing. These tags do not depend on the internals of boost-histogram, however, and could be written

5. https://boost-histogram.readthedocs.io/en/latest/usage/indexing.html

by a user or come from a different library; the are mostly simple callables, with minor additions to make their *repr*'s look nicer.

There are several tags provided: `bh.loc(float)` converts a data-coordinate into bin coordinates, and supports addition/subtraction. For example, `hist[bh.loc(2.0) + 2]` would find the bin number containing 2.0, then add two to it. There are also `bh.underflow` and `bh.overflow` tags for accessing the flow bins.

Slicing is supported, and works much like NumPy, though it does return a new Histogram object. You can use tags when slicing. A single value, when mixed with a slice, will select out a single value from the axes and remove it, just like it would in NumPy (you will see later why this is very useful). Most interesting, though, is the third parameter of a slice - normally called the step. Stepping in histograms is not supported, as that would be a set of non-continuous but non-discrete bins; but you can pass two different types of tags in. The first is a "rebinning" tag, which can modify the axis -- `bh.rebin(2)` would double the size of the bins. The second is a reduction, of which `bh.sum` is provided; this reduces the bins along an axes to a scalar and removes the axes; `builtins.sum` will trigger this behavior as well. User provided functions will eventually work here, as well. Endpoints on these special operations are important; leaving off the endpoints will include the flow bins, including the endpoints will remove the flow bins. So `hist[::sum]` will sum over the entire histogram, including the flow bins, and `hist[0:len:sum]` will sum over the contents of the histogram, not including the flow bin. Note that Python's *len* is a perfectly valid in this system - start and stop tags are simply callables that accept an axis and return an index from `-1` (underflow bin) to `len(axis)+1` (overflow bin), and axes support `len()`.

Setting is also supported, and comes with one more nice feature. When you set a histogram with an array and one or more endpoints are empty and include a flow bin, you have two options; you can either match the inner size, which will leave the flow bin(s) alone, or you can match the total size, which will fill the flow bins too. For example, in the following snippet the array can be either size 10 or size 12:

```
hist = bh.Histogram(bh.axis.Regular(10, 0, 1))
hist[:] = np.arange(10) # Fills regular bins
hist[:] = np.arange(12) # Fills flow bins too
```

You can force the flow bins to be explicitly excluded if you want to by adding endpoints to the slice:

```
hist[0:len] = np.arange(10)
```

Finally, for advanced indexing, dictionaries are supported, where the key is the axis number. This allows easy access into a large number of axes, or simple programmatic access. With dictionary-based indexing, Ellipsis are not required. There is also a `.project(*axes)` method, which allows you to sum over all axes except the ones listed, which is the inverse to listing `::sum` operations on the axes you want to remove.

Performance when Filling

A histogram can be viewed as a lossy data compression tool; you lose the exact details of each data point, but you have a have a representation that does not depend on the number of data points and has several very useful properties for computation. One common use beyond plotting is distribution fitting; you can fit an arbitrarily large number of data points to a distribution as long as you choose a binning dense enough to capture the details of your distribution function. The performance of the fit is based on the number of bins, rather than the number of measurements made. Many distribution fitting packages available outside of HEP, such as lmfit [LMFIT], are designed to work with binned data, and binned fits are common in HEP as well.

Setup	Single threaded	X	Multithreaded	X
NumPy 1D	74.5 ± 2.4 ms	1		
BH 1D	41.6 ± 0.7 ms	1.8	13.3 ± 0.2 ms	5.5
BHNP 1D	43.1 ± 0.8 ms	1.7	13.8 ± 0.2 ms	5.4
NumPy 2D	874 ± 22 ms	1		
BH 2D	77.6 ± 0.6 ms	11	28.7 ± 0.7 ms	30
BHNP 2D	85 ± 3 ms	10	29.6 ± 0.5 ms	29

TABLE 1: *Comparison of several filling methods and NumPy. BH stands for boost-histogram object mode (as seen above). BHNP stands for boost-histogram NumPy clone, which provides the same interface as NumPy but powered by Boost.Histogram calculations. Multithreaded was obtained by passing `threads=8` while filling. The X column is a comparison against NumPy. Measurements done on an 8 core 16 MBP, 2.4 GHz, Regular binning, 10M values, 32-bit floats.*

Filling performance was a key design goal for boost-histogram. In Table 1 you can see a comparison of filling methods with NumPy. The first comparison, a 1D histogram, shows a nearly 2x speedup compared to NumPy on a single core. For a 1D `Regular` axes, NumPy has a custom fill routine that takes advantage of the regular binning to avoid an edge lookup. If you use multiple cores, you can get an extra 2x-4x speedup. Note that histogramming is not trivial to parallelize. Internally, boost-histogram is just using simple Python threading and relying on releasing the GIL while it fills multiple histograms; the histograms are then added into your current histogram. The overhead of doing the copy must be small compared to the fill being done.

If we move down the table to the 2D case, you will see Boost-histogram pull away from NumPy's 2D regular bin edge lookup with an over 10x speedup. This can be further improved to about 30x using threads. In both cases, boost-histogram is not actually providing specialized code for the 1D or 2D cases; it is the same variadic vector that it would use for any number and any mixture of axes. So you can expect excellent performance that scales well with the complexity of your problem.

The rows labeled "BHNP" deserve special mention. A special module is provided, *bh.numpy*, that contains functions that exactly mimic the functions in NumPy. They even use a special, internal axes type that mimics NumPy's special handling of the final upper edge, including it in the final bin. You can use it as a drop-in replacement for the histogram functions in NumPy, and take advantage of the performance boost available. You can also add the `threads=` keyword. You can pass `histogram=bh.Histogram` to return a Histogram object, and you can select the storage with `storage=`, as well. Combined with the ability to convert Histograms via `.to_numpy()`, this should enable smooth transitions between boost-histogram and NumPy for Histogram filling.

One further performance benefit comes from the flexibility of combining axes. In a traditional, NumPy based analysis, you may have a collection of related histograms with different cuts or criteria for filling. We have already seen that it is possible to use axis and then access the portion you want later with indexing; but if you have categories or boolean selectors, you can still combine multiple histograms into one. Then you no longer loop over the input multiple times, but just once, filling the histogram, and then make your selections later. Here is an example:

```
value_ax = bh.axis.Regular(100, -5, 5)
valid_ax = bh.axis.Integer(0, 2,
                           underflow=False,
                           overflow=False)
label_ax = bh.axis.StrCategory([], growth=True)

hist = bh.Histogram(value_ax, valid_ax, label_ax)

hist.fill([-2, 2, 4, 3],
          [True, False, True, True],
          ["a", "b", "a", "b"])

all_valid = hist[:, bh.loc(True), ::sum]
a_only = hist[..., bh.loc("a")]
```

Above, we create three axes. The second axis is a boolean axes, which hold a valid/invalid bool flag. The third axis holds some sort of string-based category, which could label datasets, for example. We then fill this in one shot. Then, we can select the histograms that we might have originally filled separately, like the `all_valid` histogram, which is a 1D histogram that contains all labels and all events where `valid=True`. In the second selection, `a_only`, a 2D histogram is returned that consists of all the events labeled with `"a"`.

This way of thinking can radically change how you design for a problem. Instead of running a series of histograms over a piece of data every time you want a new selection, you can build a large histogram that contains all the information you want, prebinned and ready to select. This combination of multiple histograms and later selecting or summing along axes is a close parallel to the way Pandas combines multiple NumPy arrays in a single DataFrame using columns, allowing you to group and select from the full set.

Distributing

Building a Python library designed to work absolutely anywhere on a C++14 code base provided several challenges. Binding for boost-histogram is accomplished with PyBind11 [PyBind], and all Boost dependencies are included via git submodules and header-only, so a compatible compiler is the only requirement for building if a binary is not available. Serialization, which optionally depends on the non-header only Boost.Serialization, was redesigned to work on top of Python tuple picking in PyBind11 reusing the same interface internally in Boost.Histogram (one of the many benefits of a close collaboration with the original author).

The first phase of wheel building was a custom set of shareable YAML template files for Azure DevOps. This tool, azure-wheel-helpers[6], became the basis for building several other projects in Scikit-HEP, including the iMinuit fitter[7] and the new Awkward 1.0 [Awkward]. Building a custom wheel production from scratch is somewhat involved; and since boost-histogram is expected to support Python 2.7 until after the first LTS release, it had to include Python 2.7 builds, which make the process even more convoluted. To get C++14 support in manylinux1, a custom docker repository (`skhep/manylinuxgcc`[8]) was developed with GCC 9. The azure-wheel-helpers repository is a good place to look for anyone wishing to learn about wheel building, but recently boost-histogram moved to a better solution.

6. https://github.com/scikit-hep/azure-wheel-helpers
7. https://github.com/scikit-hep/iminuit
8. https://github.com/scikit-hep/manylinuxgcc

As the cibuildwheel [CIBW] project matured, boost-histogram became the first Scikit-HEP azure-wheel-helpers project to migrate over. Several of the special cases that were originally supported in boost-histogram are now supported by cibuildwheel, and it allows a custom docker image, so the modified manylinux1 image is available as well. This has freed us from lock-in to a particular CI provider; boost-histogram now uses GitHub Actions for everything except ARM and Power PC builds, which are done on Travis CI. This greatly simplified the release process. The scikit-hep.org developer pages now have extensive tutorials for new developers, including setting up wheels; much of that work was inspired by boost-histogram.

An extremely important resource for HEP is Conda; many of our projects (such as CERN's ROOT toolkit) cannot reasonably (at least yet) be distributed by pip. Scikit-HEP has a large number of packages in conda-forge; and boost-histogram is also available there, including ARM and PowerPC builds. Only Python 2.7 on Windows is excluded due to conda-forge policies on using extra SDKs with Python.

Conclusion and Plans

The future for histogramming in Python is bright. At least three more projects are being developed on top or using boost-histogram. **Hist**[9] is a histogram front-end for analysts, much like Pandas is to NumPy, it is intended to make plotting, statistics, file IO, and more simple and easy; a Google Summer of Code student is working on that. One feature of note is named axes; you can assign names to axes and then fill and index by name. Conversions between histogram libraries, such as the HEP-specific ROOT toolkit and file format are being developed in **Aghast**[10]. The **mplhep**[11] library is making common plot styles and types for HEP easy to make, including plots with histograms. The **scikit-hep-tutorials**[12] project is beginning to show how the different pieces of Scikit-HEP packages work together, and one of the first tutorials shows boost-histogram and Aghast. And a new library, **histoprint**[13], is being reviewed for including in Scikit-HEP to print up to five histograms at a time on the command line, either from ROOT or boost-histogram.

An example of mplhep and boost-histogram interaction is shown in Figure 5:

```
import mplhelp
mplhep.histplot(hist)
```

We hope that more libraries will be interested in building on top of boost-histogram. It was designed to be a powerful back-end for any front-end, with Hist planned as the reference front-end implementation. The high performance, excellent flexibility, and universal availability make an ideal choice for any toolkit.

In conclusion, boost-histogram provides a powerful abstraction for histograms as a collection of axes with an accumulator-backed storage. Filling and manipulating histograms is simple and natural, while being highly performant. In the future, Scikit-HEP is rapidly building on this foundation and we expect other libraries may want to build on this as well. At the same time, Boost.Histogram in C++ is continuously improved and expanded with new features,

9. https://github.com/scikit-hep/hist
10. https://github.com/scikit-hep/aghast
11. https://github.com/scikit-hep/mplhep
12. https://github.com/scikit-hep/scikit-hep-tutorials
13. https://github.com/scikit-hep/histoprint

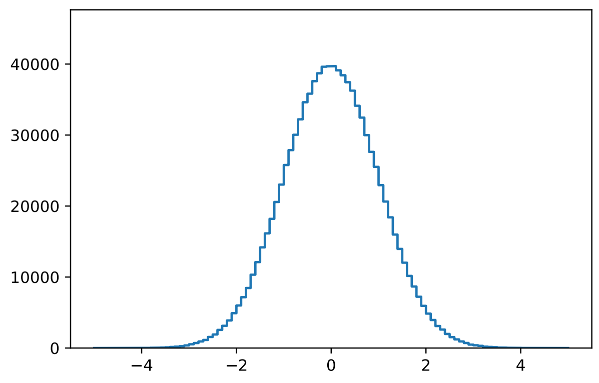

Fig. 5: *An example of a 1D plot with mplhep. It is not completely trivial to get a proper "skyline" histogram plot from Matplotlib with prebinned data, while here it is simple.*

from which boost-histogram benefits nearly automatically. The shared code-base with C++ allows Python to profit, while boost-histogram in C++ is profiting from ideas feed back from Python, creating a win-win situation for all parties.

Acknowledgements

Support for this work was provided by the National Science Foundation cooperative agreement OAC-1836650 (IRIS-HEP) and OAC-1450377 (DIANA/HEP).

References

[Pandas] Wes McKinney. *Data Structures for Statistical Computing in Python*, Proceedings of the 9th Python in Science Conference, 51-56 (2010), DOI:10.25080/Majora-92bf1922-00a

[NumPy] Stéfan van der Walt, S. Chris Colbert and Gaël Varoquaux. *The NumPy Array: A Structure for Efficient Numerical Computation*, Computing in Science & Engineering, vol. 13, 22-30 (2011), DOI:10.1109/MCSE.2011.37

[iMinuit] *iminuit — A Python interface to Minuit*, https://github.com/scikit-hep/iminuit

[Matplotlib] J. D. Hunter. *Matplotlib: A 2D graphics environment*, Computing in Science & Engineering, vol. 9, no. 3, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[ROOT] Rene Brun and Fons Rademakers *ROOT — An Object Oriented Data Analysis Framework* Nucl. Inst. & Meth. A, vol. 386, no. 1, 81-86 (1997), DOI:10.1016/S0168-9002(97)00048-X

[Boost] *The Boost Software Libraries*, https://www.boost.org

[SciPy] Pauli Virtanen et al. *SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python*, Nature Methods, in press. DOI:10.1038/s41592-019-0686-2

[IRIS-HEP] *Institute for Research and Innovation in Software for High Energy Physics*, https://iris-hep.org

[SkHEP] Eduardo Rodrigues. *The Scikit-HEP Project*, EPJ Web Conf. **214** 06005 (2019), DOI:10.1051/epjconf/201921406005

[Neu74] A. Neumaier. *Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen*, Zeitschrift für Angewandte Mathematik und Mechanik (1974), DOI:10.1002/zamm.19740540106

[Numba] Siu Kwan Lam, Antoine Pitrou, Stanley Seibert. *Numba: a LLVM-based Python JIT compiler*, LLVM '15: Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC, 7, 1-6 (2015), DOI:10.1145/2833157.2833162

[LMFIT] Matthew Newville et al. *LMFIT: Non-Linear Least-Square Minimization and Curve-Fitting for Python*, Zenodo (2020), DOI:10.5281/zenodo.3814709

[PyBind] Wenzel Jakob, Jason Rhinelander, Dean Moldovan. *pybind11 -- Seamless operability between C++11 and Python*, https://github.com/pybind/pybind11

[Awkward] Jim Pivarski, Peter Elmer, David Lange. *Awkward Arrays in Python, C++, and Numba* Preprint arXiv:2001.06307

[CIBW] Joe Rickerby et al. *cibuildwheel*, https://github.com/joerick/cibuildwheel